PYTHON YOLU

Proqramlaşdırma öyrənmək üçün sənə nə 600 bal,
nə də bahalı noutbuk lazım deyil, sadəcə yola çıx
və get!

OpenTechnology

Əziz Nadirov

MÜNDƏRİCAT

Ön Söz

Xoş gördük, əziz oxucu! Bu kiçik paraqraf yazıda sənə kitab barədə qısa məlumat verib, diləklərimi yazacam.

Düşünürəm ki biz çox maraqlı bir dönəmdə yaşayırıq - istər geosiyasi , istərsə də texniki-tərəqqi nöqteyi cəhətdən. Kitabın adından da göründüyü kimi, burada bizi daha çox məhz texniki məqamlar izləyəcək. "süni intellekt işimizi əlimizdən alacaq", "qiyam qaldıracaq" və sairə kimi nağılların fonunda, bu kitabı iki ildən çox bir müddətdən sonra bitirib sənə təqdim edirəm. Kitabın əsas məqsədi - heç bir proqramlaşdırma biliyi və təcrübəsi olmayan oxucuya proqramlaşdırmanın əsaslarını, bir az da irəli gedib - hazırda ən çox istifadə olunan proqramlaşdırma dillərindən olan - Python dilini səthi deyil, demək olar ki orta səviyyədə öyrətməkdir. Mən güman edirəm ki sən nə "proqramlaşdırma" nə də "Python" sözü ilə tanışsan, bunu sənə göstərmək, öyrətmək - kitabın vəzifəsi olacaq. Sual verə bilərsiniz - niyə proqramlaşdırma öyrənməliyəm ki? Cavabımı bir neçə aspektdən əsaslandıra bilərəm:

- Bu gəlirlidir: hal-hazırda proqramlaşdırma və bununla əlaqəli sahələr ən çox tələb olunan və qazanc gətirən sahələrdir. Pul hərisi deyiləm, ancaq kim daha rahat yaşamaq istəməz ki...
- İşə başlayandan sonra "rahat işləyə" bilərsən: iş prosesinin özü adətən hər hansı torpağa, buruğa və ya sənədlər dolabına bağlı olmadığı üçün, bəzən uzaqdan - möhtərəm internet üzərindən işləmək mümkündür. Ofisdə işləsəz belə çox vaxt dress code kimi məhdudiyyətlərdən azad olursunuz.
- İşiniz zehni və oturaq olduğu üçün, bir masa, noutbuk, elektrik yuvası və bir fincan qəhvə iş üçün tam bəs edir. Yuxarıda ancaq "iş" haqqında danışdım, ancaq şəxsən özüm üçün kəşf etdiyim məqam - mən bir proqramçı, mühəndis kimi - işlək, hər hansı problemi həll edən şeylər hazırlamağı sevirəm. Bu bir mühəndislikdir, sanki körpü inşa edən memar kimi işinizin bəhrəsinə baxıb, qürür duyursunuz və əlbəttə ki bunun müqabilində aldığınız maddi ödənişə baxırsınız :D...

Kitab kimlər üçün faydalı ola bilər: hər kəs üçün, ciddən deyirəm. Hər yaşda, istənilən sahədən bu kitaba göz gəzdirmək olar. Necə deyərlər, bilmək olmaz, bəlkə həyatınızı dəyişəcək bir şey ola bildi(ümid edirəm yaxşı tərəfə). Ola bilsin siz: tələbə, artıq 10 ildir pedaqoji fəaliyyət göstərən ədəbiyyat müəllimi, taksi sürücüsü, neft mühəndisi, tibb işçisi və sairə ola bilərsiniz. Ola bilsin ki proqramlaşdırma ilk öncə sizə bir hobbi, asudə vaxtını maraqlı keçirmək üçün bir vasitə kimi gələ bilər. Yuxarıdakı cümlələrin əsas məqsədi sizi bu kitabı oxumağa sövq etmək deyil, ümumiyyətlə proqramlaşdırmaya bir hobbi kimi göz atmağa dəvət etməkdir, bu kitabı kənara atıb digər Youtube kimi açıq qaynaqlardan da istifadə edə bilərsiniz, mən buna da razıyam.

Kitabda nə var: kitabın I fəsli çox qısa müddətdə məktəbdəki informatika fənni kimi - say sistemləri, proqram və proqramlaşdırma haqqında anlayış yaratmağa çalışır. Daha sonra, Python proqramlaşdırma dilini nümunələr ilə sadə anlayışlardan başlayıb, mürəkkəb məqamlara qədər mövzu-bə-mövzu öyrədir. Hər fəslin sonunda tapşırıqlar, kitabın sonunda isə, mümkün həlləri yer alır. Bunun sayəsində tək kitabla az-çox praktiki məşğul olmaq mümkündür. Mövzuları mümkün qədər sadə sadə və anlaşıqlı tərzdə çatdırmağa çalışmışam, nə dərəcədə alınıb - oxucu dəyərləndirəcək.

Paralel olaraq: kitabla yanaşı - Open Technology Azerbaijan Youtube kanalında Python Yolu pleylisti də mövcuddur, kitabdan fərqli olaraq, pleylist bəzi məqamlarda mövzunu daha dolğun əhatə

edir.

Son olaraq bir neçə məsləhət: düşünürəm, çoxları sonu bir nailiyyət olan yola çıxdıqda, bu yolu mümkün qədər tez və rahatlıqla keçmək istəyir və bu təbiidir. Ancaq, faktiki olaraq biz daha səylə çalışaraq bu yolu müəyyən qədər qısalda bilərik. Yolun özü nə qədər çətin olarsa, sonunda bizi gözləyən nailiyyət də bir o qədər böyük olar. Çünki, rahat və qısa yolu çoxları keçə bilər, əziyyət isə, təmkin və əzm tələb edir. Odur ki, ruhdan düşməyib təkrar və təkrar yoxlamaq lazımdır. Ola bilsin ki kitabda hər hansı mövzu anlaşılan deyil, ola bilsin ki heç də dediyim qədər sadə və anlaşıq şəkildə mövzunu açа bilməmişəm, belə olan təqdirdə mövzunu digər bir mənbədən oxumağa və ya baxmağa çalışın. Youtube, məqalələr, Teleqramdakı icmamız, ən sonda GPT-dən soruşmaq olar. İstənilən halda, cavab tapmaq mümkündür.

Daha bir məqam - ingilis dili və tərcümə məsələsidir. Nəinki proqramlaşdırma, bütün İT sahəsinin təsviri dili - ingilis dilidir. Hal-hazırda dilimizdə bu sahədə ədəbiyyat demək olar ki olmadığı üçün(olsa idi bəlkə də bu kitabı yazmazdım), məcburən nəyi tərcümə edib, nəyi olduğu kimi saxlamalı olduğuma qərar verməli olurdum. Tərcümə edilən bütün termin və anlayışları ilk dəfə istifadə hallandırdıqda önündə orijinal - ingilis dilindəki yazılışı verilir. Bəzi terminlərin tərcüməsi artıq var olduğu üçün sual yaratmırlar: loop-dövr, variable-dəyişən. Ancaq, əksər hallarda artıq mövcud olub, oturuşmuş tərcümə tapmaq mümkün olmur: instance-nüsxə, closure-qapanma, documentation-sənədləşmə və sairə. İstənilən halda, əsas məqsəd - kitabı Azərbaycan dilində yazıb, oxucunu xarici mənbələrə mehriban yetişdirmək idi. Buna görə də bütün kod nümunələrində dəyişənlər ingilis dilində adlandırılıb.

Mətnin formatı barədə: kitabda xeyli nümunə kod var. Proqram kodunun rahat oxunması üçün demək olar ki həmişə mono şriftlə yazılır: `nümunə sətr`. Üstəlik, Python dilinin sintaksisinə uyğun şəkildə rənglənmədən istifadə olunub. Bəzi hissələrdə kod və ya ad içərisində dəyişilə bilən hissələr - şablonlar olur. Belə olan halda onlar `< >` içərisində verilir. Məsələn, `hi, <person name>` sətindən belə çıxır ki `<person name>` əvəzinə hər hansı şəxsin adı yazılacaq. Kitab tam hazır olmazdan öncə Linkedin və Telegram kimi platformalarda geridönüşlərin alınması və təkmilləşdirilməsi üçün açıq şəkildə paylaşılmışdı və bu prinsip hələ də keçərlidir. Əgər hər hansı sual və təkliflәriniz varsa, platformamızın icmalarına buyura bilərsiniz.

Kitabdakı Tapşırıqlar: ikinci bölmədən başlayaraq, hər bir mövzunun sonunda tapşırıqlar verilir. Tapşırıqların həllini aşağıdakı linkdən görmək olar. Hər bir bölmənin tapşırıqları ayrıca Python Noutbuku kimi verilib:

https://github.com/AzizNadirov/python-road-book/blob/master/code/tasks/

Daha rahat olması üçün, kitabın tapşırıqları üçün, Open Technology platformamızda ayrıca forum açmışıq. Burada, tapşırıqlara baxa, müzakirə edə və əlbəttə ki sual verə bilərsiniz. https://open-tech.co/forum/pyroad/f/tasks

Müəllifdən

Mən - Əziz Nadirov orta məktəbi kiçik kənd məktəbində, bakalavrı Gəncədə, magistraturanı isə Sumqayıtda bitirmişəm. Python dilinin tədrisinə demək olar ki elə Bakalavr pilləsində başlamışdım və hazırda, kitabı bitirdiyim an 1 il proqram təminatı mühəndisi və 3 il Data Scientist iş təcrübəm var. Proqramlaşdırma öyrəndiyim müddət ərzində Frontend proqramlaşdırmadan tutmuş, Java, C++ dillərinə qədər yol keçmişdim, bunun sayəsində "insan istəsə, hər şeyi özü öyrənə bilər" fikrinə gəlib, bir fakt kimi qəbul etmişəm. Əlinizdə tutduğunuz kitab - proqramlaşdırmanı öz dilimizdə, tam pulsuz və ümid edirəm anlaşılan tərzdə çatdırmaq cəhdimdir.

Open Technology:

Open Technology platforması və bu platformanın məhsulu olan bu kitab ölkəmizdə texnoloji ədəbiyyat və kommunikasiyası qıtlığını az da olsa aradan qaldırmaq yaradılıb. Əsas məqsəd - hər kəsin rahatlıqla, pulsuz, öz dilində təcrübəsini bölüşə biləcəyi məkan yaratmaqdır. Platformanın manifesti: https://open-tech.co/guidelines#community

Kontakt və mənbələr:

- Open Technology web saytı: https://open-tech.co/
- Kitaba həsr olunmuş məlumat-forum səhifəsi: https://open-tech.co/forum/pyroad
- Youtube Kanalımız: http://www.youtube.com/@otechaz
- Telegram səhifəsi: https://t.me/otech_az
- Azərbaycan Python İstifadəçiləri İcması: https://t.me/azepug
- Müəllif: https://www.linkedin.com/in/aziz-nadirov/
- Kitabın GitHub Reposu(nümunə kodlar və tapşırıq həlləri): https://github.com/AzizNadirov/python-road-book

İnformatikanın əsasları

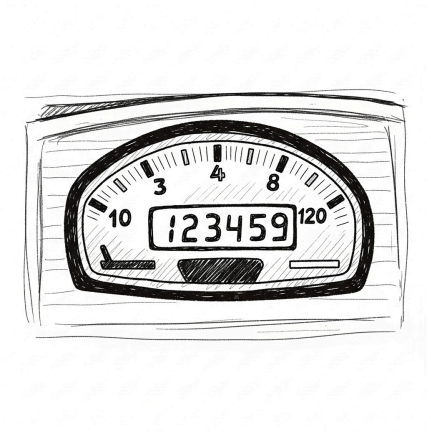
> **İnfo**
>
> Kitab boyu proqramlaşdırma ilə məşğul olacağıq. Proqramlaşdırma özü isə, ümumilikdə Kompüter Elmləri adlı(ingiliscə Computer Science) elmin tərkib sahəsidir. Bu sahə bizə İnformatika adı ilə tanışdır. Bu kiçik fəsildə sizə informatikanın əsaslarından danışıb, sizin üçün nəzəri bünövrə yaratmağa çalışacam. Bu fəsli oxumaq zəruri deyil, arada buraxıb birbaşa növbəti fəslə də keçə bilərsiniz. Ancaq, bu bölmə - kitab boyu öyrənəcəyiniz məvhumun anlamı barədə daha geniş təəssürat yaradacaq.

İnformatika digər elmlərə nəzərən gənc elmdir. Bəziləri bu elmin ilk kompüterin yaradılması ilə yarandığını deyir. Kitab boyu qəliz tərif və "əzbərvari" yanaşmadan uzaq durmuşam(çalışmışam). Elə isə gəlin görək informatika nədir. **İnformatika** – informasiya proseslərinin avtomatlaşdırılması üsullarını öyrənir. Bu tərifi başa düşmək üçün bir neçə suala cavab tapmalıyıq: informasiya nədir, informasiya prosesləri nədir, avtomatlaşdırmaq nədir. İnformasiya – ətraf mühitdən aldığımız məlumatı nəzərdə tutlur. Məsələn, əlinizi pəncərənin şüşəsinə toxunduraraq, onun temperaturunu bilə bilərsiniz. Siz bu informasiyanı toxunuş vasitəsilə almış oldunuz. Şüşə soyuq olduğu üçün "analitik yolla" yəni "düşünərək nəticə çıxartmaq yolu" ilə belə qənaətə gəlirik ki, çöldə temperatur aşağıdır, bu da başqa bir informasiyadır. İnformasiya kompüterdə də var. Bu informasiyaya ən bariz misal – kompüterdəki video, musiqi, şəkilləri və sairə misal çəkmək olar. Kompüterdəki bu informasiya - **verilən** (*data*) adlanır. Verilənlər barədə hələ danışacağıq. İnformasiya prosesləri – informasiyanın(verilənlərin) toplanması, saxlanılması, emalı və ötürülməsi prosesidir. Bu proseslərin hər biri vacibdir. İnformasiyadan necəsə yararlanmaq istəyirsinizsə, əvvəlcə onu toplamaq lazımdır. Toplanan informasiyanı əlbəttə ki hardasa və necəsə saxlamaq lazımdır. İnformasiyadan hər hansı fayda, qərar çıxartmaq üçün onu emal etmək, üzərində nələrsə etmək lazımdır. Son olaraq informasiyanı ötürmək də vacib məsələdir. Avtomatlaşdırmaq – hər hansı bir prosesin bir başa insan müdaxiləsi olmadan icrasını təmin etməkdir. Məsələn, əvvəllər maşınqayırma zavodlarında maşınların kabinə hissəsi insan əməyi ilə bir-bir qaynaq edilərək düzəldilirdi. Ancaq sonralar, bu proses avtomatlaşdırılır və bütün bu qaynaq işlərini yerinə yetirən xüsusi dəzgahlar yaradılır və əlbəttə ki istehsal və zaman xərcləri azaldılır. Daha bir misal: bir neçə on il əvvəl, mühasibatçılar gəlir və xərcləri tək-tək hesablayaraq kağız üzərində sənədləşdirirdilər. İndi isə, bu proseslər avtomatlaşdırılıb və bütün hesab-saxlama əməliyyatları avtonom(müdaxiləsiz, özü-özünə) şəkildə və olduqca sürətli şəkildə yerinə yetirir. İnformatika elminin əsasən nə olduğunu bildikdən sonra tam sürətlə davam edə bilərik.

0.1. Verilənlərin yaddaşda təsviri

Say sistemləri

Desək ki informatikanın, hətta bütün texniki tərəqqinin arxasında riyaziyyat və onun əsas özünüifadə vasitəsi olan ədədlər dayanır - zərrə qədər olsa belə şişirtmərik. İnsanlar hələ mağarada yaşayarkən nələrisə saymağa, bu sayı necəsə təsvir etməyə çalışırdılar. Ədədlərin məğzi də məhz nəyinsə miqdari-ölçüsünü göstərməkdir. Bilirik ki masa üzərində iki ədəd alma varsa - bu dörd ədəd almadan iki dəfə azdır. Biz ədədləri sayaraq kəşf etmişik. Buna görə də ədədləri təsvir etmək üçün istifadə etdiyimiz məntiqi sistem - say sistemi adlanır. Biz sayarkən onluq say sistemindən istifadə edirik. Niyə onluq, çünki, sayarkən istifadə etdiyimiz ədəd - on rəqəmdən ibarət ola bilər: `{0;1;2;3;4;5;6;7;8;9}` . Yəni, biz ədədləri bu rəqəmlər vasitəsilə təsvir edirik. Bizim üçün onluq say sistemi əlbəttə ki rahatdır, çünki uşaqlıqdan onları öyrənirik – uzun – uzadı vurma cədvəlləri və s. İndi isə təsəvvür edin ki yadplanetli ilə danışmaq sizə nəsib olub(detallarına qədər təsəvvür etmək lazım deyil ☺) və siz ona saymağı öyrətməlisiniz. Əlinizdə bir barmaq tutub 1 rəqəmini, sonra iki barmaq tutub 2 rəqəmini və beləcə bütün 10 rəqəmi beləcə göstərirsiniz. İndi də əsas məsələ - on barmaq tutaraq 10 ədədini başa salmalısınız. Niyə 1 və 0 yazdıq, sonra isə 11 və s?



Yəqin ki maşınların odometrlərini görmüsünüz - adətən spidometrin aşaöı hissəsində yerləşir və maşının nə qədər məsafə qət etdiyini göstərir. Yuxarıdakı şəkil də bir növ odometrə bənzəyir və bizə say sistemini başa düşməyə kömək olacaq. Ən sağdakı qırmızı rəqəmə fikir verin. Şəkildə ən sağdakı rəqəm - 9-dur. Bu göstəricini 1 vahid artırsaq, ən sağdakı rəqəm(yəni 9) - 0 olacaq, ondan əvvəldə dayanan rəqəm isə 1 vahid artacaq. Bu ona görə baş verir ki, bizim sayları təsvir etmək üçün istifadə etdiyimiz ən böyük, ən axırıncı simvol, rəqəm - 9-dur. Ən sağdakı xana(gəlin bunu mərtəbə adlandıraq) öz maksimal dəyərini aşdıqdan sonra sıfırlanır və özündən öncəki mərtəbəni 1 vahid artırır. Biz on ədəd simvoldan istifadə etdiyimiz üçün hər mərtəbədə maksimum on dənə rəqəmdən birini göstərə biliri, yəni 0-9 qədər. Əgər bizim ancaq iki mərtəbəmiz olsa(yuxarıdakı odometrin ancaq iki xanası olsun) o zaman maksimum `99` yaza bilərik, yəni `00-99` arsında yüz dənə ədəddən birini yaza bilərik. Əslində burda laqorifmlərlə bağlı çox qəşəng riyazi məsələ çıxır. Deyək ki say sistemimizin əsasını ifadə edən `x` var və bizim üçün `x=10` olsun.

- əgər ancaq bir mərtəbəmiz olsa, ora `x` , yəni 10 sayda fərqli rəqəm yaza bilərik, yəni 0-9 arasında. 1 mərtəbə üçün 1^x sayda, yəni 10 etdi.

- əgər iki mərtəbəmiz olarsa, ora yüz dənə fərqli rəqəmdən birini yazmaq olar, bu `00-99` aralığı deməkdir. Eyni qayda ilə iki mərtəbə üçün 2^x sayda, yəni 100 alırıq.
- üç mərtəbəyə 000-999, yəni min dənə fərqli rəqəmdən birini yaza bilərik, riyazi formula da bunu təsdiqləyir: $3^x = 1000$.

Mən demişdim ki, qəliz riyaziyyat olmayacaq, əmin edirəm ki bu riyaziyyatımızın ən qəliz kimi gələ biləcək nümunəsi idi və bitdi. Bunu harasa yazıb və ya əzbərləməyinizi istəmirəm, sadəcə bir az düşünün vəssalam.

İkilik say sistemi.

İndi isə, təsəvvür edin ki on yox, ancaq iki rəqəmdən - 0 və 1 istifadə edə bilərik. Bu zaman hər mərtəbə ancaq iki qiymətdən birini ala bilər 0-1. Burada da eyni prinsipdir: 000, 001, 010, ...

Ədəd	Onluq say sistemində	ikilik say sistemində
sıfır	0	0
bir	1	1
iki	2	10
üç	3	11
dörd	4	100
beş	5	101
altı	6	110
yeddi	7	111
səkkiz	8	1000
doqquz	9	1001
on	10	1010

Əslində, düşünsək - 2-lik say sistemi daha rahatdır. Niyə - çünki, cəmisi ikicə rəqəmdən istifadə edirsiniz. Düzdür rəqəm sayı çox olur, səkkiz ədədi 10-luqda 8 kimi yazırıqsa, 2-lik say sistemində 1000 kimi yazmalı olarıq. Onluq və ikilikdən başqa, səkkizlik və on altılıq say sistemlərindən istifadə olunur. Səkkizlik say sistemində {0 - 7}, on altılıq say sistemində isə {0 – 9 və A,B,C,D,E,F} istifadə olunur.

Binar kodlaşdırma

Kompüterin yaddaşındakı bütün verilənlər - ədədlər şəklində kodlaşdırılır. Sizcə, bu ədədlər kompüterin yaddaşında hansı say sistemi ilə təsvir edilir? Bəli, ikilik say sistemi ilə. İkilik say sisteminə həmçinin binar sistem də deyilir. Kodlaşdırmanın binar şəkildə olmasının iki əsas səbəbi var.

1. Kompüterin element bazası olan inteqral sxemin yaddaşı ilə bağlı. Siz bu yaddaşı nömrələnmiş çoxlu sayda yuvacıqlar kimi təsəvvür edə bilərsiniz. Kompüterin işləməsi üçün elektrik enerjisi lazımdır. Elektrik cərəyanı isə, elektron adlı kiçik hissəciklərdən ibarətdir(əslində onların hərəkətindən). Baxın, həmin yuvacıqda elektron varsa - 1 deməkdir, yoxdursa 0. Yəni, tək bir

elektronun hardasa olub-olmaması ilə biz istənilən sayda məlumat kodlaşdıra bilərik, biz adi insanlar isə, sadə ədədləri təsvir etmək üçün on rəqəmdən istifadə edirik.

2. Yaddaşa qənaət üçün. Biz kompüterdə hər hansı proqramla işləyərkən, həmin proqram üçün əvvəlcədən yaddaş ayrılır. Məsələn, tutaq ki, proqram yaddaşa bir ədəd daxil edəcək və bu ədəd yaddaş yuvacıqlarına yazılmalıdır. Ədəd üçün yaddaş öncədən ayrıldığı üçün, ədədin hər bir rəqəmini simvolunu nəzərə almaq lazımdır. Axı kompüter öncədən necə bilə bilər ki, siz 1 daxil edəcəksiniz ya 9? Ona görə də kompüter ən pisinə hazır olub, ən böyük rəqəm üçün, yəni - 9 üçün yer ayıracaq. Biz yuxarıda demişdik ki, yaddaş yuvacıqlarında məlumat binar formada saxlanılır, çünki vəziyyət elekronun o yuvacıqda olub-olmaması ilə təyin olunur. İndi təsəvvür edin ki tək bir rəqəmi təsvir etmək üçün həmin rəqəmin maksimal qiyməti üçün yer ayrılır. Yəni əgər söhbət onluq say sistemindən getsəydi, hər rəqəmə 9 rəqəmi qədər yer ayrılmalı olardı bu isə dörd yuvacıq edərdi(binar sistemdə $9_{10}=1001_2$. Bu isə yaddaş israfı olduğu üçün, ən optimalı – binar sistemdən istifadə olunur.

0.2. Alqoritmlər

> **Info**
>
> Bu bölmədə alqoritm, onun təsvir üsulları, xassələri və keyfiyyət göstəriciləri ilə tanış olacağıq. Ümümiyyətlə, alqoritm ilə bütün kitab boyu işləyəcəyik. Ona görə də bu bölmədə səthi, daha çox nəzəri əsaslar göstərilib.

Alqoritm anlayışı

Gündəlik həyatımızdsa əksər hərəkətlərimiz müəyyən, dəqiq ardıcıllığa malik olur. Məsələn, zəng edərkən əvvəlcə smartfonun telefon kitabçasına daxil olub, lazımi kontaktı seçib zəng et düyməsini seçirsiniz. Və yaxud, hər səhərki standart davranışlarımız: oyanmaq, əl-üz yumaq, diş fırçalamaq, qurulanmaq, səhər yeməyi yemək və i.a. Beləliklə, alqoritm – dəqiq, sonlu sayda əməllər ardıcıllığıdır. Düzgün qurulmuş alqoritmlə həll olunan məsələ(söhbət heç də ancaq riyazi məsələlərdən getmir) bizim həm vaxtımıza, həm də resurslarımıza qənaət edə bilər. Elə isə gəlin görək alqoritmin hansı xassələri var və necə qurula bilər.

Alqoritmin əsas xassələri

- *Müəyyənlik* – alqoritmin hər bir addımı dəqiq, birmənalı olmalıdır. Təsəvvür edin, kombi almısınız və üzərində əlbəttə ki, quraşdırılma haqqında təlimat – instruksiya var. Bu təlimatı açıb bilmək istəyirsiniz ki, bunu necə quraşdırmalısınız, ilk öncə nə edilməlidir. Təlimatı açdıqda isə görürsünüz: "Kombini quraşdırıb, pultda qırmızı düyməni sıxın". Bu zaman haqlı sual yaranır: bəs dəqiq demək olmazmı – mən bunu dəqiq necə quraşdırım. Buna görə də, alqoritmin icrası zamanı heç bir sual yaranmaması üçün bütün addımlar dəqiq olmalıdır.
- *Kütləvilik* – alqoritm elə qurulmalıdır ki, həlli gərəkən məsələyə bənzər digər məsələlərə də tətbiq edilə bilsin və mümkün qədər anlaşılan olsun. Çünki, əksər hallarda alqoritmlər başqa adamlar tərəfindən təkrar istifadə olunur. Məsələn, tutaq ki, sizə hər hansı siyahıdakı ədədləri böyükdən kiçiyə doğru çeşidləmək lazımdır. Bunun üçün artıq bir neçə hazır alqoritmin olduğunu bilərək, o hazır alqoritmlərdən istifadə etmək daha sərfəli olmazmı? Ancaq bunun üçün əlbəttə ki, həmin alqoritmi anlamaq, başa düşmək lazımdır, bilmədiyimiz bir şeyi necə tətbiq edə bilərik ki. Bax buna görə də alqoritm anlaşılan və təkrar istifadəyə yararlı şəkildə tərtib edilməlidir.
- *Nəticəvilik* – alqoritmin addımları sonlu sayda olmalıdır, yəni alqoritmdəki bütün addımları yerinə yetirdikdən sonra dəqiq nəticə olmalıdır.
- *Diskretlik* – hər bir alqoritm dəqiq, ardıcıl addımlardan ibarət olmalıdır. Belə olan halda, həm alqoritmin tərtibi, həm də icrası asanlaşır.

Alqoritmin təsvir üsulları

Alqoritmi tərtib etmək və digərlərinə çatdırmaq üçün onu necəsə təsvir etmək lazımdır. Bunun üçün bir sıra üsullar var:

- *Verbal təsvir* – adətən məişətdə, gündəlik həyatımızda istifadə etdiyimiz üsuldur. Daha çox – sadə alqoritmlərin təsviri üçün uyğun gəlir. Təsəvvür edin ki, şəhərdə kimdənsə filan parkın yerini soruşursunuz və həmin adam sizə həmin parka gedən yolu sözlə təsvir edir.

- *Blok-sxemlə təsvir* – bu üsuldan daha çox təhsil müəsisələrində alqoritm anlayışını çatdırmaq istifadə edirlər. Bu zaman hər bir növ əməli müəyyən bir fiqur ifadə edir. Gəlin bu fiqurarlara baxaq:

Başlanğıc və ya son		Prosesin başlanmasını və ya sonunu bildirir
Proses (hesab) bloku		Verilənlərin qiymətini, təsvir formasını və ya yerləşməsini dəyişən əməliyyat və ya əməliyyatlar qrupu
Daxiletmə bloku		Verilənlərin kompüterə daxil edilməsi
Şərt (seçmə, məntiqi blok)		Müəyyən şərtdən asılı olaraq hesablama istiqamətinin seçilməsi
Dövr bloku – Modifikasiya		Dövri strukturlu alqoritmlərin təsviri
Çapetmə		Nəticələrin kağıza köçürülməsi, çıxış bloku
Altalqoritm və ya altproqram		Əvvəlcədən hazırlanmış alqoritmlərdən (altalqoritm) və ya proqramlardan (altproqram) istifadə olunması

- *Alqoritmik dil ilə təsvir* – bu zaman hər hansı proqramlaşdırma dilindən və yaxud proqramlaşdırma dilinin sadələçdirilmiş forması olan psevdokod – dan istifadə olunur. Praktikada bu üsul daha çox istifadə edilir. Çünki, tətbiqi cəhətdən daha rahat qavranılır.

Alqoritmin keyfiyyət göstəriciləri və yaxud yaxşı alqoritmi necə seçməli

Alqoritmin keyfiyyətindən danışdıqda əsasən onun iki xüsusiyyətinə görə mühakimə edirlər:

1. İcra üçün lazım olan resurs(kompüter üçün - yaddaş)
2. Məsələnin həlli üçün tələb olunan zaman. Alqoritmin kompüterdə icrası zamanı diqqət yetirmək lazımdır – "bu məsələnin həlli üçün doğurdanmı bu qədər yaddaş lazımdır, yoxsa hardasa israfçılığa yol vermişik". Düzdür, hal-hazırda kompüterlərin yaddaş tutumu yaxın keçmişə nəzərən xeyli çoxdur, ancaq yenə də nəzərə alsaq ki, bir proqram yaradılarkən orada xeyli sayda əməliyyatlardan - alqoritmlərdən istifadə olunur, başa düşmək lazımdır ki, "dama-dama göl olur". Ona görə də alqoritmin keyfiyyəti barədə düşündükdə, mütləq onun tələb ediyi yaddaşa da nəzər salmaq lazımdır. Daha bir önəmli məsələ isə, alqoritmin icrası üçün lazım olan vaxtdır. Kompüter olduqca güclü, ağıllı bir qurğudur, onun üçün 99x99 ifadəsini həll etməyə 1 milli saniyə bəs edər(1 milli saniyə - qısaca ms = 1/1000 saniyə). Ancaq düzgün qurulmamış ən sadə alqoritm belə, kompüterinizi saatlarla "düşünməyə" məcbur edə bilər. Gəlin, problemin mahiyyətinə daha yaxşı varmaq üçün iki ədəd axtarış alqoritminə baxaq. Bu alqoritmlərin məqsədi – verilən çeşidlənmiş siyahıdan lazım olan elementi tapmaqdır. Tutaq ki, 1000 ədəddən ibarət çeşidlənmiş bir siyahımız var. Biz elə bir alqoritm tərtib etməliyik ki, bu siyahıdan bizə lazım olan ədədi tapsın.

- *Birinci alqoritm* – sadə axtarış. Adından da göründüyü kimi, bu alqoritm sadəcə siyahının əvvəlindən axırına qədər bütün ədədlərə baxaraq bizə lazım olan ədədi axtarır. Belə olan halda, əgər siyahımız 1000 ədəddən ibarətdirsə və bədbəxtlikdən bizə lazım olan ədəd ən sonda yerləşirsə, onda həmin ədədi tapmaq üçün alqoritm bütün 1000 ədədin hamısına "baxmalıdır". Bu isə, 1000 dəfədir! Yaxşı, sual yarana bilər: niyə axı ədəd məhz siyahının sonunda olmalıdır ki, bəlkə elə siyahının əvvəlində oldu və tez tapıldı? İş orasındadır ki, əksər hallarda, alqoritmin tələb etdiyi vaxtı hesablamaq üçün, məhz "ən pis haldan" istifadə edirlər. Necə deyərlər, "ən yaxşısına ümid et, ən pisinə hazır ol". Alqoritmin siyahıdakı ədədi yoxlaması – "baxması" bir

əməliyyat hesab olunur. Belə çıxır ki, bizim sadə axtarış alqoritmimiz – 1000 ədədlik siyahı üçün 1000 əməliyyat(ən pis halda) icra edir. Yəni, n elementlik siyahı üçün n əməliyyat.

- İkinci alqoritm – binar axtarış. Təsəvvür edin ki, sizinlə belə bir oyun oynayırıq: mən fikrimdə hansısa aralıqda bir ədəd tuturam, siz isə həmin ədədi tapmağa çalışırsınız. Siz hər dəfə yanlış ədəd dedikdə, mən sizə fikrimdə tutduğum ədədin siz dediyiniz ədəddən böyük və ya kiçik olduğunu deyirəm. Aydın məsələdir ki, gec-tez fikrimdə tutduğum ədədi tapacaqsız. Məsələ burasındadır ki, bunu necə daha tez etmək olar? Tutaq ki, mən fikrimdə 1-100 aralığında ədəd tutmuşam, olsun 40. Siz belə edə bilərsiniz.
  - Siz: 50?
  - Mən: ondan kiçik.
  - Siz: 25?
  - Mən: ondan böyük.
  - Siz: 37?
  - Mən: ondan böyük.
  - Siz: 43?
  - Mən: ondan kiçik.
  - Siz: 40?
  - Mən: bəli ! Məntiqi hiss edirsiniz? 100 dənə ədədin içərisindən lazım olanı cəmisi 5 cəhdə tapdıq. Necə - hər dəfə yarıya bölməklə. Hər dəfə ədədlərin yarısını kənara ataraq, aralığı sıxmış oluruq.
- ədəd 1-100 aralığındadır. Görəsən ədəd 100-ün yarısı – yəni 50-dən böyükdür, kiçikdir, ya elə 50-dir? Bunun üçün 50-ni götürürük, əgər bizə lazım olan ədəddən (40-dan) böyükdürsə, deməli ədəd 1-50 arasında yerləşir. Bir dəfəyə 50-100 aralığını qırağa atmış olduq. Daha 100 ədədin içində deyil, 50 ədədin içində axtarmış oluruq.
- 1-50 aralığının ortasını, yəni 25 götürürük və analoji yolla lazım olan ədədin 25-dən böyük və ya kiçik olduğunu müəyyən edirik. Bundan sonra cəmisi 25 ədədimiz qalır. Beləcə bölə-bölə, ədədi tapana qədər davam edirik. Sadə axtarış alqoritminə nəzərən daha sərfəlidir, elə deyilmi? Binar axtarış alqoritmi isə, ən pis halda belə n elementli siyahı üçün $\log_2 n$ qədər əməliyyat icra edir. Qısaca log nədir: log – loqarifmaya bəzən qüvvətin əksi də deyirlər, ancaq bir qədər fərqlidir. Qısası, $\log_2(8) = 3$. Çünki, $2^3=8$. Ümumiyyətlə, əgər $k^n=m$ olarsa, onda $\log_k(m)=n$ olar.

Ümumiləşdirək: ən pis halda 1000 elementlik siyahıda sadə axtarış üçün – 1000 əməliyyat, binar axtarışda isə - 10 əməliyyat($\log_2 1000 \approx 10$) tələb olunacaq. Hər əməliyyatın bir milli saniyə olduğunu yadımıza salsaq, birinci halda bizə 100 ms, ikinci halsa isə, 10 ms lazım olacaq. O qədər də böyük fərq deyik, eləmi? Amma 1000 də böyük rəqəm deyil. Gəlin 1000.000.000 – bir milyardlıq siyahı üçün əməliyyat sayını ölçək. Sadə axtarış üçün bu elə 1 milyard əməliyyat edəcək. Yəni 1 milyard milli saniyə. 1 milyard ms=11 gün! 11 günə başa gələn axtarış. İndi isə binar axtarışın 1 milyardlıq siyahı üçün tələb olunan vaxta baxaq. Bunun üçün, az öncə öyrəndiyimiz loqarifm düsturundan istifadə edək: $\log_2 1000\_000\_000 \sim= 30$. Yəni, 30 əməliyyat, 30 millisaniyə! 11 günə qarşı 30 ms, bax fərq buna deyərəm!

> ⚠ **Diqqət!**
>
> Binar axtarış alqoritmi ancaq və ancaq çeşidlənmiş siyahılar üçün keçərlidir!

Gördüyünüz kimi, zaman böyük rol oynayır. Biz alqoritmin sürəti barədə danışarkən "əməliyyat sayı" və millisaniyə anlayışlarından istifadə edirdik. Praktikada ən düzgün yolu məhz - əməliyyat sayıdır. Əməliyyat sayı – alqoritmin ən pis halda neçə addıma(əməliyyata) nəticəni alacağını, sona çatacağını göstərir. Bunu simvolik olaraq göstərmək üçün O – "böyük O" anlayışından istifadə olunur. `O(n)` kimi yazılır və burdakı n – alqoritmin icrası üçün lazım olan əməllər sayıdır. Məsələn, öncəki sadə axtarış alqoritmi üçün bu - `O(n)` bərabər idi, yəni içərisində axtarış etidiyimiz siyahının element sayı nə qədər olarsa, bir o qədər də əməliyyat yerinə yetiriləcək. Binar axtarışda isə bu $O(\log_2 n)$ bərabərdir. Alqoritmlərin sürətindən asılı olaraq, `n` müxtəif cür olur. `n` nə qədər kiçik olarsa, alqoritm bir o qədər sürətli olar.

Bu bölmədə alqoritm anlayışı ilə tanış oldunuz. Alqoritm sizə yalnız texniki məsələlərdə deyil, eləcə də şəxsi həyatınızda kömək ola bilər. Belə ki, ən kritik, həyəcanlı hallarda belə, "soyuq başla", məntiqli - alqoritmik düşünərək çətinlikdən çıxış yolu düşünə bilərsiniz.

0.3. Proqram təminatı

> **Info**
>
> Bu bölmədə proqram təminatı nədir, nəyə xidmət edir, o cümlədən, translyatorlar barədə danışacağıq. Siz burada – proqram təminatının klassifikasiyası(sistem, tətbiqi və s.) barədə heç nə görməyəcəksiniz. Çünki, praktikada bu mövzu olduqca nisbi və əksər hallarda yalnız xüsusi imtahanlar üçün əzbərlənir. Biz isə bu bölmədə, daha çox real tətbiqi fəaliyyətimiz üçün nəzəri bünövrə qurmağa çalışırıq.

Proqram təminatı

Kompüterinizin aparat təminatı ilə tanışsınız: yaddaş qurğuları, monitor, klaviatura, siçan və sairə. Siz bunları görə, toxuna bilirsiniz. Onlar sərtdirlər – metal, plastik maddələrdən hazırlanıblar. Ona görə də terminaloji olaraq: "hardware" hərfi tərcümədə - "sərt məmulat" anlamını verir. Proqram təminatı ilə isə vəziyyət bir qədər fərqlidir. Biz proqramı əlimizə ala, toxuna bilmirik. Proqram özlüyündə yaddaşın yuvacıqlarında yerləşdirilmiş elektronlar kombinasiyasından başqa bir şey deyil, prosessora ünvanlanan əmrlər toplusudur. Siz proqranı işə salan piktoqramı kliklәyirsiniz və proqram icra olunur. Monitorda gördüyünüz hər şey, müəyyən proqramın icrasının nəticəsidir. Oynadığınız oyun, internet üzərində veb saytları gəzdiyiniz brauzer, kompüteri işə saldıqda ilk olaraq ekranda görünən animasiya – bunlar hamısı proqramdır. Gördüyünüz kimi, proqramlar olmasa, kompüter adi plastik kütləyə çevrilir. Onu da qeyd etmək lazımdır ki, mən şərti olaraq məhz "kompüter" deyirəm. Smartfonlar, TV-lər, soyuducu, tozsoran bunların hamısında proqram var, bu proqram ən azından soyuducunun işini tənzimləyir: nə zaman sönməli, düymәyə basdıqda nə baş verməli və sairə. Hətta o əfsanəvi "qozqıran" Nokia-lar belə proqramlara sahib idilər – məşhur ilan oyununu yada salsaq bəsdir. Bir sözlə - proqramlar hər yerdədir. Onlar demək olar ki, həyatımızın ən müxtəlif sahələrinə təsir edirlər. Smartfonunuza baxanda nə qədər proqramlar – applikasiyalar (applications – tətbiqlər) görürsüz, Play Store və yaxud App Store – da milyonlarla applikasiya var. Bu gün əksəriyyətinin istifadə etdiyi ChatGPT kimi süni intellekt tətbiqləri də proqramdır, onların arxasında proqramlaşdırılmış riyazi modellər dayanır.

Translyatorlar və proqramlaşdırma

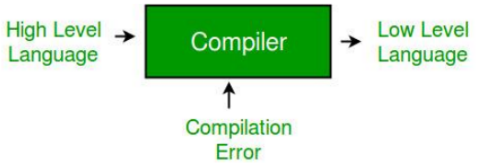
Binar kodlaşdırma sistemi barədə danışarkən qeyd etmişdim ki, prosessor binar sistem əsasında çalışır. Deməli, prosessora gedən bütün əmrlər binar şəkildə - 0 və 1-lər formasında olmalıdır. Belə çıxır ki, proqramlar da əslində əmrlər toplusu olduqları üçün, onlar da 0 və 1 şəklində olmalıdırlar? Əgər hə, o zaman başdan ayağa 0 və 1 –lərdən ibarət olan proqramı necə hazırlayırlar, kiçik ədədləri rahatlıqla ikilik say sisteminə keçirə bilərik, amma minlərlə əmri – milyonlarla 0 və 1 kombinasiyaları ilə işləmək real iş deyil axı... Bəli, real iş deyil və heç kim bu işlə məşğul olmur. Çünki, əslində rahatlıq üçün prosessor ilə proqram arasında *translyator* adlanan başqa bir proqram var. Biz proqramı təşkil edəcək əmrləri başqa bir dildə - ingilis dilindəki sözlər və xüsusi simvollardan istifadə olunan proqramlaşdırma dillərində yazırıq. Proqramlaşdırma dilində yazılmış əmrlərə, kodlara – mənbə kodu (source code) deyilir. Mənbə kodu icra ediləndə translyatora ötürülür. Translyator isə, öz növbəsində mənbə kodunu maşın dilinə, başqa sözlə maşın koduna(object code) çevirir və ötürür prosessora. Günümüzdə yüzlərlə proqramlaşdırma dili var. Demək olar ki, hər bir proqramlaşdırma dilinin öz translyatoru var. Translyatorun işi qısaca -

proqramıaşdırma dilində yazılmış kodu, prosessorun başa düşdüyü maşın koduna tərcümə etməkdir.

Translyatorların növləri.

Translyatorın mənbə kodunu tərcümə etmə ardıcıllıqlarına əsasən iki növü vardır: *kompilyator* (compiler) və *interpretator* (interpreter).

Kompilyator – mənbə kodunu "bir həmləyə" maşın koduna çevirir, proses - kompilyasiya adlanır. Təsəvvür edin, bir səhifə kod yazmısınız, verirsiniz kompilyatora. Kompilyator səhifəni əvvəldən – axıra qədər oxuyur, yoxlayır və səhv tapmasa tərcümə edib göndərir prosessora.



İnterpretator – mənbə kodunu sətir – sətir maşın dilinə çevirir, proses – interpretasiya adlanır. Yenə təsəvvür edin, bir səhifə kod yazmısınız və icra edirsiniz. Bu bir səhifə keçir interpretatorun əlinə. İnterpretator birinci sətri oxuyur, səhv yoxdursa maşın koduna çevirib göndərir prosessora. Sonra – ikinci sətri yoxlayır, səhv yoxdursa onu da tərcümə edib göndərir prosessora. Beləcə sona qədər. Ortaya belə bir sual çıxır: bəs bunlardan hansı daha yaxşıdı? Bu suala müəyyən mənada cavab vermək üçün ilk öncə iki məqama diqqət yetirək:

1. Yazdığımız mənbə kodunu translyatora verdikdən sonra nəticəsini görmək üçün nə qədər gözləməli olacağıq?
   - kompilyator üçün gərək gözləyək ki, kompilyator tam olaraq bütün kodu oxuyub tərcümə etsin. Yalnız bundan sonra, xəta tapılmasa mənbə kodumuz tərcümə olunub prosessora göndəriləcək.
   - interpretator üçün isə o qədər gözləməyə ehtiyac yoxdur. Mənbə kodunun ilk sətrini oxuyan kimi, səhv tapılmasa tərcümə edib ötürür prosessora və biz birinci sətrin nəticəsini görmüş oluruq. Daha sonra ikinci, üçüncü və beləcə sona qədər. Yəni, mənbə kodunu icra edən kimi, nəticəsini görmüş oluruq. Bu məqamda üstünlük interpretatorun tərəfindədir. interpretator:1, kompilyator : 0 !
2. Bunlardan hansında tərcümə olunan proqram daha sürətlə icra ediləcək: kompilyator ya interpretator?
   - kompilyator üçün gözləməyə ehtiyac yoxdur. Çünki, mənbə kodu kompilyasiya edilən kimi, tam olaraq prosessora ötürülür. Yəni, prosessor bir dəfəyə bütöv mənbə kodunu alıb icra edir.
   - interpretator isə, mənbə kodunu prosessora sətr – sətr göndərir. Yəni prosessor bir dəfəyə bütöv mənbə kodunu almır. Ona görə də fasilələr edir. Bu məqamda üstünlük kompilyator tərəfindədir və hesab 1:1 olur. Nəticə etibarilə, hər iki translyator yaxşıdır, ancaq hərəsinin öz müsbət və mənfi tərəfi var. İnterpretatorda - proqramın hazırlanması(mənbə kodunun

yazılması, test edilməsi və s.) prosesi sürətli gedir. Ancaq, proqram ləng icra olunur. Kompilyator isə tam əksinə - proqramın hazırlanması ləng, icra edilməsi isə, sürətli baş verir.

Hibrid translyatorlar. Günümüzdə yuxarıdakı mənfi cəhətləri minimallaşdırmaq üçün hibrid – qarışıq translyatorlardan istifadə edirlər. Bu translyatorlar müxtəlif texniki fəndlərin köməyi ilə sürət itkisini azaldırlar. Buraya müxtəlif virtual maşınlar, tərcümə olunmuş maşın kodunun növbəti dəfəki icra üçün yaddaşda saxlanması və sairə daxildir. Növbəti bölmədə python proqramlaşdırma dili barədə oxuyanda bu mövzu ilə təkrar qarşılaşacaqsınız.

Proqramlaşdırma dillərinin səviyyələri. Proqramlaşdırma dillərini insan dilinə yaxınlıqlarına görə yuxarı və aşağı səviyyəli olmaqla iki qrupa ayırırlar. Aşağı səviyyəli dillərə - assemblerlər(ilk proqramlaşdırma dilləri – maşın koduna yaxındır) daxildir. Yuxarı səviyyəli dillərə isə müasir proqramlaşdırma dilləri – Python, C++, Java, Ruby, Java Script və sairə aiddir. Əslində bu səviyyə məsələsi şərti xarakter daşıyır. Məsələn, Python Java-dan, Java isə C++ dilindən yuxarı səviyyəli hesab olunur. Bu dillərinin hər birinin öz geniş tətbiq sahəsi var və birinin digərindən aşağı səviyyəli olması, onun daha az tələbata malik olması demək deyil. Biz isə kitabda yolumuza Python proqramlaşdırma dili ilə davam edəcəyik. Amma bu barədə növbəti bölmədə!

0.4. Məntiq Cəbrinin Əsasları

> **Info**
>
> Günümüzdə kompüter, TV və digər elektron qurğuların işləməsi - Məntiq cəbrinin və ya Bul Cəbrinin tövhəsidir. Bundan başqa, biz bu əməllərdən proqramlaşdırmada, bitlərlə müxtəlif çevirmələr aparanda istifadə edəcəyik. Bu bölmədə riyazi məntiqin “riyazi dərinlikləri”nə enmədən, əsas anlayış və əməllərindən danışılacaq.

Yəqin ki Aristotel, Sokrat, Platon və s. qədim yunan filosoflarının adlarını eşitmisiniz. Onlar fəlsəfə, riyaziyyat, tibb, məntiq və sair mövzularda xeyli təməl biliklər vermişlər. Bizi hal-hazırda maraqlandıran məntiqdir. Məntiq – sadə və qısa desək, düşünmə formasıdır. Hər hansı mülahizənin doğru və ya yanlış olması barədə fikir söyləmək məhz məntiqdən irəli gəlir. Yaxşı, bəs mülahizə nədir? Mülahizə - barəsində doğru və yanlış deyə biləcəyimiz hər hansı fikirdir. Məsələn, “Yer kürə formasındadır”. Burada konket bir fikir – mülahizə ortaya qoyulur və biz onun doğru və ya yanlış olması barədə fikir söyləyə bilərik. Gəlin, bir neçə mülahizə üçün doğru və yanlış olması haqqda mühakimə yürüdək. Ancaq, hər dəfə doğru və yanlış sözlərini yazmamaq üçün onları 1 – doğru və 0 – yanlış kimi işarə edək.

Mülahizə	Mühakimə
Yaşıl yarpaq yaşıl rəngdədir.	1
1+1=1	0
11 > 10.5	1
Yer kürə formasındadır.	1

Yəqin ki, hiss etdiniz, elə bil ki nə isə düz gəlmir. Sanki, mülahizələrimizdə çatışmazlıqlar var. Lap götürək ilk “Yarpaq yaşıl rəngdədir” mülahizəmizi. Bu mülahizəni doğru kimi qəbul etdik, bəs heç dəqiqləşdirildimi – söhbət hansı fəsildən, hansı bitkinin yarpağından gedir. Yarpaq yaz, yay – yaşıl, payızda, qışda – sarı olur. Bundan başqa, heç də bütün bitkilərin yarpaqları yaşıl olmur. Burdan belə nəticəyə gələ bilərik ki, hər hansı mülahizənin doğru və ya yanlışlığı – baxılan məsələnin aid olduğu sistemə, topluya nəzərən təyin olunur. Biz “Yer kürə formasındadır” dedikdə düşünməliyik. İlk zamanlar yerin yastı olduğunu düşünürdülər, daha sonra aydın oldu ki kürə formasındadır. İnsan oğlu kosmosa ayaq açandan sonra bəlli oldu ki, yer heç də ideal kürə formasında deyil – qütblərdən bir qədər basılıdır(səbəb – qütblərdən təsir edən maqnit sahəsi). Bundan başqa, söhbət əgər planetin ancaq quru hissəsindən gedərsə, yəni okean çökəklikləri, dağları nəzərə alsaq, ümumiyyətlə planetin daha əndrabadi forma aldığını müşahidə etmək olar. Məntiq cəbrində mülahizə yalnız iki hökmdən birini ala bilər: doğru(1) və ya yanlış(0). Bu bir növ riyazi funksiya(lar)dır. Parametr kimi n ədəd parametr alır və nəticəsi parametrlərdən (və yaxud funksiyanın nə iş gördüyündən, nə funksiyası olduğundan) asılı olaraq 0 və ya 1 qaytarır. Təsəvvür edin ki bir *f* funksiyası var. Bu funksiya iki parametrlidir – *f(x, y)* deyək. Burada *x* və *y* parametrlərdir. Bundan başqa bu parametrlərin hər biri uyğun olaraq iki qiymətdən birini ala bilər: `0` və ya `1`. Bu funksiya nəticə olaraq da ancaq 0 və ya 1 qaytara bilər. Funksiya ancaq iki qiymətli olduğu üçün asanlıqla(parametrlərin sayı az olduqda) funksiyanın hansı parametrləri ala biləcəyi və hansı nəticəni qaytaracağını yazmaq, cədvəl formasında tərtib etmək olar. Baxaq:

x	y	f(x,y)
0	0	0 və ya 1
0	1	0 və ya 1
1	0	0 və ya 1
1	1	0 və ya 1

Biz bu 2 parametrli *f* funksiyası ilə cəmisi 16 funksiya yaza bilərik. Funksiya sayı = $2^{2^n} = 2^4 = 16$. Gəlin əsas funksiyalara baxaq:

- VƏ - yalnız hər iki parametr doğru olduqda doğru qiymətini qaytarır. Buna həmçinin məntiqi vurma, *konyuksiya* da deyilir. • , ∩ işarələrindən də istifadə edilir. Belə başa düşmək olar: sizə deyirlər mağazadan a və b al. Belə olan halda siz hər ikisini – həm a, həm də b-ni almalısınız ki doğru olsun. Yəni a və b hər ikisi ödənməlidir – doğru olmalıdır. Əgər ancaq ikisindən birini və yaxud heç birini alsanız, onda şərti ödəməmiş olursunuz və yenidən mağazaya yollanırsınız.

x	y	və
0	0	0
0	1	0
1	0	0
1	1	1

- VƏ YA – parametrlərdən biri və ya hər ikisi doğru olduqda doğru qiymətini qaytarır. Buna həmçinin məntiqi toplama, *dizyuksiya* da deyilir. +, ∪ ilə də işarə olunur. Yenə, təsəvvür edin ki sizə deyilir: mağazadan a və ya b al. Yəni belə çıxır ki, bunlardan birini alsanız bəs edir. Yaxşı, bəs hər ikisini almış olsaz? Bədxərclik olsa da şərti ödəmiş olur, yəni hər iki parametr(şərt) doğru olduqda belə doğru qaytarır.

x	y	və ya
0	0	0
0	1	1
1	0	1
1	1	1

- **İnkaredici VƏ YA** və ya **XOR** – parametrlərdən yalnız və yalnız biri doğru olduqda doğru qiymətini alır. Adətən XOR kimi işarə edilir. Yəni “və ya” barədə çəkdiyimiz məsələdə XOR əməli üçün hər ikisini – həm a, həm də b almış olsaydıq yanlış hərəkət etmiş olardıq, çünki bizə XOR üçün uyğun olaraq yalnız ikisindən birini almaq tələb olunurdu, hər ikisini deyil.

x	y	x XOR y
0	0	0
0	1	1
1	0	1
1	1	0

- **İmplikasiya** – “ilkin şərtə uyğun olaraq son şərt qənaətinə gəlmək” kimi başa düşə bilərik. Simvolik olaraq “→” və ya “=>” kimi işarə edilir. Gəlin cədvəli araşdıraq:

	x	y	x → y
(1)	0	0	1
(2)	0	1	1
(3)	1	0	0
(4)	1	1	1

“Çöldə yağış yağır” → “çöldə torpaq yaşdır”. Burda x = “Çöldə yağış yağır” – ilkşərt, y = “çöldə torpaq yaşdır” – isə sonşərt adlandıraq. (4) Əgər ilkşərt və sonşərt doğrudursa(hər ikisi 1- ə bərabərdirsə), deməli implikasiyamız – “nəticə çıxartmağımız” doğrudur. Doğurdan da, əgər çöldə yağış yağırsa və biz qənaətə gəlsək ki, torpaq yaşdır – deməli düzgün düşünmüşük. Qısası, doğru ilə başlayıb, doğru ilə bitirmişiksə, deməli hər şey okay. (3) Əgər ilkşərtimiz doğru, sonşərtimiz isə yanlışdırsa, implikasiyamız yanlışdır. Niyə: “çöldə yağış yağır” doğrudursa və biz “çöldə torpaq yaşdır” sonşərtinin düzgün olmaması qənaətinə gəlmişiksə, deməli nəsə səhv etmişik. Qısası, doğru ilə başlayıb, yanlışla bitirmişiksə, səhv etmişik. (2) Əgər ilkşərtimiz yanlış, sonşərtimiz isə doğru olsa, implikasiyamız doğru olmuş olar. Bir qədər yanlış səslənsə də belədir. Bu halı əksər hallarda riyazi bərabərliklərlə izah edirlər: tutaq ki `2+2=5` deyək və bu olsun ilkşərt – gördüyünüz kimi yanlışdır. Tranzitivliyə görə(toplananların yerini dəyişəndə, cəm dəyişmir qaydası) `2+2=5` əvəzinə `5=2+2` yaza bilərik. Hər bir ədədin özü-özünə bərabər olması taptalogiyasından irəli gələrək(taptalogiya – həmişə doğru olan mülahizədir), 5-in yerinə elə `2+2` yaza biilərik. Yəni, `2+2=2+2` ! Doğrudur, elə deyilmi? Yalan üstündə doğru mülahizə yürütdük. (1) Əgər ilkşərt və sonşərt yanlış olarsa, implikasiyamız doğrudur. Baxaq: “çöldə yağış yağır” yanlışdır(yağış yağmır) və “çöldə torpaq yaşdır” da yanlışdır(torpaq qurudur). Qısası, yanlışla başlayıb, elə yanlışla da bitirdiksə, bizlik bir şey yoxdur, biz işimi düzgün yerinə yetirmişik.

- **Ekvivalentlik** – adi bildiyimiz “bərabərlik” deməkdir. “ ≡ ” kimi də işarə edirlər. Yəni doğru = doğru, yanlış = yanlış. Başqa sözlə, əgər hər ikisi eynidirsə - doğru, fərqlidirsə - yanlış.

x	y	x ≡ y
0	0	1
0	1	0
1	0	0
1	1	1

- **Anti-ekvivalentlik** – fərqlidir deməkdir. Yəni ekvivalentliyin əksidir, “ `≠` ” kimi işarə olunur.

x	y	x ≡ y
0	0	0
0	1	1
1	0	1
1	1	0

İndi isə, gəlin bütün cədvəlləri birləşdirək və baxaq:

x	y	x ● y	x + y	x XOR y	x → y	x ≡ y	x ≠ y
0	0	0	0	0	1	1	0
0	1	0	1	1	1	0	1
1	0	0	1	1	0	0	1
1	1	1	1	0	1	1	0

Gördüyünüz kimi, XOR ilə anti-ekvivalentlik eynidir.

- **İnkar**. Əvvəlki əməllər hamısı binar, yəni ikilik əməllər(funksiyalar) idi. Çünki, iki parametr tələb edirdi. Məsələn, məntiqi tiplamada(VƏ YA) iki ədəd parametr tələb olunur, olsun a və b. Onda toplama üçün a + b yaza bilirik. İnkar isə unar – yəni tək parametrlidir(bir funksiya kimi baxsaq). Yalnız bir parametr alır və onu inkar edir. Yəni yanlışı – doğru, doğrunu isə yanlış edir. Adətən üstdən xətt ilə işarə edilir. Əgər ifadə A olarsa, onun inkarı – $\overline{A}$ olar.

x	$\overline{x}$
0	1
1	0

Əməllərin prioritet(üstünlük) ardıcıllığı

Bilirsiz ki, riyaziyyatda dörd əsas əməl(toplama, çıxma, vurma və bölmə) xüsusi üstünlük dərəcəsinə malikdir. Əvvəlcə vurma və ya bölmə, daha sonra isə uyğun olaraq soldan-sağa toplama və ya çıxma əməli yerinə yetirilir. O məşhur “2+2x2” misalını yadınıza salın. Yuxarıda tanış olduğumuz məntiqi əməllərin də o cür üstünlük dərəcələri var. Onları azalan sıra ilə(ən yüksəyindən, ən kiçiyinə doğru) göstərək: `Mötərizələr -> İnkar -> Dizyuksiya -> Konyuksiya -> İmplikasiya -> Ekvivalensiya`

> ⚠ **Warning**
>
> Öncə də dediyim kimi, bu bölmədə öyrəniləni mütləqa əzbərləmək, üstündə ilişib qalmaq məcburiyyətində deyilsiniz. Gələcəkdə pythonda məntiqi tiplər haqqında danışanda lazım olan hissələri yenidən təkrarən izah edəcəm. Ancaq, bu bölmədən həyati vacib biliklər çıxarda

bilərik - məntiq və məntiqə söykənən mülahizələr yürütmək. Yaşadığımız dönəmdə ən qənimətli dəyər - informasiyadır. Biz isə burnunu hər şeyə soxan, daima yeni informasiya, fikir və təəssüratlar axtaran varlıqlarıq. Odur ki, bu "axtarışlarımızda" olduqca ehtiyatlı və soyuqqanlı olmalıyıq. Hər bir təklifdə, hər bir baxdığımız reklam və ya çağırışda bizə "doğru" kimi qəbul etdirmək istədikləri, satmaq istədiqkləri fikirlər var. Bu fikirlər, ideyalar arasında bəzən bir ovuc buğda dənələri arasında gizlənmiş çınqıl kimi gizlədilmiş "alt-fikirlər" ola bilər. Odur ki, məntiqinizlə dost olun, düşünün və unutmayın ki insanları qiymətləndirən zaman 0 və 1 arasında bir kainat ədəd dayanır.

Python Yolu

✓ **Təbriklər!**

Və budur! Biz artıq proqramlaşdırma öyrənməyə başlayırıq! Kompüter önündə qeyd dəftərçəsi və bir fincan isti qəhvə ilə oturub qollarınızı çərmələyə bilərsiniz. Bu bölmə kitabın əsasını təşkil edir və həm nəzəri, həm də praktiki materiallarla zəngindir. İstisna deyil ki, bəzi məqamlarda yazılanı ilk dəfədən anlamaq alınmayacaq. Nə qədər sadə və ətraflı yazmağa çalışsam da bu heç də hər zaman alınmır. Bəs o zaman sən necə davranmalısan - bir dəfə də asta-asta oxu, başa düşə bilmədiyin hissələrə bax, bir az "google etməyə" çalış və ya bir başa bizim python qrupumuza yazıb soruş. Amma, əsla təslim olma! Eşq olsun!

1. Python barədə ümumi məlumat. Yüklənməsi və quraşdırılması

1.1. Niyə məhz Python

Kitabın əsasən – Python ilə proqramlaşdırmaya həsr olunub. Belə bir sual yarana bilər: bu qədər adlı – sanlı proqramlaşdırma dilləri ola-ola, niyə məhz python? Çox uzatmadan bir neçə paraqrafla ümumi mənzərəni anlatmağa çalışacam. Beləliklə Python:

- **Rahat başa düşüləndir**: Python ilə yazılan proqram kodu daha aydın və başa düşüləndir. Bunun başqa adı – "readable", mənası – "oxunan" deməkdir. Dilin sintaksisi - əmrlərin yazılış forması elə quruluşa malikdir ki, azacıq belə olsa proqramlaşdırma biliyi olan insan, bu dildə yazılmış kod fraqmentinə baxdıqda bu kodun təqribi də olsa, nə iş gördüyünü anlaya bilir. Üstəlik burada bir çox başqa dillərdəki kimi fiqurlu mötərizələr, nöqtə-vergül istifadə etməyə ehtiyac yoxdur. Bundan başqa, dilin öz "fəlsəfəsi" var(https://peps.python.org/pep-0020/). Qısası – bu fəlsəfəyə görə qısa, əndrabadi koddansa, bir az uzun, amma, oxunan kod daha yaxşıdır.
- **Dəbdədir**: Python – 1990-cı illərin əvvəllərində Qvido Van Rossum adlı cənnətə layiq bir adam tərəfindən yaradılmışdır. Lakin dil 2010-ların əvvəlində məşhurlaşmağa başlamışdır və son illərdə müxtəlif mənbələrdə top 3-lükdə yer alır. Kitabın yazıldığı vaxt python-ın məşhurluğu nəinki aşağı düşmür, əksinə, yüksəlməyə davam edir. Hal-hazırda dünyanı ağuşuna alan süni intellekt sahəsinin tərtibatında ən çox istifadə edilən dildir.
- **Universaldır**: Python universaldır. Başqa sözlə, seçəcəyiniz istiqamətdən asılı olmayaraq hər halda köməyinizə gələ bilər. Python – web, kiber təhlükəsizlik, sistem və administratorluğu, data analitikası, maşın öyrənimi, hətta, android tətbiqlərin hazırlanmasında istifadə etmək olar. Ümumilikdə isə, demək olar ki, Python skript dilidir. Skript – adətən bir və ya bir neçə fayldan ibarət olan proqram kodudur. Yəni, otur, bir neçə dəqiqəyə yaz, istifadə et. Başlanğıc kimi Python seçməyimin əsas səbəblərindən biri də budur. Çünki, əksər hallarda başlayan zaman, hansı istiqamətdə ixtisaslanacağınızı bilmirsiniz və yaxşı olardı ki, öyrənəcəyiniz dil seçəcəyiniz yöndən asılı olmadan gələcəkdə sizə lazım olsun.
- **Aktivdir**: Python açıq mənbəli məhsuldur. Odur ki, pulsuzdur və hansısa firma və şirkətə bağlı deyil. Dil bu qədər tələbatcıl və açıq mənbəli olduğundan aktiv olaraq inkişaf etdirilir. Demək olar ki, hər ay yeniliklər edilir və xətalar aradan qaldırılır. Ona görə də, aktuallıq baxımından narahat olmağa gərək yoxdur, çünki, python – "həmişəlazımlıdır".

1.2. Bir az Python barədə

Yuxarıda deyildiyi kimi, Python ümumi təyinatlı dildir. Siz ondan müxtəlif məqsədlər üçün yararlana bilərsiniz: kompüterin fayl sistemini idarə etmək - başqalarının əl ilə saatlarla excel ilə etdiyini pythonda bir toxunuşla etmək, saytın məntiqi(backend) hissəsini yığmaq, "xakerlik" etmək, data analtika, riyazi hesablamalar və daha nələr - nələr... Hər şey sizin xəyal gücü və məqsədinizə bağlıdır. Qayıdaq Python-ın texniki tərəfinə. Python interpretə olunan dildir: bunun sayəsində proqram kodunun yaradılma prosesi, sürətlə baş verir - translyatorlardan bildiyimiz kimi. Ancaq digər tərəfdən, python həm də hibrid adlandırmaq olar. Çünki, python ilə yazılmış proqram kodunu ilk dəfə icra edəndə icradan sonra proqramın binar şəkli olan forması yaradılır və saxlanılır. Növbəti dəfə proqram icra edildikdə artıq proqram sətr-sətr tərcümə edilmir, bir- başa binar fayldan icra olunur. Bu isə əlavə sürət bonusu deməkdir. Siz hər dəfə orijinal proqram koduna nəsə dəyişiklik etdikdə proqram kodu tərcümə edilir və binar fayl yenilənir. Ona görə də həmin binar fayl həmişə "aktual" qalır. Python kodunun bu cür binar şəklə çevirən mexanizm – Python Virtual Maşınıdır(Python Virtual Machine - PVM). Bu mexanizm python interpretatorunun tərkib hissəsidir, yəni əlavə nəsə yükləməyə ehtiyac yoxdur. Ancaq bu mexanizm elə qurulub ki, bütün işlər pərdəarxası baş verir və bizim ona müdaxiləmiz gərəksizdir. Ona görə də bu barədə narahat olmağa ehtiyac yoxdur, bilin ki belə bir şey var, vəssalam.

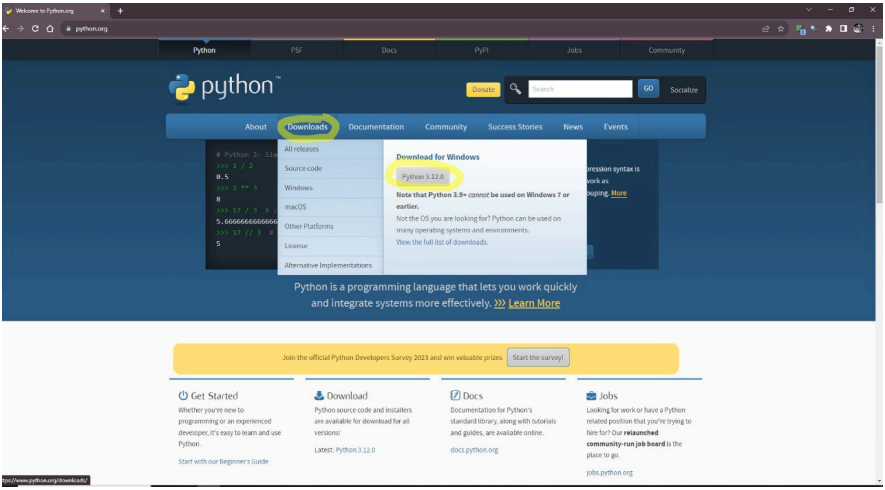
1.3. Python-la harada işləyə bilərəm

Python rəsmən Windows, Macos və Linux əməliyyat sistemləri üçün rahatlıqla əlçatandır. Ancaq bədayaqda, siz ondan smartfonlarınızda da istifadə edə bilərsiniz. Məsələn Android ƏS smartfonlar üçün "Pydroid" adlı tətbiqi məsləhət görə bilərəm. Play Marketdən pulsuz yükləyib istifadə edə bilərsiniz. Linux istifadəçiləri üçün çox zaman python yükləməyə ehtiyac qalmır. Çünki, əksər distribütivlər sistemin işi üçün pythondan istifadə edir, yəni artıq əvvəlcədən artıq python sistemdə quraşdırılmış olur. Amma, düşünmürəm ki sizdə Linux ola, çünki, bu ƏS daha çox təcrübəli İT istifadəçiləri tərəfindən istifadə edilir. Sizdə isə, çoxgümanki, windows ƏS olacaq. Ona görə də kitabın bu hissəsindəki nümunələr bu ƏS üzərindən göstəriləcək. Onu da qeyd edim ki, python – krossplatformalıdır(crossplatform), yəni yazılan proqram kodu – bütün ƏS-lərində eyni cür işləyir.

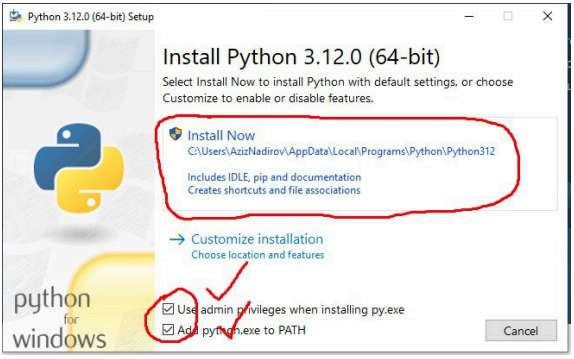
1.4. Yüklənmə və quraşdırılma

Yüklənməsi

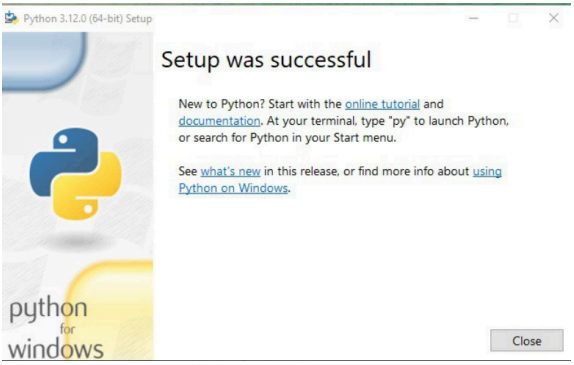
Kompüterə Python yükləmək 2x2 qədər sadədir. İlk olaraq *python.org* səhifəsinə keçirsiniz, `Downloads / Python 3.xx` seçirsiniz. Sayt avtomatik olaraq sizin əməliyyat sisteminizə uyğun ən son versiyanı təklif edir.



Bundan sonra Python quraşdırıcı fayl yüklənir - Windows üçün `.exe` formatlı fayl olacaq. Faylı açdıqda aşağıdakı kimi bir pəncərə açılır, şəkildə olduğu kimi xanalara "quş" qoyub, "Install Now" seçirik:



Ortaya çıxan dialoqda razılaşırıq və sonda mübarək uğur pəncərəsi görürük:

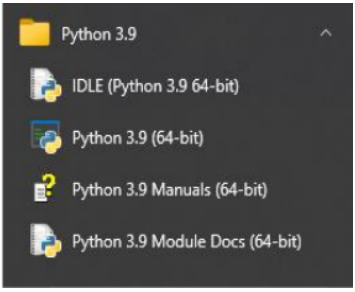


Bununla da python quraşdırılması sona çatdı.

> ⚠ **Diqqət!**
>
> Windows 7 əməliyyat sistemi artıq bir neçə ildir ki rəsmən "ölü" hesab edilir, buna görə də əksər proqram alətləri daha bu ƏS ilə işləmir. Odur ki, kitabda göstərilən kodlarla işləmək üçün daha yeni Windows (Windows 10 və ya 11) tələb olunur. Ola bilər ki bu əməliyyat sisteminə öyrəşmisiniz, amma bəzən gələcəyə doğru getmək üçün, keçmişi buraxmalı olursan...

Hardan və necə işə salım

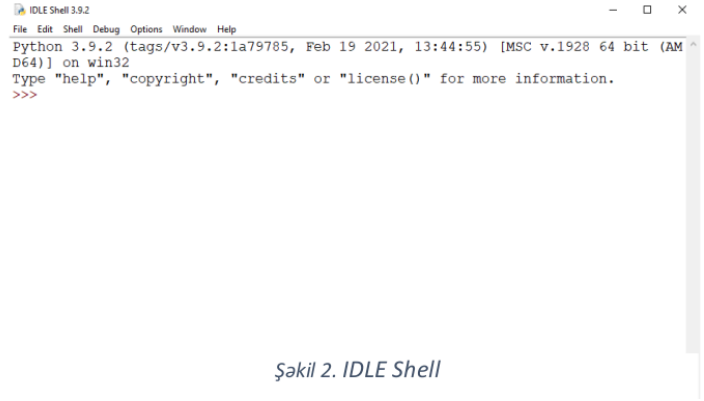
Python işə salmağın çoxlu yolları var(daha dəqiq python interpretatoru). Pythonı quraşdırdıqdan sonra ƏS-nin start menyusunda təqribən belə bir qovluq yaranacaq: Mən kitabı yazan vaxt ən son python versiyası `3.9` idi. Sizdə isə, çox güman ki, daha yeni birisi olacaq. O qədər də fərqi yoxdur ancaq.



Python skript faylı. Şəkildəkiləri anlatmağa başlamazdan əvvəl "python skripti" nə olduğunu bilməliyik. Bilirsiniz ki, faylın genişlənməsi – onun adından sonra nöqtə ilə ayrılan hissəsidir. Məsələn əgər `music.mp3` kimi bir faylınız varsa, faylın adı – music, genişlənməsi isə - `mp3` olacaq. Faylın genişlənməsi – faylın tipini göstərir. Belə ki, `music.mp3` faylına baxan kimi, sonundakı `.mp3` genişlənməsinə görə, bilirik ki fayl - musiqi faylıdır və musiqi pleyerləri ilə açılmalıdır. Kompüter də faylları məhz bu cür – genişlənmələr ilə tanıyır: `mp3` faylı musiqi kimi, `mp4` – video kimi və sairə. Python ilə yazacağımız kod da fayl kimi yaddaşda saxlanıla bilir. Bu faylın genişləməsi – `py` olur və biz bu fayla *skript* deyəcəyik. Əslində skript - adi mətn faylından fərqlənmir. Siz asanlıqla adı mətn faylı yaradıb, içərisinə python dilində kod yazıb, sonra isə faylın genişlənməsini .txt(mətn faylının genişlənməsi) əvəzinə .py yazmaqla mətn faylını python fayla çevirmiş olursuz. Belə çıxır ki, python skriptləri adi notepad ilə də yazmaq olar. Əsas məsələ sonra bu skriptləri icra etdirək, yəni – python interpretatoruna çatdırmaqdır. Yuxarıda gördüyünüz şəkildə dörd ədəd piktoqram görürsünüz: bunlardan son ikisi məlumat və təlimat üçündür.

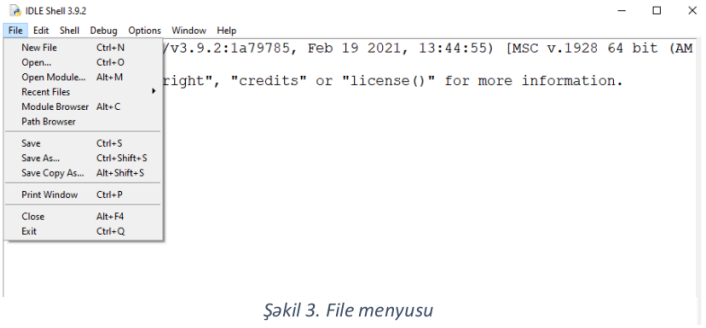
Python IDLE: proqram kodunu yazmaq və skript kimi yadda saxlamaq üçün inteqrallaşdırılmış – uyğunlaşdırılmış mühitdir. Demişdim ki, skriptləri adi notepad ilə də yazmaq olar, amma praktikada belə etmək - nahaq əziyyətdən başqa bir şey deyil. Python IDLE ilə iki rejimdə işləmək olar: *interaktiv* və *adi skript*. Python IDLE ilk dəfə işə saldıqda interaktiv formada işə düşür: bu rejim `IDLE Shell` (aydl şell) adlanır. Shell rejimində proqram kodu sətrbəsətr yazılır və icra edilir. Gördüyünüz kimi, sətrin əvvəlində `>>>` işarələri var. Bu işarələr istifadəçini bir növ kod daxil etməyə dəvət edir.

Hər bir yeni sətr bu simvollarla başlayır. Qısa proqram kodunu sınamaq, skript yaratmadan, sadəcə "yazım və işə salım" məqsədi üçün shell ideal vasitədir. Onu da deyim ki, siftələrimizi bu rejimdə işləyəcəyik. Ona görə də işləyə-işləyə shell ilə dostlaşacaqsınız.



Şəkil 2. IDLE Shell

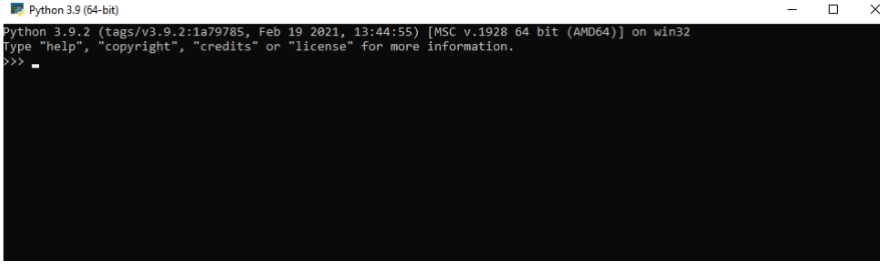
IDLE skript rejimində skriptlər yaratmaq, onları yaddaşda saxlamaq, mövcud olanları açmaq, qısası: yarat- aç-saxla etmək mümkündür. Bunun üçün shell `file` menyusundan lazımi əmri seçmək lazımdır, hər bir əmrin qarşısında qısayolu da verilir:



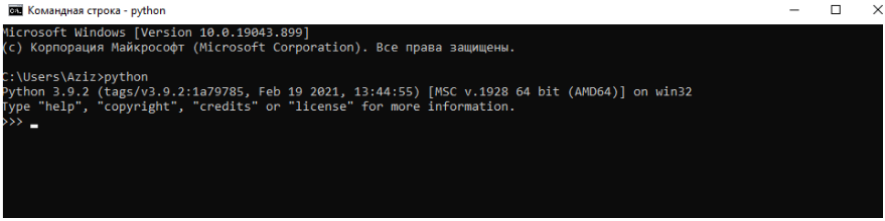
Şəkil 3. File menyusu

• `New file` – yeni skript yaradır. Açılan pəncərədən proqram kodunu daxil edib, `ctrl+S` əmri ilə skripti adlandırıb, saxlanılacağı qovluğu seçib yaddaşda saxlayırsını. • `Open` – mövcud olan skripti açmaq üçündür. Açılan pəncərədən qovluqlar arasından skripti tapıb seçirsiniz.

Hələki, bizə lazım olan bunlardır. aşağıdakı şəkildən gördüyünüz ikinci piktoqram - python interpretatorunun öz piktoqramıdır. İşə salsaz, görərsiz ki, interpretator da interaktiv rejimdə açılır:



Eyni şeyi ƏS-nin əmrlər sətri ilə də etmək olar. Windowsda bu "command prompt" adlanır. İşə salmağın ən qısa yollarından biri `Win+R` ilə Run pəncərəsini çağırmaq və ora " `cmd` " daxil etməkdir. Nəticədə ƏS-nin əmrlər sətri açılır. Buraya "python" yazsaz, eynilə interpretator işə düşəcək.



İlk kodunuzu yazacağınız alətlərlə tanış oldunuz, artıq proqramlaşdırma ilə məşğul olmağa başlaya bilərik.

> **Tip**
>
> Günümüzdə peşəkar proqramlaşdırma ilə məşğul olmaq üçün çoxlu sayda xüsusi mühitlər – IDE-lər mövcudddur. Bu mühitlər xüsusi imkanlara – kod sintaksisinin rənglərlə fərqləndirilməsi, müxtəlif debaq(bu anlayışla gələcəkdə tanış olacağıq) alətləri, ağıllı tamamlamalar və sairə. Aparılmış müxtəlif sorğulara görə, yeni başlayanların belə "bərkgedən" mühitlərdən istifadə etmələri ancaq çaşqınlıqlar yaradır və heç məsləhət görülmür. Belə mühitlərə misal kimi: Pycharm, VS Code, Anaconda və sairə göstərmək olar.

> ⚠ **Diqqət!**
>
> Ola bilsin ki bu bölmədə göstərilən nümunələr sizə bir qədər dayaz görünsün. Bu bölmənin əsas məqsədi – üzərində işləyəcəyimiz, alqoritmlərimiz üçün "material" rolunu oynayacaq verilənlərlə tanış olmaqdır. Növbəti bölmədə sizinlə dövrlər, daha sonra isə funksiyalarla tanış olacağıq. Bütün vaxt ərzində isə, biz bu yeni mövzuları məhz indi tanış olacağınız verilənlər üzərində, nisbətən daha dərin nümunələr üzərində öyrənəcəyik. Odur ki, bu bölmə - sadəcə dəhlizə aparan bir pillədir. Buna rəğmən, çox vacib bir pillədir.

2.1. Verilənin tipi dedikdə

Verilənin tipi – bu verilən üzərində apara biləcəyimiz əməliyyatları təyin edir. Məsələn, tutaq ki, bizə iki ədəd verilib. Biz ədədləri toplaya, çıxa, vura, bölə və sairə əməliyyatlar apara bilərik. Niyə - çünki, ədədlərə bunları etmək mümkündür. Bəs sətr? Məntiqlə yanaşsaq, sətrləri təşkil edən hərfləri saya, hecalara bölə bilərik. Birinci halda verilənin tipi ədəd, ikinci halda isə - sətr oldu. Ümumiyyətlə, verilənlər kompüter elmlərində böyük rol oynayır. Biz alqoritmləri verilənlər üzərində icra edirik. Binar axtarış alqoritmini xatırlayın, biz bu alqoritmdə ədədləri müqaisə edirdik. Biz bu bölmədə verilənlərin tipləri, onlar üzərində bəzi əməliyyatları, tətbiq nümunələri ilə tanış olacağıq. Ümumilikdə isə, bütün kitab boyu bu verilənlər strukturlarından istifadə edəcəyik. Öyrənəcəyimiz verilənlər tiplərindən bəziləri:

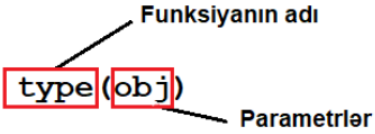
tipin adı	nümunə
ədəd	`9; -1; 0.25; Decimal(0.3); Fraction(1, 3)`
sətr	`"hello world"; 'salam'; 'python is awesome!'`
siyahı	`[1, 2, 3]; ['a', 1, 2, "3"]; [[1,2], [3,4],[5,6]]`
yazı	`(1, 2, 3); ('a', 1, 2, "3"); ([1,2], [3,4],[5,6])`
məntiqi	`True; False; bool(1)`
lüğət	`{'a':1, 'b':2}; dict(a=1, b=2)`
çoxluq	`{1,2,4,6,0}, set([1,2,4,6,0])`

Şərhlər. Şərhlər proqram kodunu təsvir etmək, izah etmək üçün istifadə olunur və proqramın icrasına(işləməsinə) heç bir təsir etmir. Şərhlər bir sətrdən və ya çox sətrdən ibarət ola bilər. Bir sətrli şərhlər üçün `#` simvolundan istifadə olunur və ondan sonra gələn bütün yazılar şərh hesab olunur(ancaq həmin sətr üçün). Çox sətrli şərhlər üçün isə, üçqat dırnaq `'''şərh'''` və ya üçqat apastrof `""" şərh """` istifadə olunur. Şərhlərdən bütün kitab boyu istifadə olunub və kodda baş verənləri "şərh etmək" üçün istifadə olunur.

2.2. Ədəd tipləri

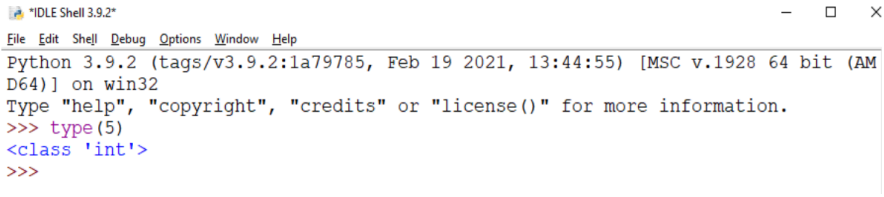
Pythonda ədədlərin 2 əsas tipini ayırd etmək olar: *tam* ədədlər – `integer` və *üzən nöqtəli* ədədlər(onluq kəsr ədədlər) – `float`. Pythonda hər hansı bir obyektin tipinə baxmaq üçün onu `type` funksiyasına ötürmək lazımdır. İndi isə iki şeyə diqqətinizi yönəltmək istəyirəm.

1. Bilirsiz ki, python obyektyönlü proqramlaşdırma dilidir. Bu dildəki hər bir şey bir obyektdir. Biz bu obyektləri binanın tikintisi üçün istifadə etdiyimiz kərpiclərə bənzədə bilərik. Verilənlər, funksiyalar və sairə bunlar hamısı obyektdir.
2. Biz 4-cü bölmədə funksiyalar ilə yaxından tanış olub, öz funksiyalarımızı yazacağıq. Ancaq ona qədər hazır, pythonın öz içərisində olan – qurma (built-in) funksiyalardan istifadə edəcəyik. Buraya `print`, `input`, verilənlərlə müxtəlif manipulyasiyalar aparmaq üçün lazım olan bir sıra funksiyar və əlbəttə ki, yuxarıda adını çəkdiyim `type` funksiyası da daxildir. Bu funksiya parametr kimi tipini bilmək istədiyimiz obyekti alır, nəticə kimi isə onun tipini qaytarır(demək olar ki).



Tam ədədlər

Bu ədədlər adi bildiyimiz tam ədədlərdir – kompüter elmlərində integer kimi adlandırlar. Mənfi və ya müsbət, fərqi yoxdur. Baxaq:



Yuxarıda `type(5)` daxil etdim və `<class 'int'>` kimi bir cavab aldım. Bu cavab içərisində `int` tipin adıdır, int - yəni `integer`. Eyni şey mənfi tam ədədlər üçün də keçərlidir:

```
>>> type(-5)
    <class 'int'>
```

Ədədlər üzərində müxtəlif əməliyyatlar icra etmək olar - toplama, çıxma, vurma, bölmə, qüvvətə yüksəltmə, bölmədən alınan qalıq(mod), bölmədən alınan tam(div) və sairə. Onlara əvvəlcə cədvəldə, daha sonra isə praktikada baxaq:

əməliyyatın adı	işarəsi	nümunə	nəticə
toplama	`+`	`5 + 5`	`10`
çıxma	`-`	`5 - 5`	`0`
vurma	`*`	`5 * 5`	`25`
bölmə	`/`	`5 / 5`	`1`
qüvvətə yüksəltmə	`**`	`2 ** 3`	`8`

əməliyyatın adı	işarəsi	nümunə	nəticə
`mod` - bölmədən alınan qalıq	`%`	`10 % 3`	`1`
`div` - bölmədən alınan tam	`//`	`10 // 3`	`3`

```
>>> 5 + 5
    10
>>> 5 - 5
    0
>>> 5 * 5
    25
>>> 5 / 5
    1.0
>>> 2 ** 3  #  2^3
    8
>>> 10 % 3   # 10 / 3 = 3 tam qalıq 1. Qalığı götürürük
    1
>>> 10 // 3  # 10 / 3 = 3 tam qalıq 1. Tamı hissəni götürürük
    3
```

Əlavə olaraq, `divmod` adlı funksiyanın köməyi ilə ədədlərin `div` və `mod` -nu eyni zamanda almaq olar:

```
>>> divmod(10, 3)   # div və mod
    (3, 1)
```

Üzən nöqtəli ədədlər

Üzən nöqtəli ədədlər bir-birindən nöqtə ilə ayrılmış tam və kəsr hissələrdən ibarətdir. Riyaziyyatda adətən bu cür ayırıcı kimi vergüldən istifadə ediriksə də, əksər proqramlaşdırma dillərində, o cümlədən də pythonda bu rol üçün nöqtədən istifadə olunur. Çünki, vergül özlüyündə ayırıcı rolunu oynayır. Gəlin bir `type` ilə bu ədədlərin tipinə baxaq:

```
>>> type(2.5)
    <class 'float'>
```

Göründüyü kimi, üzən nöqtəli ədədlər `float` (ing float – üzmək) adlanır. Pythonda ədədləri bir-birinə bölərkən nəticə üzən nöqtəli olur.

```
>>> 20 / 5
    # integer / integer = float
    4.0
```

Daha bir neçə nümunəyə baxaq:

```
>>> 2.5 * 2
    5.0
>>> 10.1 + 9.9
```

```
    20.0
>>> 1 / 3
    0.3333333333333333
    # nöqtədən sonra 16 rəqəm dəqiqliyi ilə
```

Ən sondakı uzun ədəd hesablama dəqiqliyindən irəli gəlir. `1/3` nəticəsi irrasional ədəddir, yəni sonsuza qədər uzanır, π və ya e ədədləri kimi. Yuxarıdalı nümunə ilə bağlı 2 hal ola bilər:

1. Sizə bu dərəcədə dəqiq cavab lazım deyil, sizə yalnız nöqtədən sonra ilk iki rəqəm bəs edir. Bunun üçün misalın nəticəsini kəsr hissənin ilk ikinci rəqəminə qədər yuvarlaqlaşdırmalıyıq. Bunu round funksiyası ilə etmək daha rahatdır. Bu funksiya birinci parametr kimi – yuvarlaqlaşdırılmalı olan ədəd(və yaxud nəticəsi bu ədəd olan ifadə), ikinci parametr isə - nöqtədən sonra neçənci rəqəmə qədər yuvarlaqlaşdırılmalı olduğunu bildirən ədədi alır.

```
    >>>round(1/3, 2)
        0.33
    >>>round(0.333333333, 3)
        0.333
```

2. Daha böyük dəqiqlik tələb edən hallarda decimal modulunun `Decimal()` istifadə edə bilərsiniz. Bunun üçün, hər bir ədəd ayrılıqda `Decimal("rəqəm")` kimi yazılmalıdır. Yəni `""` və ya `''` arasında.

```
>>> from decimal import Decimal
>>> Decimal('1') / Decimal('3')
    Decimal('0.33333333333333333333333333333') # 30 rəqəm!
```

Yuxarıda Decimal istifadə etmək üçün onu decimal adlı moduldan import etməli oldum. Modullar barədə ayrıca 2 dəfə danışacam. Düşünün ki, pythonda xeyli sayda hazır funksiyalar filan var və onlar o qədər çoxdurlar ki, onlardan daha rahat istifadə etmək, tapmaq üçün müvafiq qovluqlara çeşidləyiblər. Hər belə bir qovluğa modul deyək(modul heç də qovluq deyil, sadəcə hələ ki, belə qəbul edək).

> **Info**
>
> Xarici ədəbiyyatlarda bəzən tam hissəsi `0` olan üzən nöqtəli ədədlərdə `0` rəqəmi buraxılır. Bunu pythonda da etmək olar və əksər hallarda bu cür edilir də. Məsələn, `0.1` əvəzinə sadəcə `.1` və ya `0.75` əvəzinə `.75` yazmaq. Yəni, belə bir yazılış görsəniz, bilin ki səhv deyil.

Kəsr ədədlər – Fractions

Əlavə olaraq, pythonda kəsr ədədlərlə – nisbətlərlə də rahatlıqla işləmək olar. Bunun üçün `fractions` modulunun `Fraction()` istifadə edə bilərik. `Fraction` iki ədəd parametr qəbul edir: *sürət* və *məxrəc*. Tutaq ki bir $\frac{a}{b}$ kəsri verilib. Burda `a` – kəsrin sürəti, `b` isə məxrəcidir. Onda kəsrimiz `Fraction(a, b)` kimi olacaq.

```
>>>from fractions import Fraction
>>>Fraction(1,3) * Fraction(3,1)  # (1/3) * (3/1)
    Fraction(1,1)                 # 1
>>> Fraction(2,6)
    Fraction(1, 3)
# mümkün olduqda sürət və məxrəc ixtisar olunur.
```

Kəsrləri həmçinin üzən nöqtəli ədədlərdən də almaq olar. Bunun üçün üzən nöqtəli ədədi sətr kimi – `""` daxilində `Fraction` funksiyasına ötürmək lazımdır:

```
>>>Fraction("0.5")    # 0.5 = 5/10 = 1/2
    Fraction(1, 2)
>>>Fraction("1.25")
    Fraction(5, 4)
```

> **Tip**
>
> Kəsr və ədədlər arasında arifmetik əməliyyatlar, qüvvətə yüksəltmə etmək olar. Özünüz sınaya bilərsiniz.

`int` və `float` funksiyaları.

`int` funksiyası verilmiş ədədi tam ədədə - integer tipinə çevirir. Funksiya üzən nöqtəli və kəsr ədədləri tam ədədlərə çevirir. Bu çevrilmədə tipin dəyişilməsi baş verdiyi üçün xarici ədəbiyyatda buna *type casting* deyilir.

```
>>> int(1.5), int(0.01), int(1.99)
    (1, 0, 1)
```

Gördüyünüz kimi `float -> integer` çevrilməsi zamanı `float` ədədin kəsr hissəsi atılır. `int` funksiyası həmçinin sətr kimi verilmiş tam ədədləri ədədlərə çevirir.

```
>>> int('1'), int('3'), int('5')   # '5'-sətr, 5 – isə ədəddir.
    (1, 3, 5)
```

`float` funksiyası da eynilə bu cür işləyir, ancaq tam ədədlərə deyil, üzən nöqtəli ədədlərə çevirir:

```
>>> float(1), float("3.14"), float(33)
    (1.0, 3.14, 33.0)
```

> **Warning**
>
> Sətrdən ədədlərə çevirmə apararkən, sətr çevrilən olmalıdır. Məsələn, `"5"` sətri `5` ədədinə çevrilir. `"5a"` sətri isə çevrilmir. Tip çevirmələri barədə daha detallı gələcəkdə danışacam.

Kompleks və başqa say sistemlərindəki ədədlər

Kompleks ədədlər

Kompleks ədədlər adətən o qədər də geniş istifadə olunmur. Ancaq pythonda baza ədəd tipi kimi mövcuddur. Yazılışı belədir:

```
həqiqi hissə + xəyali hissə.
```

Həqiqi hissə verilmədikdə sıfıra bərabər hesab edilir, xəyali hissənin sonunda isə `j` və ya `J` yazılır:

```
>>>3+2j
(3+2j)
>>>3+2j + 3+2j
(6+4j)
>>>3+2j+2j
(3+4j)
>>>3+2j * 2
(3+4j)
>>>(3+2j) * 2
(6+4j)
```

Xəyali və yaxud həqiqi hissə mənfi işarəli olarsa, eləcə qarşısında `-` yazılmalıdır:

```
>>> 1+-2J
(1-2j)
>>> -3+-5j
(-3-5j)
```

Digər say sistemli ədədlər

Pythonda yalnız onluq ədədlər ilə deyil, həmçinin ikilik, səkkizlik və on altılıq ədədlərlə də işləmək mümkündür. İkilik ədədi yazarkən qarşısında `0b` , səkkizlik üçün `0o` , onaltılıq üçün isə `0x` yazılır:

```
>>> 0o1, 0o20, 0o377   # səkkizlik
(1, 16, 255)
>>> 0x01, 0x10, 0xFF   # onaltılıq
(1, 16, 255)
>>> 0b1, 0b10000, 0b11111111
(1, 16, 255)
```

Göründüyü kimi, python susmaya görə ədədləri onluq say sistemində çap edir. Ancaq əks keçid də mümkündür, yəni, onluqdan müvafiq say sisteminə. Bunun üçün `bin` – ikilik, `oct` – səkkizlik və `hex` – onaltılıq say sistemi üçün funksiyalarından istifadə olunur.

```
>>> bin(16)
'0b10000'
>>> oct(16)
'0o20'
```

```
>>> hex(16)
'0x10'
```

Eyni şeyi `int` funksiyasının köməyi ilə də etmək olar. Belə ki, `int` funksiyası ikinci parametr kimi – birinci parametrdə qəbul edilmiş ədədin say sisteminin əsasını qəbul edir. Bu zaman birinci parametr sətr kimi verilməlidir. Məsələn, `int("100")` bizə `100` qaytaracaq. Ancaq, `int("100", 2)` bizə `4` qaytaracaq. Çünki, ikinci parametrdə 2 ötürməklə, ötürdüyümüz 100 ədədinin onluqda deyil, ikilik say sistemində olduğunu bildirdik. O isə onluqda 4 qaytarır.

```
>>> int("10", 16), int("20", 8), int("10000", 2)
(16, 16, 16)
>>> int("0x10", 16), int("0o20", 8), int("0b10000", 2)
(16, 16, 16)
```

2.3. Dəyişənlər və sətrlər

> **Info**
>
> Bu bölmədə əksər proqramlaşdırma dillərinin ilk öyrədilən mövzusu – dəyişənlər; bundan sonra, buradək adını çəkdiyimiz və qismən istifadə də olsa istifadə etdiyimiz sətrlər, onlar üzərində əməllər, funksiya və metodları barədə öyrənəcəksiniz. Daha sonra, yaxın bölmələrdə məlumatları daxil və ya xaric etmək üçün `input` və `print` funksiyaları barədə ətraflı danışacam.

2.3.1. Dəyişənlər

Dəyişən(ing. *Variable*) sözünü ola bilsin ki riyaziyyat fənnindən xatırlayırsınız. Dəyişənlər olduqca xeyirli bir şeydir. Onların köməyi ilə, bir obyektdən dəfələrlə və rahatlıqla istifadə etmək olar. Sadə dillə desək, dəyişən – obyektə verilən addır. Bu "adlandırma" prosesi sintaktik olaraq belə yazılır: `dəyişənin_adı = dəyər` Buradakı dəyər(ing. *Value*) – adlandırmaq istədiyimiz obyektdir. `=` isə *mənimsətmə* operatorudur. Beləliklə, `a = 5` yazılışı o deməkdir ki, biz `a` adlı dəyişənə `5` dəyərini nənimsədirik, "5-i a kimi adlandırırıq".

Dəyişənin adlandırılması. Dəyişəni adlandırarkən bəzi qaydalara riayət etmək lazımdır. Dəyişənin adı Unicode kodlaşdırma sistemi(demək olar ki bütün əlifbaları əhatə edir, bizim ə, ğ, ç kimi hərflərimizdən tutmuş, Çin heroqriflərinə qədər) hərflərindən, rəqəmlərdən və `_` simvolundan ibarət ola bilər. Ancaq, ilk simvol rəqəm ola bilməz! Bəzi düzgün və yanlış nümunələrə baxaq:

Düzgün	yanlış	niyə
Deyishen	D eyishen	boşluq olmaz
Deyishen0_1	1Deyishen	rəqəmlə başlamaz
Deyishen	De-yishen	- olmaz
_deyishen	Deyi$en	$ olmaz

Bundan başqa, pythonda xüsusi açar sözlər(*keywords*) var ki, onlar python üçün xüsusi məna daşıyır və biz bu sözlərdən dəyişən adı kimi istifadə edə bilmərik. Bu açar sözlərə `keyword` modulu

vasitəsilə baxa bilərik:

```
>>> import keyword
>>> print(keyword.kwlist)
    ['False', 'None', 'True', '__peg_parser__', 'and', 'as',
    'assert', 'async', 'await', 'break', 'class', 'continue',
    'def', 'del', 'elif', 'else','except','finally', 'for',
    'from', 'global', 'if', 'import', 'in', 'is', 'lambda',
    'nonlocal', 'not', 'or', 'pass', 'raise', 'return',
    'try', 'while', 'with', 'yield']
```

Uzatmadan misallara baxaq:

```
>>> a = 5
>>> b = 2
>>> a
5
>>> b
2
>>> a + b
7
>>> a * b    # 5*2
10
>>> a ** 2   # 5**2
25
```

İndi isə açar sözlərdən istifadə etməyə çalışaq:

```
>>> for = 10
# for və import açar sözlərdir
SyntaxError: invalid syntax
>>> import = 0
SyntaxError: invalid syntax
```

`import` açar sözündən kitabxanaları proqrama "ixrac" etmək üçün istifadə edirik və `import` açar sözündən sonra kitabxananın adı yazılır. Bizdə isə, `import = 0` yazıldığı üçün `Syntax error` – Sintaksis xətası alırıq. Dəyişənə ilk dəfə qiymət mənimsətmək – onu *inisializasiya* etmək(*initialisation*) deməkdir. Bir dəyişənə sonsuza qədər qiymət mənimsətmək olar. Çünki, hər dəfə sadəcə olaraq bu dəyişənin dəyəri yenilənir – köhnə dəyər məhv edilir(bu barədə ətraflı söhbət edəcəyik), əvəzinə yeni dəyər mənimsədilir:

```
>>> v = 1
>>> v = 3
>>> v = 4
>>> v = 0
>>> v
0        # sonda 0
```

Qrup mənimsətmə və " ; " məsələsi. Təsəvvür edin ki, çox saylı dəyişən təyin etməlisiniz. Qoy bunlar `a=1`, `b=2`, `c=3`, `d=4` və sairə olsun. Bütün bunları bir-bir, hər birini yeni sətrdən yazmaq nə qədər "ət tökən" görünür... Xoşbəxtlikdən biz bu mənimsətməni qrup şəklində həyata keçirə bilərik. Bu zaman əvvəlcə vergül ilə dəyişənlər sadalanır, daha sonra `"="` işarəsindən sonra uyğun qiymətlər analoji yolla sadalanır:

```
>>> a, b, c, d = 1, 2, 3, 4
>>> a, b, c, d
(1, 2, 3, 4)
```

Bundan öncə başqa proqramlaşdırma dilləri ilə maraqlanmısınızsa, bilirsiniz ki, əksər proqramlaşdırma dillərində sətrlərin sonunda nöqtə-vergül qoyulur. Hər sətr bir əmr olduğundan, bu yolla kodu icra edəcək translyatora sətrin harada bitməsini göstərirlər. Pythoda isə buna ehtiyac yoxdur. Çünki, pythona görə - sətr harada bitdisə, deməli əmr də orda bitdi. Yəni adətən bir əmr bir sətrdə yazılır və `;` üçün ehtiyac qalmır. Amma, buna baxmayaraq pythonda ; istifadə etmək olar, yəni, dil buna imkan verir və heç bir xəta baş vermir.

```
>>> a=1; b=2;
-----------------
>>> a=1
>>> b=2
-----------------
>>> a, b = 1, 2
```

Yuxarıdakı nümunələrin üçü də eyni işi görür, yəni, ekvivalentdirlər. Ümumiyyətlə, pythonda `;` çox nadir hallarda istifadə olunur. Dövrlər və bloklar ilə tanış olduqdan sonra "Python fəlsəfəsi(Python Zen)" ilə tanış olacağıq və orada bir daha bu mövzuya dönəcəyik.

Case-sensitive. Pythonda dəyişənləri adlandırarkən diqqət etməli olduğunuz daha bir məqam – hərfin hərfin yuxarı və ya aşağı registrdə olmasıdır, yəni, hərfin böyük və ya kiçik formada yazılmasıdır. Belə ki, pythonda "A" ilə "a" fərqli hesab olunur.

```
>>> x = 9
>>> X = 99
>>> x
9
>>> X
99
```

Tip

Qeyd etmişdim ki dəyişənləri adlandırarkən istənilən əlifbadan istifadə etmək olar. Ancaq, proqramlaşdırma adətən kollektiv bir məşğuliyyət olduğu üçün əzəldən indilərə qədər yaranmış yanaşmalar, konvensiyalar var. Proqramlaşdırmanın dili ümumidünya dili - ingilis dilidir. Buna görə də dəyişən adları, şərhlər - məhz bu dildə yazılır. Təsəvvür edin ki, bir çinlinin koduna baxırsınız və adam hər şeyi Çin heroqrifləri ilə yazıb... Daha bir məsələ - dəyişənlərin mənalı adlandırılmasıdır. Təsəvvür edin ki, 1000 sətrlik yazmısınız və bu kodda 5000 dəyişən var. Bunların hamısını - qısa, mənasız şəkildə adlandırsanız, hansı dəyişənin

özündən nə ehtiva etdiyini, nə üçün yaradıldığını özünüz belə unudacaqsınız. Ona görə də real kod yazarkən mənalı adlardan istifadə etməyə çalışın. Adlar bir neçə sözdən ibarət olduqda, ayırıcı kimi `_` istifadə etmək olar: `apple_count` və sairə.

2.3.2. Sətrlər

Info

Bu hissədə ədədlər qədər əhəmiyyət kəsb edən və indiyə qədər dəfələrlə rastlaşdığımız sətrlərdən danışacam. Ədədlər kimi sətrlər üzərində də olduqca maraqlı və faydalı əməliyyatlar aparmaq olar. Bundan başqa, bölmədə obyektyönlü proqramlaşdırmanın üç nəhəng filləri sayılan üç məvhumdan biri - "polimorfizm" ilə tanış olacaqsınız. Hazırlaşın, yavaş-yavaş proqramlaşdırmağa baş vururuq!

İlkin tanışlıq

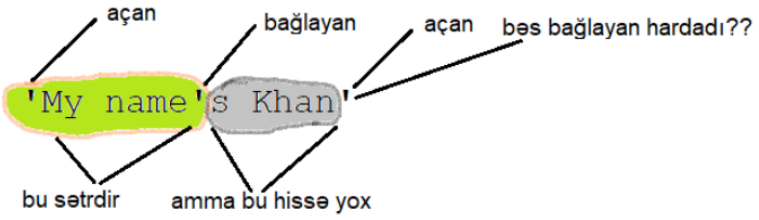
Sətr(ing: *string*) – apastrof (`'` və `'`) və ya dırnaq işarələri (`"` və `"`) daxilində yazılan simvollardan ibarətdir. Bu simvollar istənilən simvol ola bilər.

```
>>> my_string = "My awesome string"
>>> my_string2 = 'My another awesome string'
>>> empty_string = ""
```

İndi isə bir yanlış hala baxaq:

```
>>> s = 'my name's Khan'
SyntaxError: invalid syntax
```

Yuxarıdakı nümunədə tanış sintaktik xəta aldıq. Səbəb isə, yazılışın səhv olmasındadır. Sətr `"` və ya `'` ilə başladıqda növbəti `"` və ya `'` rast gəlindikdə sətrin sonu hesab edilir. `'my name's Khan'` yazılışında yaşıl rənglə verilmiş hissə sətrdir. Əgər sətrin əvvəlində və sonundakı dırnaqları bir növ açan və bağlayan mötərizə hesab etsək, xətamızn səbəbini belə bir paint cızma-qarası ilə göstərmək olar.



Əgər sətrin içərisində apastrof işlətmək istəyiriksə, ən yaxşı üsul sətri dırnaqlar daxilində yazmaqdır və əksinə, daxilində dırnaqlardan istifadə etmək istəyiriksə sətri apastroflar daxilində yazmaq olar. Belə ki, aşağıdakı nümunələr düzgündür:

```
>>> s = "My name's Khan"
>>> s = 'My name is "Khan"'
```

Daha bir üsul isə simvolun sətrin bir hissəsi olduğunu göstərməkdir. Bunun üçün həmin simvolun qarşısında `\` - *backslash*("bəksləş") işarəsi yazılır.

```
>>> s='my name\'s Khan'
>>> s
"my name's Khan"
```

Gördüyünüz kimi s sətrində \ simvolu çap edilmir, çünki, onun təyinatı özündən sonra gələn xüsusi simvolu sətrin bir parçasına çevirməkdir. **Çiy sətrlər.** Sətr daxilində xüsusi apastrof və ya dırnaqlardan istifadə etmək üçün qarşısına
işarəsi yazdıq. Bəs `\` işarəsinin özünü istifadə etmək istəsək necə edək? Tutaq ki, sətrimiz belə olmalıdır: `"backslash: '\' "`. Bu sətri olduğu kimi yazsaq:

```
>>> "backslash: '\'"
"backslash: ''"
```

Yuxarıdakı yazılışda `\` simvolu özündən sonra gələn ' simvolunu sətrin bir hissəsinə çevirərək, çap olunmur. Amma \ simvolunun özünün də çap olunmasını istəyirik. Bunun üçün ən rahat yol *çiy sətrlərdən*(raw strings) istifadə etməkdir. Bunun üçünsə, sətrin qarşısına `r` hərfi yazılır:

```
>>> r"backslash:'\'"
"backslash:'\\'"
```

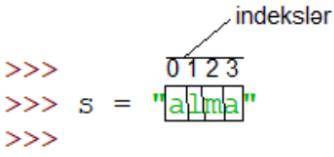
Adətən sətr icra edilməzdən əvvəl içərisindəki \ simvolları öz işlərini görür – yəni məsələn apastrof və ya dırnaqları sətrin bir hissəsinə çevirir. Ondan sonra hazır sətr icra edilir. Ancaq, `r` hərfi ilə bu mərhələ arada buraxıldığı üçün sətr bir növ "çiy" qalır, "çiy sətr" anlamı da burdan gəlir . Çiy sətrlərdən ən çox kompüterdə fayl və ya qovluq yollarının yazılışında istifadə edirıir. Bu yol ingiliscə - path(cığır, yol) adlanır. Məsələn, Windows əməliyyat sisteminə xas olan belə bir yolumuz var: `C:\Users\Aziz\Downloads\book.docx`. Yolun daxilində çoxlu sayda `\` olduğu üçün bu sətri daxil etməyin ən rahat yolu – onu çiy sətrə çevirməkdir:

```
>>> path = r"C:\Users\Aziz\Downloads\book.docx"
>>> path
"C:\Users\Aziz\Downloads\book.docx"
```

Sətrlər üzərində əsas əməllər

İndekslər. Sətr özlüyündə simvollar ardıcıllığıdır, əlbəttə ki, bu ardıcıllıq yalnız bir somvoldan ibarət ola(`"a"`) və ya boş ola(`""`) da bilər. Ardıcıllıq olduğu üçün isə biz sətrlər üçün indekslərdən istifadə edə bilərik. İndeks, sadə dildə desək – bir və ya çoxlu elementləri/üzvləri olan yığmada hər bir elementi fərqləndirən, unikal olan bir xüsusiyyətdir. Qayıdaq sətrlərə, tutaq ki `s = "alma"` sətri verilib. Sətrdə iki ədəd "a" hərfi var və sizə sətrdəki "a" hərfini almaq lazımdır. Bunu necə etmək bir yana, ilk olaraq sual yaranır ki "hansı "a" hərfini almaq tələb olunur, 1-ci hərfin yerində duran "a", ya

3-cü hərfin yerində duran "a" ?". Gördüyünüz kimi, ardıcıllıqda elementlər ardıcıl olduğu üçün – hər bir elementin öz mövqeyi olduğu üçün, onlara mövqelərinə görə müraciət etmək daha rahat görünür. Məsələn, "s sətrinin 1-ci simvolu" və s. Buna görə də ardıcıllıqlarda unikal indeks kimi – elementin sıradakı yerindən, yəni indeksindən istifadə edilir. Ancaq bir mühüm məsələ var: sıra daxilində indeksləmə(sayım) 1-dən deyil, 0-dan başlayır. Bu o deməkdir ki, "alma" sətrindəki ilk "a" simvolu 1 yox, 0 mövqeyində yerləşir. Pythonda sətrin uyğun indeksində yerləşən simvolu almaq üçün `string[n]` yazılışından istifadə olunur. Burada: `string` – sətr, `n` – indeksdir.




```
>>> str1 = "spam"
>>> str1[0]
's'
>>> str1[1], str1[2], str1[3]
('p', 'a', 'm')
```

Hələki çətin heç bir şey yoxdur. Sətr və ümumiyyətlə istənilən ardıcıllığın uzunluğunu təyin etmək üçün `len()` funksiyasından istifadə edə bilərik(ing: *length* - uzunluq). Bu funksiya ardıcıllıqdakı elementlərin sayını qaytarır. Siz bilirsiz ki, ardıcıllıqlarda indeksləmə 0-dan başlayır. `len` funksiyası isə uzunluğu qaytarır. Belə çıxır ki, ardıcıllığın son elementinin indeksi `=len(sequence) – 1` olacaq:

```
>>> str1
'spam'
>>> len(str1)   # uzunluq = 4 simvol
4
>>> str1[3]     # bilirik ki sonuncu 'm'-dir və onun indeksi 3-dür.
'm'
>>> str1[len(str1) - 1]
'm'
```

Pythonda həmçinin mənfi indekslərdən də istifadə etmək olar. Əgər adi halda `str1[1]` bizə 2-ci elementi qaytarısa, `str1[-1]` bizə sağdan, yəni sondan 1-ci elementi qaytaracaq. Belə olan halda sonuncu elementi daha asan yolla almaq olar:

```
>>> 'spam'[-1]    # sondan 1-ci
'm'
>>> 'setr'[-2]    # sondan ikinci
't'
>>> 'setr'[-0]    # belə bir şey təbiətdə yoxdur, ancaq 0 indeksli elementi
qaytarır
```

Bir az düşünsək, `len` funksiyasının köməyi ilə sonuncu elementi aldığımız kimi, mənfi indekslə birinci elementi də ala bilərik, sadəcə mənfi indeksləmə 0-dan deyil, `-1`-dən başladığı üçün `-1` yazmırıq:

```
>>> s ='spam'
>>> s[-len(s)]
's'
```

Maraqlıdır, bəs mövcud olmayan indeksə müraciət etməyə cəhd etsək nə olar? Belə olan halda `IndexError` (indeks xətası) - xətasını almış olarıq:

```
>>> 'spam'[5]    # 5 indeksi mövcud deyil
Traceback (most recent call last):
File "<pyshell#1>", line 1, in <module>
'spam'[5]
IndexError: string index out of range
```

Xüsusi idarəedici simvollar

Bu simvollar `\` ilə yazılır və sətrlərin daxilində xüsusi əməliyyatı yerinə yetirir. Bunlardan ən çox istifadə olunanlar aşağıdakılardır:

simvol	təyinatı
`\n`	yeni sətrə keçid
`\t`	bir tab(4 boşluq) - üfüqi
`\\`	`\` simvolu
`\"`	`"` simvolu
`\'`	`'` simvolu

Hər bir xüsusi simvol **bir** simvol kimi sayılır, yazarkən bir simvoldan çox istifadə etsək də.

```
>>> len('spam')
4
>>> s = 's\np/na/nm'    # \n – yeni sətr
>>> s
s
p
a
m
>>> len(s)
7
>>> hi = 'My name is:\tPython'  # \t – tab
>>> hi
'My name is:    Python'
```

Sətrlərin bəzi metodları

Mən əvvəldə qeyd etmişdim ki, işlədiyimiz bütün verilənlər – *obyektlərdir*. *Metodlar* isə, ümumilikdə obyekt yönlü proqranlaşdırmanın mövzusu olduğu üçün bizim hələki metodlar barədə ancaq onu bilməyimiz kifayıtdir ki: metod – demək olar ki, adi funksiyadır, sadəcə müəyyən obyektə aid olur və ondan nöqtə ilə ayrılaraq yazılır. Aşağıdakı cədvəldə bəzi sətr metodları verilib. Sətr metodları sətrdən nöqtə ilə ayrılır. Məsələn: `"setr".metod()` Aşağıdakı cədvəldə ən çox istifadə olunan sətr metodları verilib:

metod	təyinatı
`s.find('pa')`	'pa' altsətrinin indeksini qaytarır
`s.strip()`	sətrin sağ və sol tərəfindəki xüsusi boşluq simvolu silir
`s.rstrip()`	sətrin sağ tərəfindəki xüsusi boşluq simvolu silir
`s.lstrip()`	sətrin sol tərəfindəki xüsusi boşluq simvolu silir
`s.replace('pa', 'li')`	Birinci altsətri, ikincisi ilə əvəz edir
`s.split(',')`	Sətri ötürülmüş bağlayıcı rast gəlinən hissələrdən parçalayır
`s.upper()`	Hərfləri yuxarı registrə çevirir(böyük hərflərə)
`s.lower()`	Hərfləri aşağı registrə çevirir(kiçik hərflərə)
`s.capitalize()`	Sətrin ilk hər simvolunu yuxarı registrə çevirir
`s.endswith('am')`	Ötürülmüş altsətrlə bitməsini yoxlayır
`s.startswith("sp")`	Ötürülmüş altsətrlə başlamasını yoxlayır
`s.join(strlist)`	Ötürülmüş siyahıdakı elementlərin aralarına s sətrini qoyaraq birləşdirir
`s.isdigit()`	Sətrin ədəd olmasını yoxlayır
`s.count('a')`	Sətrdə ötürülmüş altsətrin sayını qaytarır
`s.encode('latin-1')`	Sətri ötürülmüş kodlaşdırma sistemində kodlaşdırır
`b.decode('utf8')`	Bayt sətrini ötürülmüş kodlaşdırma sisteminə qaytarır

```
>>> s = 'spam'
>>> s2 = 'programming'
>>> s1.find('am'), s2.find('t') # axtarış. Axtarılanın indeksini qaytarır.
(2,-1)
>>> '\nstring\t'.strip()
'string'
>>> '\nstring\t'.rstrip()
'\nstring'
>>> '\nstring\t'.lstrip()
'string\t'
>>> s2.replace('program', 'gam'), s2.replace('g', 'G') # əvəzləmə
('gamming', 'proGramminG')
>>> s.upper(), 'SPAM'.lower(), s.capitalize()
('SPAM', 'spam', 'Spam')
>>> s2
'programming'
```

```
>>> s2.count('m'), s2.count('p'), s2.count('j')
(2, 1, 0)
```

Digər metodları izah etmək üçün sizə səthi olaraq məntiqi tip və siyahılar barədə məlumat verəcəm. Biz bu iki tip verilənlər ilə gələcəkdə ətraflı tanış olacağıq, narahat olmayın.

Məntiqi tip. Siz artıq bool cəbrinin əsasları ilə tanışsınız. Mülahizələrə Doğru və Yanlış kimi hökmləri verə bilirsiniz. Pythonda *Doğru* – `True` , *Yanlış* isə - `False` olaraq yazılır. Ədədlər arasında münasibət bildirən riyazi operatorlar üçün: kiçikdir: `<` ; böyükdür: `>` ; böyükdür və ya bərabərdir: `>=` ; kiçikdir və ya bərabərdir: `<=` ; fərqlidir: `!=` ; bərabərdir: `==` .

```
>>> 2 == 2
True
>>> 2 < 1.9, 99 > 90, 3 == 3.0
(False, True, True)
>>> 2 == -2, 100<=100, 2 != 2
(False, True, True)
```

Siyahılar. Siyahılar pythonda ardıcıllığın bir tipidir. Siyahının elementləri istənilən tipdə(ədəd, sətr, siyahı və s.) ola bilər və bu elementlər vergül ilə ayrılır. Siyahılar kvadrat mötərizələr ilə yazılır.

```
>>> li = []    # boş siyahı
>>> li2 = [1,2,3]    # ədədlər
>>> li3 = ['a', 'b', 'c'] # sətrlər
>>> li4 = [1, 'a', 2, 'd']    # ədədlər və sətrlər
>>> li5 = [li2, li3, li4]    # ədədlər, sətrlər və siyahılar
>>> li5
[[1, 2, 3], ['a', 'b', 'c'], [1, 'a', 2, 'd']]
```

Öncə dediyim kimi, bu sadəcə səthi baxışdır. Hər iki tip barədə daha ətraflı növbəti bölmələrdə danışılacaq. İndi isə qalan metodlara baxaq:

```
>>> s = 'spam,ham,py'
>>> s.split(',')
['spam', 'ham', 'py']
>>>
>>> names = "tom+harry+bruce"
>>> names.split('+')
['tom', 'harry', 'bruce']
>>>
>>> '+'.join(['tom', 'harry', 'bruce'])
"tom+harry+bruce"
>>>
>>> file_path = r'\'.join(['Documents', 'books', 'learning python.pdf'])
>>> file_path
"Documents\books\learning python.pdf"
>>>
>>> 'Python'.startswith('Py'), 'Python'.endswith('N')
(True, False)
```

```
>>> '777'.isdigit(), '-777'.isdigit(), '777.0'.isdigit()
(True, False, False)
```

Sonuncu nümunədən gördüyünüz kimi isdigit metodu yalnız müsbət tam ədədlər üçün doğru qiyməti qaytarır.

Sətrlərdə dilimləmə əməliyyatı

Sətrlərdə indekslərlə işləməyi artıq bilirsiniz. Məsələn:

```
>>> 'spam'[0], 'spam'[-1]
('s', 'm')
```

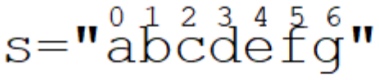
Dilimləmə əməliyyatı bizə indekslər vasitəsilə yalnız bir deyil, bir neçə simvoldan ibarət hissəni almağa kömək edir. Üstəlik, bu simvollar ardıcıl olmaya da bilər. Dilimləmə əməliyyatının sintaksisi belədir.

```
string[start_index: stop_index: step]
```

Buradakı üç ədəd parametr:

- `start_index` – başlanğıc indeks
- `stop_index` – son indeks(özü daxil deyil)
- `step` – addımın ölçüsüdür. Ümumiləşdirsək, başlanğıc indeksdən başlayaraq, son indeksə qədər(son indeksin özü xaric olmaqla), hər `addım`-ıncı simvol götürülərək birləşdirilir. Sonuncu – addım parametrini buraxa da bilərik. Bu zaman, susmaya görə addım=1 götürülür. Yəni ardıcıl bütün simvollar götürülür.

```
>>> 'string'[1:4]      # 1-dən 4-ə qədər, addım = 1 ilə
'tri'
>>> 'Azerbaycan'[-3:]   # son indeks də buraxılıb. -3-dən, sona qədər
'can'
>>> "spam"[::]      # bütün parametrlər buraxılıb, sətrin özü qaytarılır.
'spam'
>>> 'spam'[::-1]    # başlanğıc və son indekslər buraxılıb, addım = -1,
'maps'              # bu isə, tərsinə yazılış anlamına gəlir
>>> digits = '0123456789'  # rəqəmlər sətri
>>> digits[2::2], digits[1::2] # cüt və tək rəqəmlər
('2468', '13579')
```



Addım parametri mənfi olduqda indeksləmə sağdan-sola doğru gedir. Bu isə o deməkdir ki, başlanğıc və son bir növ yerlərini dəyişirlər. Ona görə də əgər `s[1:4:1]` yazılışı bizə 1-dən 4-ə qədər 1 addımı ilə dilim qaytarırdısa, `s[1:4:-1]` boş sətr qaytaracaq. Çünki, addım parametrimiz `-1`-dir, deməli dilim sağdan-sola doğru alınacaq. Başlanğıc=1, son=4 kimi parametrlərdə isə

sağdan-sola doğru getmək olmur. Yəni, başlanğıc parametrindən başlayıb sola deyil, sağa doğru gedirik, ona görə də məntiqi yanaşmada mənfi addımlı dilimdə başlanğıc parametr, son parametrindən böyük olmalıdır. Başqa sözlə başlamalı olduğumuz simvol, sonlamalı olduğumuz simvolun sağında olmalıdır. Məsələn, `s[4:1:-1]` yazılışı doğrudur və bizə `"edc"` qaytaracaq. Özünüz də, əminlik üçün dilimlərlə oynaya bilərsiniz.

`slice` funksiyası. `slice` (ing: dilim) funksiyasının köməyi ilə dilim obyekti yaratmaq olar. Bu obyektin vasitəsilə, biz eyni cür dilimi müxtəlif ardıcıllıqlara tətbiq edə bilərik. Məsələn, tutaq ki, bir neçə ardıcıllıq var. Bu ardıcıllıqların hamısından deyək ki, `[2:-1: 2]` parametrli dilimlər almaq istəyirsiniz. Bu zaman məcburən hər bir sətr üçün bütün bu parametrləri yazmaq məcburiyyətində qalacaqsınız. Bu isə, olduqca əziyyətli iş ola bilər. Ancaq, slice funksiyasının köməyi ilə siz lazımi parametrləri funksiyaya ötürərək, xüsusi dilim obyekti alırsınız. Bu dilim obyektini dəyişənə mənimsədib dilim parametrləri əvəzinə istifadə edə bilərsiniz:

```
>>> s = 'qwerty'
>>> s2 = 'abcdef'
>>> s[1:5:2]
'wr'
>>> s2[1:5:2]
'bd'
>>> dilim = slice(1,5,2)
>>> s[dilim], s2[dilim]
('wr', 'bd')
>>> type(dilim)
<class 'slice'>
```

Birləşdirmə və təkrarlama əməlləri. Polimorfizm ilə ilkin tanışlıq.

`+` və `*` operatorları ilə artıq tanışsınız, ədədlər üçün onlar uyğun olaraq toplama və vurma əməlini icra edirdilər. Ancaq, bu operatorları sətrlər üçün də tətbiq etmək olar. Gəlin düşünək, iki sətri toplasaq (`+` operatoru binar operatordur, yadınızdadırsa) nəticə olaraq nə ala bilərik? Məsələn, `"salam" + "dünya"` üçün. Əminəm ki, əksəriyyət başlıq və məntiqdən irəli gəlib `"salamdünya"` deyər və bu doğrudur! Sətrlər üçün `+` operatoru - birləşdirmə əməlini yerinə yetirir. Bu əməlin başqa bir adı isə *konkatenasiya* əməlidir. Konkatenasiya əməli əksər proqramlaşdırma dillərində var və aktiv istifadə olunur. Bəs `*` operatoru necə? İki sətri bir-birinə vursaq, məntiqli bir şey alınmayacaq. `"salam" * "dünya"` nə ola bilərki? Cavab: heç nə! Xəta alacağıq, çünki iki sətr üçün `*` operatoru təyin edilməyib və bu məntiqlidir. Bəs `"salam" * 3` desək? Baxaq:

```
>>> 'hello' + 'World'
'helloWorld'
>>> 'hello' + ' ' + 'world' + ' ' + '!!'
'hello world !!'
>>> "hello" * 3, 3 * "hellohellohello"
("hellohellohello", "hellohellohello")
>>> '-' * 2, '-' + '-'
('--', '--')
>>> '-' * 2 == '-' + '-'
True
```

Gördüyünüz kimi, `sətr * ədəd` bizə sətrin ədəd dəfə təkrarını verir. Özü də fərqi yoxdur, `sətr*ədəd` ya `ədəd*sətr`, lap klassik vurma əməli kimi. Gəlin xətalı nümunələrə də baxaq:

```
>>> 'hi' * 'hi'
TypeError: can't multiply sequence by non-int of type 'str'
>>> 'hi' + 4
TypeError: can only concatenate str (not "int") to str
```

Birinci dəfə, sətri sətrə vurmağa cəhd etdik və "tip xətası" aldıq. Xətanı "sətr tipli ardıcıllığı, qeyri-tam ədəd tipinə vurmaq olmur" kimi tərcümə etmək olar. Yəni biz ancaq ədədi - ədədə və ardıcıllığı ədədə vura bilərik. Sətr də artıq bildiyiniz kimi, ardıcıllığın bir növüdür. İkinci dəfə isə, sətr ilə ədədi toplamağa cəhd etdik və yenə də tip xətası aldıq. "sətri ancaq sətr ilə konkatenasiya etmək olar". Bu o deməkdir ki, python interpretatoru + operatorunu "toplama" olaraq deyil, konkatenasiya əməli olaraq tanıyır. Niyə, çünki + operatorunun solunda(birinci operandın yerində) sətr durur. Ona görə də python bunu avtomatik olaraq konkatenasiya olaraq tanıyır və `+` operatorunun sağ tərəfi üçün də sətr tipli obyekt gözləyir. Yaxşı, bəs sətr və ədədin yerini dəyişsək, `"hi" + 4` əvəzinə `4 + "hi"` yazsaq?

```
>>> 4 + "hi"
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

"+ üçün dəstəklənməyən operand tip(lər)i: ədəd və sətr". Bu dəfə python `+` konkatenasiya olaraq tanımadı. Sadəcə olaraq, `+` əməli ədəd və sətr arasında təyin olunmayıb.

Polimorfizm ilə ilkin tanışlıq.

Öyrəndiniz ki, `+` operatoru iki ədəd arasında toplama, iki sətr arasında isə konkatenasiya əməlini yerinə yetirir. Yəni, operatorun mənasını iştirak edən operandların tipləri müəyyən edir. Bu cür davranış – polimorfizm adlanır. Polimorfizmin köməyini ən azından sətr və ədədlər üzərində gördük və növbəti bölmədə siyahılar üzərində də görəcəyik. Ancaq bu hələ aysberqin görünən tərəfidir. Belə ki, bu paradiqmanın köməyi ilə obyekt yönlü proqramlaşdırmada öz obyektlərimizlə xaliqələr yarada bilərik. Kitabın üçüncü hissəsində bu barədə ətraflı danışacağıq.

2.4. Məntiqi tip

Artıq müəyyən qədər bul məntiqi ilə tanışsınız. Bundan əlavə, ədədlərlə işləyən zaman məntiqi ifadələrdən istifadə etmişik. Bu bölmədə isə bul məntiqinə əsaslanan - məntiqi tip ilə daha yaxından tanış olacağıq. Biz istənilən məntiqi ifadəyə, fikrə doğru – `True` və ya yanlış – `False` cavabını verə bilirik. Pythonda `True` və `False` məntiqi tipli verilənlərdir.

```
<<< type(True)
<class 'bool'>
```

Pythonda bir sıra məntiqi operatorlar var, onlar aşağıdakı cədvəldə verilmişdir.

operator	ifadə	təyinatı
`>`	`a > b`	a böyükdür b-dən

operator	ifadə	təyinatı
`<`	`a < b`	a kiçikdir b-dən
`>=`	`a >= b`	a böyük-bərabərdir b-dən
`>=`	`a <= b`	a kiçik-bərabərdir b-dən
`==`	`a == b`	a bərabərdir b-ə
`!=`	`a != b`	a fərqlidir b-dən
`in`	`a in [1,2,3]`	a `[1,2,3]` içərisində mövcuddur
`is`	`a is b`	a və b eyni obyektdir
`and`	`a and b`	a VƏ b
`or`	`a or b`	a VƏ YA b
`not`	`not a`	DEYİL a

Müqayisə operatorları. Müqayisə operatorları riyaziyyatdan bəlli olduğu qayda ilə işləyir.

```
>>> 1 > 0     # True
>>> 0 < -1    # False
>>> 2 == 2    # True
>>> 9 >= 9    # True
>>> 1 != 1    # False
```

Sətrlər üçün müqaisə operatorları. `ord` və `chr` funksiyaları. Pythonda sətrlər yunikod (unicode) kodlaşdırma sistemi ilə kodlaşdırılır. Bu barədə biz gələcəkdə daha ətraflı tanış olacağıq. Yunikod və digər kodlaşdırma sistemlərində hər bir simvol müəyyən ədədlə işarə edilir. Məsələn, elə həmin yunikodda "A" - 65, "a" isə 97 ilə işarə edilir. Beləliklə, "B" – 66, "C" – 67 və sairə. Biz sətrləri müqaisə edərkən, bir növ onların arxasında duran ədədləri müqaisə etmiş oluruq. Yeri gəlmişkən, simvolun kodlaşdırma sistemindəki ədədi qiymətini almaq üçün, simvolu `ord()` funksiyasına, əksinə, ədədi qiymətin hansı simvolu ifadə etdiyini bilmək üçün isə, ədədi `chr()` funksiyasına ötürmək lazımdır.

```
>>> ord('a')    # 97
>>> chr(97)     # 'a'
>>> ord('b')    # 98
>>> chr('98')   # 'b'
>>> 'a'<'b'     # bir növ 97 < 98 deməkdir – True
```

`in` operatoru obyektin ardıcıllıqda olub-olmamasını yoxlamağa imkan verir:

```
>>> 'a' in 'abc'      # True
>>> 'ab' in 'xyabce'  # True
>>> 'z' in 'abc'      # False
```

`is` və `==` arasında fərq. `id` funksiyası. Bir az da dəyişənlər barədə.

Dəyişənlər dəyər mənimsətdikdə = mənimsətmə operatorundan istifadə edirik. Bu zaman, dəyər obyekt yaddaşda yarıdılır, dəyişən isə sadəcə özündə - həmin obyektin yerləşdiyi yaddaş ünvanını saxlayır. Təsəvvür edin ki, bütün yaddaş(söhbət əməli yaddaşdan gedir) ədədlərlə işarələnmiş çoxlu xanalardan ibarətdir. Biz hər hansı dəyər yaradarkən, bu dəyər – bu obyekt o yaddaşın müəyyən bir hissəsində yaradılır. Biz isə, bildiyiniz kimi bu obyekti nəzərdə saxlamaq, ondan istənilən vaxt istifadə edə bilmək üçün, onu hər hansı bir dəyişənə mənimsədirik. Hər bir dəyər fərqli bir yaddaş ünvanında olduğu üçün, həmin yaddaş ünvanı bütün dəyərlər(bundan sonra obyektlər deyəcəm) üçün fərqli olacaq. Məsələn, tutaq ki, proqramda `a=1` , `b=2` dəyişənlərlə adlandırılmış obyektlər var. Bu iki obyektin hərəsinin öz unikal yaddaş ünvanı var. Bu ünvana əsaslanaraq, pythonda hər bir obyektin unikal identifikatorunu – onun heç yerdə təkrarlanmayan id nömrəsini qaytaran `id` funksiyası var. Bu funksiya, parametr kimi, id kodu qaytarılmalı olan obyekti alır və onun 13 rəqəmli id kodunu qaytarır.

```
>>> a , b, c= 1, 2, 3
>>> id(a), id(b), id(c)  # 3 dənə fərqli 13 rəqəmli kod alınacaq.
```

İndi isə, gəlin iki dənə eyni dəyərə malik olacaq dəyişən yaradaq və onların idlərini müqayisə edək:

```
>>> v1 = 999, v2 = 999
>>> id(v1) == id(v2)    # False
>>> v1 == v2            # True
```

Gördüyünüz kimi, `v1` və `v2` dəyişənlərinin dəyərləri qiymətcə eyni olsa da, id kodları fərqlidir. Burdan belə qənaətə gəlmək olar ki, deməli yaddaşda ayrı ayrılıqda iki dənə integer tipli `999` obyekti var. Bu o deməkdir ki, iki dəyərin bir birinə bərabər olması, heç də o demək deyil ki, bunlar tək bir obyektdir. Belə edək:

```
>>> a = 99999
>>> b = a
>>> id(a) == id(b)    # True
```

Yuxarıdakı nümunədə a və b dəyişənlərinin ikisi də eyni obyektə ünvanlanır. Yəni tək bir obyektə həm `a` , həm də `b` ünvanlanır. Bu o deməkdir bir növ: " a bizdə b-dir və ya əksinə - b bizdə a-dır", çünki eyni bir obyektdir. İki dəyişənin bu cür eyni olub-olmamasını biz is operatoru ilə yoxlaya bilərik. `is` – ingilis dilindən elə "*olmaq*" kimi tərcümə olunur. Yəni, yuxarıdakı eyni `==` olan a və b üçün `a is b` desək, bu – "a olmaq b" kimi başa düşülür və `True` qaytarır.

```
>>> a is b    # True
>>> b is a    # True – sağ ya sol tərəf olması fərq etmir
```

Bir neçə eyni dəyərə sahib dəyişən yaradıldıqda python yalnız bir obyekt yaradır və bütün dəyişənlər bu obyektə ünvanlanır. Bu yaddaşa qənaət üçün edilir. Daha sonra? Bu dəyişənlərdən hansınlınsa dəyəri dəyişdirilərsə, bu dəyişən üçün yeni qiymətli yeni obyekt yaradılır, beləliklə digər dəyişənlərə təsir etmir. Ancaq, bu hər zaman baş vermir. Belə ki, dəyərlər böyük olduqda, həyişənlərin ünvanlandıqları obyektlər ayrı-ayrı olur. Baxaq:

```
>>> a = 9
>>> b = 9
>>> a is b       # True
>>> a = 1
>>> a is b       # False
>>> a, b         # (1, 9)
>>> c = 9999
>>> d = 9999
>>> c is d       # False
```

Mürəkkəb şərtlər. `and` , `or` , `not` operatorları

Bu əməliyyatlar haqqında "Məntiqi cəbrin əsasları" bəhsində danışmışdım. Qısaca olaraq bir daha izah etməyə çalışacam.

- `and` - həm də "VƏ" kimi tərcümə olunur və hər iki tərəf(sol və sağ) doğru olduqda geriyə `True` qaytarır. Yəni, ancaq `True and True` olan halda geriyə `True` alırıq.
- `or` - həm də "VƏ YA" kimi tərcümə olunur və iki tərəfdən heç olmasa biri doğru olarsa, geriyə `True` qaytarır. Yəni, hər iki tərəf doğru olsa belə `True` alacağıq.
- `not` - unar operatordur, yəni işləməsi üçün yalnız bir ədəd verilən, bir operand lazımdır, gördüyü iş isə "inkar etməkdir". İnkar etmək - doğrunu-yanlış, yanlışı isə - doğru etmək deməkdir. Yəni `not True` bizə `False` , `not False` isə `True` qaytaraacaq.

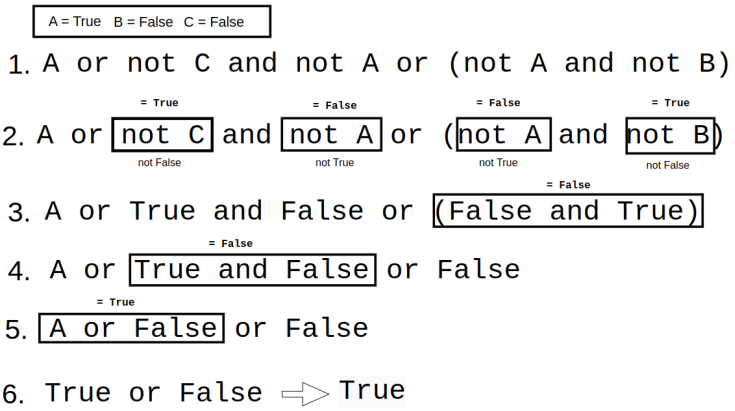
```
>>> 2 == 2 and 1 > 0     # True and True = True
>>> 9 != 8 and 2 != 2    # True and False = False
>>>
>>> 2+2 == 4 or 3>-3     # True or False = True
>>> 1+1 == 3 or 9-9 == -1   # False or False = False
>>>
>>> not 1+1 == 3         # not False = True
>>> not not 1+1 == 3     # not(not(False)) = False
```

Bu məntiqi əməllərin üstünlük ardıcıllığı.

Üstünlük ardıcıllığı bul cəbrindən görmüş olduğumuz kimidir və `or -> and -> not -> ()` istiqamətində artır. Yəni, əvvəlcə mötərizələr, daha sonra inkar, daha sonra and, ən sonda isə or icra olunur. Məsələn:

```
>>> a, b, c = True, False, False
>>> a or not c and not a or (not a and not b)   # True
```

Bu 'mazaxist' ifadəni anlamaq üçün şəklə baxaq:



`bool` funksiyası və None. Əvvəl də dediyim kimi, pythonda boş olmayan hər şey – `True` , boş olan hər şey isə - `False` qaytarır. Boş dedikdə - boş sətr, siyahı, `0` və s nəzərdə tutlur. Bundan başqa, Pythonda `None` adlanan və "heç nə", "boşluq" bildirən xüsusi obyekt də var. bool funksiyası, bizə istənilən tip veriləni məntiqi tipə çevirməyə imkan verir:

```
>>> bool(0), bool(''), bool([]), bool(None) # False,False,False,False
>>> bool(1), bool('hi'), bool([1,2])   # True,True, True
```

2.5. Siyahılar

> **Info**
>
> Bu bölmədə proqramlaşdırmanın əsas komponentlərindən olan massiv anlayışı ilə, həmçinin onlar üzərində bir sıra maraqlı əməllər ilə tanış olacaqsınız. Daha sonra öyrəndiyimiz obyekt tiplərin "dəyişilə bilən" və "dəyişilə bilməyən" adlı xüsusiyyətləri, literal və ifadə anlayışları ilə tanış olacaqsınız. Gördüyünüz kimi, yavaş-yavaş və qəti şəkildə irəliləyirik!

Massiv nədir

Artıq ədədlər və sətrlərlə tanışıq. Fikir verdinizsə, sətrlər ədədlərlə müqaisədə daha mürəkkəbdirlər. Yəni, sətrlər müəyyən sayda simvollar ardıcıllığıdır. Bir çox tarixi proqramlaşdırma dillərində char(simvol) tipli verilənlər mövcuddur. Bu tip özündə bir apastroflar arasında yazılan bir simvolu ehtiva edir. Pythonda belə bir xüsusi verilən tipi yoxdur, bunun əvəzinə sadəcə bir simvolluq sətrsən istifadə edə bilərik. Ancaq, bir anlıq təsəvvür edin ki, belə bir verilən tipi var. O zaman, sətri bu simvol tipli verilənlərin müəyyən ardıcıllığından ibarət bir mürəkkəb struktur adlandıra bilərdik və əslində adlandırmalıyıq da. Çünki, hər hansı xüsusi simvol tipinin olmamasına rəğmən, bir sətr özlüyündə müəyyən sayda altsətrlər saxlayır. Məsələn, `"spam" == 's'+'p'+'a'+'m'` . Buna əsaslanaraq da indekslərdən istifadə edirik. Qısası, bir sətrdə istədiyimiz qədər simvol saxlaya bilirik, sətr bir növ simvolların sıra ilə yığıldığı bir qutuya bənzəyir. Yaxşı, eyni qutu analogiyasını davam etdirsək, elə bir "qutu" verilən tipi varmı ki, onun içərisinə ədədlər və ya elə sətrlərin özlərini

yerləşdirək? – bəli, var! Bu tip ümumi olaraq massiv adlanır. Massiv – özündə müəyyən sayda eyni və ya müxtəlif tipli, müəyyən ardıcıllığa malik verilənlər saxlayan verilənlər strukturudur. Başqa sözlə, massiv elə bir şeydir ki, biz onun içərisinə başqa şeylər yığa bilirik və bu başqa şeylərin ardıcıllığı olur, necə ki sətrlərdə simvolların ardıcıllığı var. Bu isə o deməkdir ki, biz sətrlərə tətbiq etdiyimiz indeksləməni massivlərə də tətbiq edə bilərik.

Pythonda massiv. İndi isə qayıdaq pythona. Qeyd etmişdim ki, massivlər eyni və ya müxtəlif tipli verilənlərdən ibarət ola bilər. Python dili özlüyündə tiplərlə çox da məqudlaşmayan, "dinamik tipləşdirməyə"(gələcəkdə bu mövzuya qayıdacağıq) malik proqramlaşdırma dili olduğu üçün, massivdə müstəlif tip verilənlərdən istifadə edə bilərik. Pythonda massivlərə ən bariz misal – siyahılardır. Siyahı – özündə müxtəlif tipli verilənlər saxlaya bilən, dəyişilə bilən ardıcıllıqdır. "Dəyişilə bilən" məsələsinə bölmənin sonunda baxacağıq. Siyahını təşkil edən verilənləri - onun elementləri adlandıraq. Siyahının elementləri kvadrat mötərizələr daxilində yazılır və bir-birindən vergül ilə ayrılır. Məsələn, 1-dən 5-ə qədər ədədlərdən ibarət olan siyahı: `[1,2,3,4,5]`

```
>>> L, L2 = [1,2,3], [4,5,6]
>>> L
[1,2,3]
>>> L2
[4,5,6]
```

İndi isə fərqli tipli elementlərdən ibarət siyahılara baxaq:

```
>>> L3 = ['s',0,'spam',.5,3.14,1]
>>> L3
['s', 0, 'spam', .5, 3.14, 1]
```

Siyahılarda indeksləmə və iç-içə siyahılar

Sadə siyahılar üçün indeksləmə sətrlərdə olduğu kimidir və 0-dan başlayır:

```
>>> L = ["spam", 44, 3.14, "hi!"]
>>> L[0]     # 0 indeksli, birinci element
"spam"
>>> L[-1]
["hi!"]
>>> L[1], L[2]
(44, 3.14)
```

Burada heç bir çətinlik yoxdur, düşünürəm. Bundan başqa, tanıdığımız len funksiyasını siyahılara tətbiq edərək, siyahının uzunluğunu – elementlərin sayını ala bilərik.

```
>>> L
["spam", 44, 3.14, "hi!"
>>> len(L)
4
```

İç-içə(`nested`) siyahılar. Siyahıda müxtəlif tipli verilənlər ola bilər, hətta başqa siyahılar da:

```
>>> L = [1, [2,3], [4,5], 6]
>>> len(L)
4
>>> L[0]
1
>>> L[1]
[2,3]
>>> L[1], L[2]
([2,3], [4,5])
```

Gördüyünüz kimi `L` siyahısının 1 və 2 indeksli elementləri siyahıdırlar. `L[1]` bizə `[2,3]` qaytarırsa, belə bir şey edə bilərik:

```
>>> i = L[1]     #[2,3][0], [2,3][1]
>>> i[0], i[1]
(2, 3)
```

Əslində yeni heç bir şey görmədiniz. Ancaq bu bizi gətirib belə bir suala çıxıra bilir: əgər `L` siyahısının `i` indeksli elementi, başqa bir siyahıdırsa, o zaman bu bu içdəki siyahının j indeksli elementini almaq üçün elə birbaşa `L[i][j]` yaza bilərikmi? – Bəli, bilərik! Daha əndrabadi bir `L2` siyahısını yaradaq: `>>> L2 = [[1,2,3], "abcd", [[4,5], [6,7]]]` `L2` siyahısının `len(L2)` uzunluğu bizə 3 qaytaracaq. Bu siyahının birinci(`L2[0]`) elementi-adi siyahı(yaşıl rəngli), ikinci(`L2[1]`) elementi-sətr(göy rəngdə), üçüncü(`L2[2]`) elementi isə, mürəkkəb siyahıdır(qəhvəyi rəngdə). İndi isə baxaq:

```
>>> L2[0]       # [1,2,3]
>>> L2[0][0]    # 1
>>> L2[1]       # "abcd"
>>> L2[1][-1]   # "d"
>>> L2[2]       # [[4,5], [6,7]]
>>> L2[2][0]    #[4,5]
>>> L2[2][0][0]  # 4 (ikincinin sıfırıncısının sıfırıncısı)
```

Ümidvaram ki sizə də asan gəldi. Sadəcə "dərinlik" artdıqca, müraciət zamanı istifadə edə biləcəyimiz indekslərin sayı da artır. Məsələn, yuxarıdakı listinqdə sonuncu dəfə `L2[2][0][0]` ilə 4 aldıq. Əgər əlavə olaraq `[0]` da yazsaq, yəni `L2[2][0][0][0]` kimi, o zaman xəta almış olarıq. Çünki, bu zaman `4[0]` yazmış oluruq, bu isə mənasızdır, sadə verilən tipi olan ədəddən indeks almaq. Bu cür *"iç-içə"*(ing. nested) yazılış xətti cəbrdəki *matrislərin* yazılması üçün ideal vasitədir. Məsələn, tutaq ki, M matrisi verilir:

$$M = \begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{matrix}$$

Bu matrisi massiv şəklində yazmaq üçün iç-içə siyahılardan istifadə edə bilərik. Bunun üçün matrisin hər bir sətrini, `M` siyahısının tərkibi siyahısı kimi yaza bilərik: `M=[[1,2,3], [4,5,6], [7,8,9]]`. Daha gözəl oxunması üçün enterlərlə siyahımızı düzənləyə də bilərik:

```
M = [
     [1,2,3],
     [4,5,6],
     [7,8,9]
     ]
```

Daha səliqəli görünməyindən başqa, öncəki M siyahısından başqa heç bir şeylə fərqlənmir. Kitab boyu irəlilədikcə, çalışacağıq ki, daha səliqəli və effektiv kod yazaq. Bu barədə hələ danışacağıq.

Siyahı metodları və bəzi funksiyalar. Dəyişiləbilən və dəyişiləbilməyən tiplər

Biz hələ sətrlərlə işləyərkən, onların metodları və funksiyaları ilə tanış olmuşduq. Sətrlər kimi, siyahıların da metodları var. Fikir vermisinizsə, sətrlərin bütün metodları bizə nə isə qaytarır, amma aid olduqları sətri heç cürə dəyişmirlər. Ekran təsvirinə baxaq: Göründüyü kimi, `capitalize()` və `upper()` metodları yalnız sətrin dəyişilmiş sürətini qaytarır, movie sətrinin özü isə dəyişilməz olaraq qalır. Yəni, `movie.capitalize()` etməklə, movie "HARRY POTTER" olmayacaq. Bunun üçün nəticəni təkrarən movie dəyişəninə mənimsətmək lazımdır:

```
>>> movie = movie.upper()
>>> movie
"HARRY POTTER"
```

```
>>> movie = "Harry Potter"
>>> movie.capitalize()
'Harry potter'
>>> movie
'Harry Potter'
>>> movie.upper()
'HARRY POTTER'
>>> movie
'Harry Potter'
>>>
```

Mənimsətmə operatoru(=) sağ tərəfdəki ifadənin nəticəsini(sarı ranglə), sol tərəfdəki dəyişənə(movie) mənimsədir. Yəni, əvvəlcə sağ tərəfdəki ifadə hesablanır, daha sonra mənimsətmə baş verir. Ona görə də sağ tərəfdə `movie = "Harry Potter"` , ifadə: `movie.upper() = "HARRY POTTER"` və sonda `movie = "HARRY POTTER"` alınır. İndi gəlib çatdıq, "dəyişilə bilən və dəyişilə bilməyən tip" məsələsinə. Yuxarıdakı `movie` sətri tətbiq etdiyimiz `upper()` metoduna rəğmən dəyişmir. Ümumiyyətlə, sətrin bütün metodları sətri dəyişmir. `movie = movie.upper()` kimi yazmaq da əslində sətri dəyişmək demək deyil, çünki təkrar mənimsətmə baş verir. Bu zaman isə `movie` dəyişəni köhnə obyektdən ("Harry Potter") "açılır" və mənimsətmə operatorunun sağ tərəfindəki ifadənin nəticəsinə bağlanır. Sual yaranır: bəs, köhnə obyekt ilə nə baş verir? - Köhnə obyektə daha heç bir dəyişən ünvanlanmırsa, deməli bu obyekt daha lazım deyil və python interpretatoru onu yaddaşdan silir. Bu əməliyyat bir növ "zibil təmizləyə" bənzəyir. Bir çox dillərdə bu avtomatik yerinə yetirilir, pythonda da həmçinin. Ancaq, məsələn C++ proqramlaşdırma dilində bu proses proqramçının özü tərəfindən həyata keçirilməlidi, bu da əlavə əmək deməkdir. Belə nəticəyə gəlmək olar ki sətrlər dəyişilə bilməyən tipdir. Əgər movie dəyişəninə yenidən mənimsətmə etməsək, o heç zaman dəyişilə bilməz. İndi isə, qayıdaq siyahılara. Siyahının indeksinə obyekt mənimsədə bilərik. Belə:

```
>>> authors = ['M. Araz', 'H. Cavid', 'M. Mushvig']
>>> authors[2]
'M. Mushvig'
>>> authors[2] = "Anar"
>>> authors
['M. Araz', 'H. Cavid', 'Araz']
```

Siyahını üçüncü elementinə yeni sətr mənimsətdik və siyahı obyekti dəyişdi. Buradan iki şey öyrənmiş olduq:

1. Siyahının hər hansı elementinə indeksinə görə müraciət edib, ona yeni dəyər mənimsətdikdə bu element yeni bu yeni dəyər ilə əvəz olunur.

```
>>> l = ['a','b','c']
>>> l[-1] = 'g'
>>> l
['a','b','g']
>>> l2 = ['a', [1, 2], 'b']
>>> l2[1][0] = 99
>>> l2
['a', [1, 99], 'b']
```

2. Siyahılar, sətrlərdən fərqli olaraq, dəyişilə biləndirlər. Yuxarıda gördüyümüz kimi, siyahının indeksinə görə mənimsətdikdə siyahı özü dəyişmiş olur. Bundan başqa, siyahının bir sıra elementləri aid olduğu siyahını dəyişir, bu metodlarla indi tanış olacağıq.

Siyahı metodları

Siyahı – pythonda əsas verilən struktur tipidir. Ona görə də, metod sarıdan qıtlıq yaşamır. Gəlin, bu metodların bəzilərinə baxaq.

metod	təyinatı
`append(e)`	Siyahının sonuna `e` əlavə edir
`clear()`	Siyahını təmizləyir
`copy()`	Siyahının sürətini qaytarır
`count(e))`	Siyahıdakı `e` elementinin sayını qaytarır
`extend(i)`	Siyahını `i` -nin elementləri ilə genişləndirir
`index(e))`	`e` elementinin indeksini qaytarır
`insert(i, e)`	Siyahının `i` indeksli mövqeyinə `e` elementini daxil edir
`pop(i)`	Siyahının `i` indeksli elementini silir və qaytarır
`remove(e))`	Siyahıda soldan ilk `e` elementini silir
`reverse()`	Siyahının elementlərini əks sıra ilə düzür
`sort()`	Siyahının elementlərini artan sıra ilə düzür

```
>>> letters = ["a", "b", "c", "d"]
>>> letters.append("e")
>>> letters                # ["a", "b", "c", "d", "e"]
>>> lt_copy = letters.copy()
>>> lt_copy                # ["a", "b", "c", "d", "e"]
>>> letters.clear()
>>> letters                # []
>>> letters.append("j")    # letters = ["j"]
>>> letters.extend(lt_copy)
>>> letters                # ["j", "a", "b", "c", "d", "e"]
>>> letters.pop(0)
'j'
>>> letters           # ["a", "b", "c", "d", "e"]
>>> letters.remove("e")
>>> letters           # ["a", "b", "c", "d"]
>>> nums = [1,2,34,2,.2,7]
>>> nums_c = nums.copy
>>> nums.sort()
>>> nums        # [0.2,1,2,2,7,34]
>>> nums_c
>>> nums_c.sort(reverse = True)
>>> nums_c      # [34,7,2,2,1,0.2]
>>> letters.reverse()
>>> letters     # ["d","c","b","a"]
>>> letters.index("d")
0
>>> letters      # ["d","c","b","a"]
>>> sorted(letters)
["a","b","c","d"]
>>> letters
["d","c","b","a"]
```

Gördüyünüz kimi, metodların əksəriyyəti məxsus olduqları siyahıları dəyişir.

Siyahılar üçün `+` və `*`

Yadınızdadırsa, sətrlər üçün `+` konkatenasiya, `*` isə təkrarlama əməlini yerinə yetirirdi:

```
>>> "hello" + "_"+"world"  # 'hello_world'
>>> 'hi!' * 3              # 'hi!hi!hi!'
```

Demək olar ki, bu əməllər siyahılar üçün də analoji qaydada keçərlidir.

```
>>> [1,2,3] + [4,5,6]    # [1,2,3,4,5,6]
>>> [1,2,3] * 3          # [1,2,3,1,2,3,1,2,3]
```

Gördüyünüz kimi, siyahılar arasında `+` operatoru bu siyahıları "birləşdirir", siyahı və tam ədəd üçün isə, təkrarlayır.

2.6. Yazılar

Verilənin dəyişiləbilən və ya dəyişiləbilməyən olması olduqca vacibdir. Məsələn, siyahıların əksər metodları onları dəyişir. Bu bir tərəfdən rahat, digər tərəfdən isə təhlükəlidir. Təsəvvür edin ki, böyük bir proqram üzərində işləyirsiniz və bu proqramın mürəkkəb bir məntiqi var. Siz hər hansı "sabit" verilənləri bir siyahıda saxlayırsınız və fikrinizdə tutmusunuz ki, bu siyahı sona qədər dəyişilməyəcək, yəni bu siyahıya daha heç nə əlavə etməyəcək və silməyəcəksiniz. Amma, yenə də, siyahı dəyişiləbilən olduğuna görə təsadüfən də olsa hardasa siyahının dəyişilməsinə gətirib çıxardan səhvə yol verə bilərsiniz və çox güman ki proqramınız sonda "qəribə" işləyəcək. Sonra isə bu qəribəliyin səbəbini axtarmalı olacaqsınız, bu isə adətən uzun və yorucu olur. Yaxşı olardı ki, siyahıya bənzər bir verilənlər strukturumuz olaydı və bu verilənlər strukturu siyahılardan fərqli olaraq, dəyişiləbilməyən olaydı. Məhz bu hallarda yazılar (ing tuple) istifadə olunur. Yazılar, siyahılar kimidir, ancaq onlardan fərqli olaraq, dəyişiləbilməyəndirlər və buna görə də onların siyahılar kimi `append`, `pop` və sairə *obyektdəyişən* metodları yoxdur. Yerdə qalan bütün digər xüsusiyyətlər isə siyahılar ilə eynidir. Yazı yaratmaq üçün "yumru mötərizələrdən – `()`" istifadə olunur.

```
>>> tp = (1,2,3)
>>> type(tp)
<class 'tuple'>
```

Yazıları mötərizələrsiz də yaratmaq olar:

```
>>> tp = 1,2,3
>>> type(tp)
<class 'tuple'>
```

Siyahılar kimi, yazılar da ardıcıllıqdır, odur ki, eyni indeksləmə və dilimləmə əməllərindən burada da yararlana bilərik.

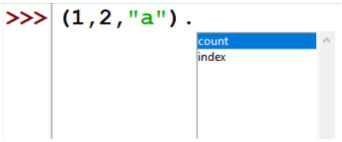
```
>>> len(tp)   # 3
>>> tp[0]     # 1
>>> tp[-1]    # 3
>>> tp2 = ('str', (7,8), 4, .9, 1, 0)
>>> tp2[1:3]      # ((7,8), 4)
>>> tp2[::-1]     # (0,1,0.9,4,(7,8),'str')
```

Siyahılar kimi, yazılar da müxtəlif dərinlikdə ola, müxtəlif tipli elementlərdən ibarət ola bilərlər:

```
>>> t = ((('a','b'),('c','d')),'e', 'f',(1,2,3))
>>> t[0]          # (('a','b'),('c','d'))
>>> t[0][0[0]]    # 'a'
```

Yazı metodları

İndiyə qədər sətr və siyahı metodları ilə tanış olmusunuz. Üstəlik, sətrlər və siyahılar arasında dəyişiləbilən və dəyişiləbilməyən tiplər barədə öyrənmisiniz. İndi təsəvvür edin ki, yazı – dəyişiləbilməyən siyahı kimi bir şeydir. O zaman, siyahının metodlarından siyahını dəyişən metodları çıxın və yerdə qalacaq metodlar(siyahı dəyişməyən metodlar) yazı metodları olacaq, bunlar da siyahılarda olduğu kimi çalışır.



Tək elementli yazını necə yaratmalı

Tək elementli yazını yaratmaq üçün bu tək elementdən sonra vergül qoyulur. Bildiyiniz kimi, yazıları mötərizələrsiz də yaratmaq mümkündür, odur ki siz `t=(1)` və ya `t=1` yazdıqda `t – int` tipində olur. Odur ki, yazdığınızın yazı tipi kimi ayırd edilməsi üçün bu tək elementdən sonra vergül qoymalısınız:

```
>>> t = (1)
>>> type(t)       # <class int>
>>> t2 = 1,       # type(t2): <class 'tuple'>
>>> t3 = (1,)     # type(t3): <class 'tuple'>
```

Yazılarda `*` və `+`. Yazılarda `*` və `+` siyahılarda olduğu kimidir:

```
>>> (1,2,3) + (4,5,6)   # (1,2,3,4,5,6)
>>> (1,2) * 3           # (1,2,1,2,1,2)
```

2.7. Tip Çevirmələri

Pythonda elə tiplər var ki, onlar bir növ bənzərdirlər və bu bənzərlik onların tipini dəyişməyə imkan verir. Bu cür tip çevirmələri type casting adlanır. Məsələn, float tipli 2.5 ədədini integerə və ya `[1,2,3]` siyahısını `(1,2,3)` yazısına çevirə bilərik. Veriləni hər hansı standart tipə çevirmək üçün həmin tipin adı ilə eyni olan funksiyalardan istifadə edilir. Məsələn, tam ədədə(integer) çevirmək üçün `int()`, üzən nöqtəli ədədə çevirmək üçün `float()`,məntiqi tipə çevirmək üçün `bool()`, sətrə çevirmək üçün `str()`, siyahı üçün `list()`, yazılar üçün isə `tuple()` funksiyasından istifadə edilir. Gəlin növbə ilə bütün imkanlarla tanış olaq.

tip	çevirici funksiya
tam ədəd	`int()`
üzən nöqtəli ədəd	`float()`
məntiqi	`bool()`
sətr	`str`
siyahı	`list()`
yazı	`tuple()`
lüğət	`dict()`
çoxluq	`set()`

Ədədlər arasında. `float` ilə `integer` arasında keçid olduqca asandır. `integer – float` keçidi zamanı tamın sonuna 0-a bərabər kəsr hissə əlavə edilir, əksinə - float-integer keçidi zamanı isə, kəsr hissə atılır.

```
>>> int(2.9)    # 2
>>> int(.7)     # 0
>>> int(2.0)    # 2
>>>
>>> float(3)    # 3.0
>>> float(0)    # 0.0
```

Ardıcıllıqlar arasında. Siyahı və yazı arasında çevirmələr sadədir.

```
>>> tuple([11,21,[1,2,3], 2])  # (11,21,[1,2,3],2)
>>> list((11,21,[1,2,3],2))    # [11,21,[1,2,3],2]
```

Sətri siyahıya və ya yazıya çevirdikdə sətr elementlərinə parçalanır.

```
>>> list("hi !")   # ['h', 'i', ' ', '!']
>>> tuple("hi !")  # ('h', 'i', ' ', '!')
```

Siyahı və ya yazı sətrə çevrildikdə isə, sadəcə olaraq `""` arasına alınır.

```
>>> str([1,2,3])   # '[1,2,3]'
>>> str((1,2,3))   # '(1,2,3)'
```

Sətrdən ədədə. Sətri ədədə çevirmək üçün gərək həmin sətr "ədədə çevrilən" olsun. Məsələn, sətr `"abc"` olarsa, bunu ədədə çevirmək olmur, `"123"` isə `123` ədədinə çevrilə bilər. Eləcə də `float` üçün, `"2.5"` `2.5` -ə çevrilə bilər, amma "2.5a" yox. Ədədi sətrə çevirərkən isə, sadəcə olaraq " " arasına alınır:

```
>>> int("123")   # 123
>>> float(".9")  # 0.9
>>> int("123L")  # xəta!
```

Onu da qeyd etmək lazımdır ki, `float` funksiyasına sətr şəklində tam ədəd də ötürmək olar, amma `int` funksiyasına sətr kimi üzən nöqtəli ədəd göndərmək olmaz:

```
>>> float('9')   # 9.0
>>> int('2.5')   # xəta!
```

2.8. Lüğətlər

İndiyə qədər tanış olduğumuz ardıcıllıqlarda biz elementlərə - onların indekslərinə görə müraciət etmişik. İndi tanış olacağımız ardıcıllıq növündə isə, biz indeks deyil, açarlardan istifadə edəcəyik. *Lüğtlər* – elə ardıcıllıqdır ki, burada hər bir dəyərin öz unikal, yəni həmin lüğətdə bir daha

təkrarlanmayan açarı olur. Lüğtlərin hər hansı elementinə müraciət etmək istədikdə, həmin elementin açarından istifadə edirik. Lüğətlərin sintaksisi:

```
{açar:dəyər, açar2:dəyər2, açar3:dəyər3}
```

Bəs bu yazılışın nə kimi üstünlükləri var? Əsas üstünlük sürətdədir. Əgər məsələn, siyahıda hər hansı elementi axtarmalı olsaydıq, bütün siyahını yoxlamalı olacaqdıq(bu barədə "Alqoritmin keyfiyyət göstəriciləri və yaxud yaxşı alqoritmi necə seçməli" danışmışdıq). Amma lüğtlərdə hər dəyərin öz unikal açarı olduğu üçün daha axtarışa ehtiyac qalmır. Sintaksisdən göründüyü kimi, lüğətlər *açar*: *dəyər* cütlərindən ibarət olan ardıcıllıqdır. Bu ardıcıllıqda açar – dəyişiləbilməyən tip verilərn(sətr, ədəd, yazı və s), dəyər isə istənilən tipdə verilən ola bilər – digər bir lüğət belə. Lüğətin elementinə müraciət etmək üçün, kvadrat mötərizələr daxilində lazım olan elementin açarını daxil edirik.

```
>>> book = {"author" : "Sabir", "name": "Pocht qutusu"}
>>> book["name"]    # "Sabir"
>>> book["name"]    # "Pocht qutusu"
>>> book["year"]    # Key Error – xəta
```

Sonuncu sətrdən gördüyünüz kimi, mövcud olmayan `"year"` açarına müraciət etdikdə, `Key Error` xətasını aldıq. Elementlərə müraciətin digər bir yolu isə, get metodudur. Bu metod iki parametr alır: tələb olunan elementin açarı və element tapılmazsa, qaytarılmalı olan dəyər. Bu o deməkdir ki, əgər `book.get("year", 0)` desək, o zaman `"year"` açarlı element tapılmazsa, `0` alacağıq. İkinci parametr optimal parametrdir, yəni susmaya görə qiyməti var və bu qiymət `None` bərabərdir.

```
>>> book
{"author" : "Sabir", "name": "Pocht qutusu"}
>>> book.get("year", 0)   # 0
>>> book.get("year")      # None
>>> book.get("name", 0)   # "Pocht qutusu"
```

Lüğət yaratmağın daha bir yolu – `dict` funksiyasıdır. Bu funksiyaya ötürülən adlı parametrlər funksiyanın qaytaracağı lüğətin elementləri olacaq. Heç bir parametr ötürülməzsə, boş lüğət yaradılır.

```
>>> d = dict(name = 'John', surname = 'Smith')
>>> d             # {'name' : 'John', 'surname' : 'Smith'}
>>> d2 = dict()
>>> d             # {} boş lüğət
```

Lüğətlərin metodları

Fərz edək ki, belə bir `d` lüğətimiz var:

```
>>> d = {'a':1, 'b':2, 'c':3, 'd':4}
```

- `d.pop('key')` – `d` lüğətindən `"key"` açarlı elementi silir və onun dəyərini qaytarır.

```
>>> silinen_element = d.pop('b')
>>> d, silinen_element
({'a':1,'c':3,'d':4}, 2)
```

- `d.copy()` – `d` lüğətinin nüsxəsini qaytarır. Siyahılar kimi, lüğətlər də dəyişiləbilən olduqları üçün təsadüfi dəyişmələrin qarşısını almaq məqsədi ilə bu metoddan istifadə etmək olar.

```
>>> d2 = d.copy()
>>> d2 == d    # True
```

- `d.keys()` – `d` lüğətinin açarlarından ibarət olan, "siyahıya bənzər" bir `dict_keys` obyekti qaytarır. Bu "siyahıya-bənzər" obyekti `list()` funksiyasına ötürərək, açarlar siyahısını ala bilərik.

```
>>> d.keys()
dict_keys(['a', 'c', 'd'])
>>>list(d.keys())
['a', 'c', 'd']
```

- `d.values()` – `d` lüğətinin dəyərlərini qaytarır. `keys` metodu kimi, `values` metodu da siyahıya-bənzər obyekt qaytarır, odur ki yenə də `list` funksiyasından yararlana bilərik.

```
>>> d.values()
dict_values([1, 3, 4])
>>> list(d.values())
[1, 3, 4]
```

- `d.items()` – `d` lüğətinin açar – dəyər cütlərindən ibarət yazılar siyahısını qaytarır. Hər bir yazı–lüğətin bir elementinin açar və dəyərindən ibarətdir. Öncəki iki metod kimi, bu metod da özünə məxsus `dict_items` obyekti qaytarır.

```
>>> d.items()
dict_items([('a', 1), ('c', 3), ('d', 4)])
>>> list(d.items())
[('a', 1), ('c', 3), ('d', 4)]
```

- `d.setdefault('a', 0)` – `'a'` açarlı element `d` siyahısında olarsa onun dəyərini qaytarır. Elə bir element olmazsa, yaradılır və onun dəyəri ikinci parametr kimi ötürülmüş verilən olur, bundan sonra onun dəyəri qaytarılır. İkinci parametr optimaldır, yəni ötürülməyə də bilər. Bu zaman qiyməti `None` götürülür.

```
>>> d.setdefault('a', 9) # 'a' açarlı element mövcud olduğu üçün sadəcə
1                        # qiyməti qaytarılır
>>> d.setdefault('j', 9)   # 'j' mövcud olmadığı üçün yaradılır və
9                          # dəyəri qaytarılır
>>> d.get('j')
9
```

```
>>> d.setdefault('u')  # 'u' mövcud deyil və ikinci parametr=None
>>> d
{'a':1, 'c':3, 'd':4, 'j':9, 'u':None}
```

- `d.update(D)` – `d` lüğətini `D` lüğəti ilə yeniləyir. Əgər element `D`-də var, amma `d`-də yoxdursa, o zaman həmin element `d`-ə əlavə edir. Əgər element hər iki lüğətdə varsa, o zaman onun dəyəri `D` lüğətindəki dəyərə bərabər olur.

```
>>> d.update({'c':0, u: -1, z:99})
>>> d
{'a':1, 'c':0, 'd':4, 'j':9, 'u':-1, z:99}
```

- `d.clear()` – `d` lüğətini təmizləyir.

```
>>> d
{'a':1, 'c':0, 'd':4, 'j':9, 'u':-1, z:99}
>>> d.clear()
>>> d
{}
```

> **Tip**
>
> Python 3.7-ə qədər lüğətlərdə elementlərin ardıcıllığı saxlanılmırdı. Yəni, siz lüğəti `{'a': 1, 'b': 2}` kimi t'yin edirdiniz, lüğət isə çevrilib `{'b': 2, 'a': 1}` ola bilirdi. Python 3.7 və daha sonrakı versiyalarda isə daha ardıcıllıq pozulmur.

2.9. Çoxluqlar

Çoxluqlar – təkrar elementləri olmayan ardıcıllıqdırlar. Burdakı çoxluqlar, riyaziyyatdan tanıdığımız çoxluqlar kimidirlər. **Yaradılması**. Çoxluq yaratnağın iki yolu var: `{}` və `set` funksiyası. Çoxluğu elementləri `{}` daxilində sadalamaqla yaratmaq olar. Əgər sadalanan elemenlər arasında təkrar elementlər olarsa, avtomatik olaraq onlar silinir. Siyahılar və yazılardan fərqli olaraq, çoxluqlarda hər hansı ardıcıllıq(düzülüş) yoxdur. Bu isə o deməkdir ki, siz onlara indekslə müraciət edə bilməzsiniz.

```
>>> digits = {1,1,3,6,7,1}    # təkrarlar silinir
>>> digits
{1,3,6,7}
>>> type(digits)
<class 'set'>
```

İndi isə eyni şeyi set funksiyası ilə etməyə çalışaq. Bunun üçün funksiyaya elementlərdən ibarət ardıcıllıq(sətr, siyahı, yazı və s.) ötürmək lazımdır.

```
>>> digits2 = set([2,4,5,5,8,8])
>>> digits2
{8, 2, 4, 5}
>>> set("aabbcd")
```

```
{'c', 'b', 'd', 'a'}
>>> set(('a', 'a', 'b', 'c'))
{'a', 'b', 'c'}
```

Qeyd edim ki, ola bilsin ki siz kodu icra etdikdə elementlərin ardıcıllığı fərqli ola bilər. Çünki, çoxluqlarda ardıcıllıq şərtidir. Yadınızdadırsa, boş lüğəti `d={}` kimi yaradırdıq, yəni boş dalğalı mötərizələr bizə boş lüğət qaytarır. Bəs boş çoxluğu necə yaratmalı? Bunun üçün `set` funksiyasını boş çağırmaq olar:

```
>>> d = {}
>>> type(d)
<class 'dict'>
>>> d2 = set()
>>> type(d2)
<class 'set'>
```

Çoxluq metodları

İki çoxluğumuz var:

```
>>> st, st2 = {1,2,3,0}, {3,1,5,6}
```

Tez-tez istifadə olunan bəzi çoxluq metodlarına baxaq:

- `st.pop()` və `st.remove(e)` – `pop` metodu `st` çoxluğunun ilk elementini silib qaytarır, heç bir parametr almır. `remove` metodu isə ötürülən `e` elementini silir, əgər element çoxluqda olmazsa, `KeyError` xətası alırıq:

```
    >>> st.pop()       # sağdan ilk element olan 1 elementi silinir və
qaytarılır
    1
    >>> st.remove(0)   # 0 elementini silir.
    >>> st             # {2,3}
```

- `st.add(e)` – çoxluğa `e` elementini əlavə edir:

```
    >>> st.add(0)
    >>> st   # {2,3,0}
```

- `st.discard(e)` – `e` elementi çoxluqda varsa silir, əks halda heç nə etmir(xəta baş vermir)

```
    >>> st    # {2,3,0}
    >>> st.discard(99)    # heç nə baş vermir.
    >>> st.discard(0)     # 0 silinir
    >>> st                # {2,3}
```

- `st.update(seq)` – `st` çoxluğunu `seq` ardıcıllığı ilə yeniləyir. Yəni, əgər ardıcıllığın hər hansı elementi çoxluqda olmazsa əlavə olunur. Yenə də, ardıcıllıq kimi sətr, siyahı, yazı və yaxud başqa bir çoxluq istifadə etmək olar:

```
>>> st              # {2,3}
>>> st.update([1,0])
>>> st              # {2,3,1,0}
```

- `st.union(st2,st3 …)` – `st` çoxluğunun ötürülmüş `st2`, `st3` və sairə ilə birləşələrini qaytarır. `st` özü dəyişmir:

```
>>> st.union({4,5},{7,8,9})
{2,3,1,0,4,5,7,8,9}
```

- `st.difference(st2,st3 …)` – `st` çoxluğunun ötürülmüş `st2`, st3 `və` sairə ilə fərqini qaytarır. `st` özü dəyişmir. Təqribən belə işləyir: st çoxluğunun st2-dən fərqi tapılır, tapılmış fərqin `st3` -dən fərqi tapılır və beləcə sonadək:

```
>>> st      # {2,3,1,0}
>>> st.difference({2,3,1})    # 0
>>> st.difference({2,1}, {1,0})   # 3
>>>(st.difference({2,1}, {1,0}) == \
            (st.difference({2,1}).union(st.difference({1,0}))))
True
```

- `st.intersection(st2,st3 …)` – `st` çoxluğunun ötürülmüş `st2`, `st3` və sairə ilə kəsişməsini qaytarır. `st` özü dəyişmir. İki çoxluğun kəsişməsi – bu çoxluqların ortaq elementlərindən ibarət olan bir çoxluq qaytarır:

```
>>> st.intersection({1,4,5,2})   # st={2,3,1,0}
{1,2}
```

- `st.issubset(st2)` – `st` çoxluğu `st2` çoxluğunun altçoxluğu olarsa – `True`, əks halda `False` qaytarır. Altçoxluq nə deməkdir: əgər `A` çoxluğunun bütün elementləri `B` çoxluğunda varsa, o zaman `A` çoxluğu, `B` çoxluğunun altçoxluğudur:

```
>>> st    # {2,3,1,0}
>>> st.issubset({9,2,6,3,1,0})   # True
>>> st.issubset({9,2,6,3,1,7})   # False
```

- `st.issuperset(st2)` – öncəki `issubset` metoduna bənzəyir. Ancaq burada `st - st2` -nin altçoxluğu deyil, `st2 - st` çoxluğunun altçoxluğu olmalıdır. Ümumiyyətlə `superset` o çoxluqdur ki özündə başqa bir çoxluğu saxlayır. Əgər A çoxluğu, B çoxluğunun altçoxluğudursa - B çoxıuğu da A-nın "superset"-i olur:

```
>>> st     # {2,3,1,0}
>>> st.issuperset({1,0})      # True
>>> st.issuperset({3,1,7})    # False
```

- `st.isdisjoint(st2)` – `st` çoxluğu ilə `st2` çoxluğunun heç bir ortaq elementi yoxdursa – `True`, əks halda isə `False` qaytarır:

```
>>> st.isdisjoint({4,5,6,7})    # True
>>> st.disjoint({4,5,6,0})      # False – ortaq 0 elementi var
```

- `st.symmetric_difference(st2)` – `st` və `st2` çoxluqlarının simmetrik fərqini qaytarır. `A` ilə `B`-nin simmetrik fərqi = `A`-nın `B`-dən fərqi birləşir `B`-nin `A`-dan fərqi:

```
>>> st2 = {2,3,7,8}
>>> st.symmetric_difference(st2)
{0, 1, 7, 8}
>>> st.difference(st2).union(st2.difference(st))
{0,1,7,8}
```

Bundan başqa, copy və clear metodları da var, hansılar ki öncəki verilən tiplərindəki eyni adlı metodlar kimi işləyir. Odur ki, daha bunun üçün yer sərf etməyəcəm. Son olaraq belə bir hala baxaq: 3 qrup tələbə var – riyaziyyat, tibb və ədəbiyyat dərnəklərini ziyarət edən tələbələr. Hər qrupda 3 tələbə var. Gəlin hər qrupu bir çoxluq kimi yazaq:

```
>>> math = {'Anar', 'Ali', 'Camal'}    # riyaziyyat
>>> med = {'Ali', 'Leyla', 'Fariz'}    # tibb
>>> lit = {'Anar', 'Nigar', 'Arzu'}    # ədəbiyyat
```

İndi isə, gəlin sual-cavab oynayaq:

sual	cavab
həm riyaziyyat, həm də tibbə gedənlər	`math.intersection(med)`
riyaziyyata gedib, ədəbiyyata getməyənlər	`math.difference(lit))`
hər üç dərnəyə gedənlər	`math.intersection(lit, med)`

İndi bir az da qəlizləşdirək: yalnız və yalnız bir dərnəyə üzv olan tələbələri tapaq. Bunun üçün, hər hansı iki qrupun simmetrik fərqini tapaq, daha sonra isə alınan çoxluğun üçüncü çoxluqdan simmetrik fərqini tapaq:

```
>>> math.symmetric_difference(lit).symmetric_difference(med)
{'Leyla', 'Arzu', 'Camal', 'Nigar', 'Fariz'}
```

> ⚠ **Warning**

Diqqət: A ilə B-nin fərqi və B ilə A-nın fərqi fərqlidir!

```
>>> st.difference({2,3,1}) == {2,3,1}.difference(st)
False
```

Çoxluqlar üçün bəzi operatorlar.

Python 3.10-dan başlayaraq `|` operatorundan çoxluqlar üçün biləşdirmə əməliyyatıını aparmaq üçün istifadə edilir. Operatorlar vasitəsilə, çoxluq metodlarları ilə etdiyimiz əksər əməliyyatları icra etmək olar. Bu operatorlar binardırlar - iki çoxluq arasında icra olunurlar. Elə isə, bu operatorlara qısaca baxaq:

- `|` - birləşdirmə
- `&` - kəsişmə
- `-` - fərq
- `^` - simmetrik fərq
- `>=` - superset
- `<=` - subset(altçoxluq)
- `<` - proper-subset
- `>` - proper-superset

Son ikisi haqqında bir az danışalı olacam, belə ki `subset` və `proper subset` aralarındakı fərqi göstərmək istəyirəm. A çoxluğu, B çoxluğunun alt çoxluğudursa, burada iki hal ola bilər - B çoxluğunun içərisində A-dan fərqli elementlər də var(`B != A`) və B çoxluğundakı bütün elementlər elə A çoxluğunda da var(`A == B`). Birinci hal - proper subset adlanır, yəni, birinci çoxluq ikincinin altçoxluğudur, amma ona bərabər deyil. Bu məqamda niyə məhz `<` və `<=` operatorlarından istifadə edildiyi bəlli olur.

```
>>> s1 = {2, 5, 3, 7, 'c', 'a', 8}
>>> s2 = {3, 7, 8, 'c'}
>>> # union
>>> s1 | s2
{2, 3, 5, 7, 8, 'a', 'c'}
>>> # intersection
>>> s1 & s2
{8, 'c', 3, 7}
>>> # difference
>>> s1 - s2
{2, 5, 'a'}
>>> # symmetric difference
>>> s1 ^ s2
{2, 5, 'a'}
>>> # subset - s2==s1 də ola bilər
>>> s1 <= s2
False
>>> # proper-subset. burda isə s1, s2-nin alt
>>> s1 < s2    # çoxluğudur, s1 != s2
False
```

```
>>> # superset, eyni şey supersetlət üçün.
>>> s1 >= s2
True
>>> # proper superset
>>> s1 > s2
True
```

Sonda

Son olaraq bunları da qeyd etmək olar: tip çevirmələri barədə danışanda set funksiyasını da yad etmişdik. Bu funksiya bildiyiniz kimi ardıcıllığı çoxluğa çevirir. Okay, bu bizə nələr verə bilər? Artıq bildiyiniz kimi, çoxluqlarda təkrarların olmaması əhəmiyyətli xüsusiyyətdir və biz bundan istifadə edə bilərik. Məsələn, hər hansı siyahıdan təkrarları aradan qaldırmaq istəyiriksə, bunun üçün siyahını çoxluğa, daha sonra isə yenidən siyahıya çevirə bilərik:

```
>>> some_list = [1,1,2,3,3,1]
>>> set(some_list)               # {3,2,1}
>>> some_list = list(set(some_list))
```

Tapşırıqlar

Ədədlər

1. Fərqli ədədi tiplər üçün `num_int`, `num_float`, və `num_complex` dəyişənlərini yaradın.
2. İki ədəd dəyişəninin cəminin nəticəsini yeni `sum_result` dəyişənində saxlayın.
3. İki ədədin fərqini saxlamaq üçün `difference` dəyişəni yaradın.
4. İki ədədin hasilini saxlamaq üçün `product` dəyişəni yaradın.
5. Bölmənin nəticəsini saxlamaq üçün `quotient` dəyişənindən istifadə edin.
6. Modul operatorundan istifadə edərək bölmənin qalığını `remainder` dəyişənində saxlayın.
7. Tam bölmədən istifadə edərək `floor_result` dəyişəni yaradın.
8. Bir ədədin qüvvətə yüksəldilməsinin nəticəsini `power_result` dəyişənində saxlayın.
9. Mənfi bir ədədin mütləq qiymətini saxlamaq üçün `abs_value` dəyişəni yaradın.
10. Həqiqi ədədin (float) yuvarlaqlaşdırılmış versiyasını `rounded_num` dəyişənində saxlayın.

Sətirlər

1. "El bilir ki sən mənimsən" sətrini bir dəyişənə mənimsədim, uzunluğunu çap edin.
2. Dilimləmə (slicing) istifadə edərək `substring` dəyişənində bir altsətr saxlayın.
3. Sətrinizi böyük hərflərə çevirərək `uppercase_text` dəyişəni yaradın.
4. Müəyyən bir simvolun sayını `char_count` dəyişənində saxlayın.
5. Sətrinizdə "El" sözünü "Aləm" sözü ilə əvəz edərək `replaced_text` dəyişəninə mənimsədin.
6. Sətrinizi sözlərə bölün və nəticəsini `word_list` dəyişənində saxlayın.
7. `colors_str = "red,black,yellow"` dəyişəni var. Sətrdən rənglər siyahısı alın: `['red', 'black', 'yellow']`
8. Ancaq dilimləmə istifadə etməklə, `"-1-2-3-4-5"` sətrindən `"12345"` sətrini alın.
9. `greeting = " salam dünya \n"` dəyişəni verilib. Sətrin hər iki kənarındakı boşluqları silin.

10. Sətr tipli `var` adlı dəyişən verilib. Dəyişənin dəyəri olan sətrin düzgün dəyişən adı olun-olmamasını təyin edin.

Yazılar (tuples)

1. 4 rəng adından - "red", "blue", "green", "blue" ibarət olan yazı yaradın.
2. İkinci elementi `second_color` dəyişənində saxlayın.
3. "blue" (göy) rənginin neçə dəfə göründüyünü saxlamaq üçün `blue_count` dəyişəni yaradın.
4. "red" (qırmızı) rənginin indeksini `red_index` dəyişənində saxlayın.
5. `tuple1` və `tuple2` dəyişənlərini yaradın, sonra onları `combined_tuple` dəyişənində birləşdirin.
6. Bir iç-içə yazı yaradın və `nested_tuple` dəyişəninə mənimsədin.
7. Ədədlərdən ibarət bir yazı yaradın və onu dilimləmə vasitəsilə əksinə düzün.
8. `list_data` adlı bir siyahı dəyişəni yaradın və onu `tuple_from_list` dəyişənində yazıya çevirin.
9. Tək elementli bir yazını `single_item_tuple` dəyişənində saxlayın.
10. `nums=(1,2,3,4,5,6)` yazısı verilib. Elə edin ki `nums` dəyişənində ancaq cüt ədədlər qalsın.

Siyahılar

1. Özümdə [1-20] aralığındakı ədədləri saxlayacaq `nums` siyahısını yaradın.
2. `nums` siyahısının ancaq cüt elementlərindən ibarət olacaq `nums_even` siyahısını yaradın.
3. `nums_even` siyahısının son elementini silərək `last_even` dəyişəninə mənimsədin.
4. `nums` siyahısının 2-ci mövqeyinə (indeks 2) `last_even` ədədini daxil edin.
5. Siyahını artan sıra(kiçikdən-böyüyə doğru) ilə çeşidləyin.

Məntiqi (boolean)

1. iki ədəd dəyişən yaradın, birinin dəyəri `True`, digəri isə `False` olsun. `type` funksiyasından istifadə etməklə, onların tipini yoxlayın.
2. Deyək ki müəyyən bir kitab var. Kitabın xüsusiyyətləri üçün bu dəyişənləri var: janrı - `genre`, müəllifi - `author` və üz qabığının materialı - `cover_material`. Janrı - "philosophy", müəllifi "Freud" və ya "Jung", üz qabığının materialı isə "hard" olan kitablar üçün `True` almaq üçün bir məntiqi ifadə yazın.

Çoxluqlar

1. `{'apple', 'pear'}` və `{'tomato', 'potato'}` çoxluqları verilib. Onların birləşməsini tapıb `eatable` dəyişəninə mənimsədin.
2. Deyək ki `A = {1, 2, 3, 4, 5}`, `B = {4, 5, 6, 7, 8}` və `C = {3, 5, 7, 9}` çoxluqları verilib. Aşağıda verilənləri tapın:
   - Hər üç çoxluqda peydə olan elementləri
   - Ancaq `A` çoxluğunda olan elementləri
   - `B` və `A` çoxluqlarında olub, `A` çoxluğunda olmayan elementləri

Lüğətlər

1. Universitet qrupunda olan tələbələr: Ülvi İsmayılov, Hikmət Nəsibov və Rəşad Vəliyev. Yaşları isə uyğun olaraq - 33, 25 və 35 olacaq. Tələbələri lüğətlər siyahısı kimi göstərməyə çalışın: hər bir lüğət - bir tələbə olacaq.
2. Siyahını elə dəyişin ki Rəşadın yaşı 5 vahid azalsın, yəni 30 olsun.
3. Siyahıya yeni tələbə - Əziz Nadirov 27 yaş əlavə edin.

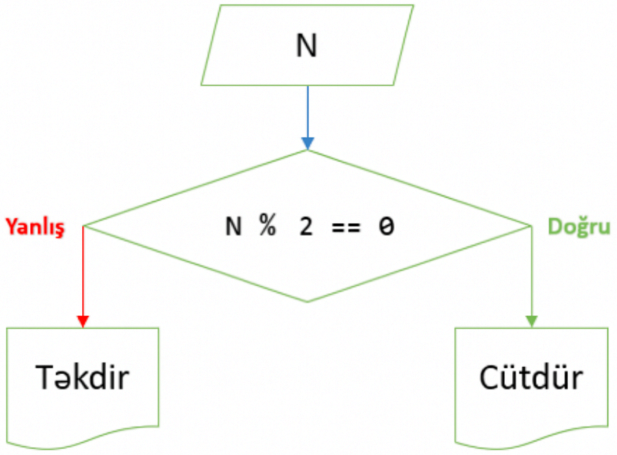
Link(həllər): https://github.com/AzizNadirov/python-road-book/blob/master/code/tasks/datastructures_2.ipynb

3. Şərti budaqlanma

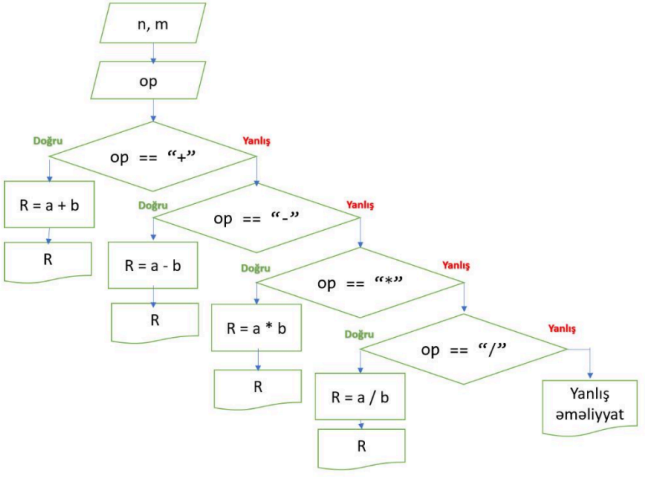
> **Info**
>
> Bu bölmədən etibarən, artıq daha "mənalı" kodlar yazmağa başlayacağıq. Öncə dediyim kimi, ilk mərhələlərdə cansıxıcı və əzbərçilk-sayağı görünə bilər, lakin irəlilədikcə - daha maraqlı, düşündürücü və hətta deyərdim ki əyləncəli məqamlar olacaq. Demək olar ki, bu bölmədən etibarən artıq yazdığımız proqram kodu seçim etmə imkanına malik olacaq. Nə isə, söhbəti uzatmayıb, başlayıram.

3.1. Seçim

Gəlin özümüzü bir az filosof kimi aparaq və qərarlar və onların həyatımıza təsiri barədə düşünək. Məsələn, hansı qərar nəticəsində bu kitabı oxumağa qərar verdin? Ola bilər ki, sadəcə vərəqləmək üçün əlinə aldın və marağını çəkdi, nəticədə hələ də oxuyursan(ya da sadəcə ortalardan açıb bu hissəyə tuş gəlmisən). Əsas məsələ odur ki, kitab əgər kitab maraqlı deyilsə, onu oxumayacaqsan. Yəni, hal-hazırda sənin ən azından iki seçimin var: kitabı oxumağa davam etmək və kitabı kənara qoymaq. Bu və ya digər seçimi etməyin isə kitabın sənin üçün maraqlı olub-olmayacağından asılıdır, yəni bu şərt bir növ bu və ya digər qərar verməyimizə bir başa təsir edir. Həyatımızda atdığımız bütün addımlar, qərarlar məhz bu cür – müəyyən şərtlər altında qəbul edilmiş qərarlardan ibarətdir. Seçdiyiniz və ya seçəcəyiniz universitet, ixtisas, tanışlıqlar bunların hamısı. Əgər seçim imkanı olmasa idi, həyat nə qədər cansıxıcı olardı! Bu yerdə, filosof modundan çıxıb qayıdaq yer kürəsinə. Ən sadə halda, geyinəcəyimiz paltar – çöldəki hava şəraitindən asılı olacaq. Bu məqamda razılaşa bilərik ki, etdiyimiz seçimlər doğurdan da vacibdir. Ümumiyyətlə, seçim etmək imkanının özü - sənə hadisələrin gedişatını təyin etməyə, idarə etməyə imkan verir. İndi isə gəl düşünək: proqramlaşdırma dilində kod yazarkən – icra olunacaq bir gedişat hazırlayarkən seçim edə bilmək nə qədər faydalı olar? Real həyatda olduğu kimi, proqram gedişatında da müəyyən şərt daxilində qərar qəbul edə bilərik. Gəl belə bir şey təsəvvür edək: Dəhşətli dərəcədə tənbəl bir adam var və bu adam heç cürə tək və cüt ədədləri ayırd edə bilmir və bizim məqsədimiz – bir proqramçı kimi bu adama verilmiş ədədin tək və ya cüt ədəd olduğunu xəbər edəcək proqram yazmaqdır. İlk öncə proqramlaşdırma dilini qoyaq kənara, ağ-qara vərəq üzərində bir şeylər cızmağa çalışaq. Əvvala: bizdən nə tələb olunur – "verilmiş ədədin tək və ya cüt olduğunu təyin edən proqram", ədədin tək və ya cüt olduğunu necə təyin edə bilərik – "əgər ədəd ikiyə tam bölünürsə - cütdür, əks halda isə təkdir.". İkinci hissə bizim şərtimizdir. "əgər ədəd ikiyə tam bölünürsə - cütdür, əks halda isə təkdir" – şərtini daha dəqiq – riyazi formada necə yaza bilərik? Belə: `"əgər, ədəd % 2 = 0 olarsa ədəd cütdür, əks halda - ədəd təkdir"`. O qədər də güclü riyaziyyat lazım olmadı, gördüyün kimi. Yadındadırsa, sənlə alqoritmlər və onların blok-sxem ilə təsviri barədə də danışmışdıq, indi isə, yuxarıda verbal – söz ilə yazdığımızı blok sxem kimi yazmağa çalışaq:



Gödüyün kimi, bir az da qabağa gedib, şərt hissəsini python sayağı – məntiqi ifadə şəklində yazmışam. `N%2==0` şərti ödənərsə ifadə - `True` əks halda isə - `False` qaytarır. Uyğun olaraq ədədin cüt və ya tək olması barədə məlumat çap olunur. İndi isə məsələni bir az qəlizləşdirək. Çox sadə bir hesablayıcı proqramının blok-sxemini quraq. Məsələ belədir: dörd ədəd əməliyyat mövcuddur – toplama, çıxma,vurma və bölmə. İstifadəçi iki ədəd – `a` və `b` daxil edir, daha sonra ədədlər arasında icra olunmalı olan əməliyyatı daxil edir, əməliyyat dedikdə `+`, `-`, `*`, `/` simvollarından biri nəzərdə tutulur.



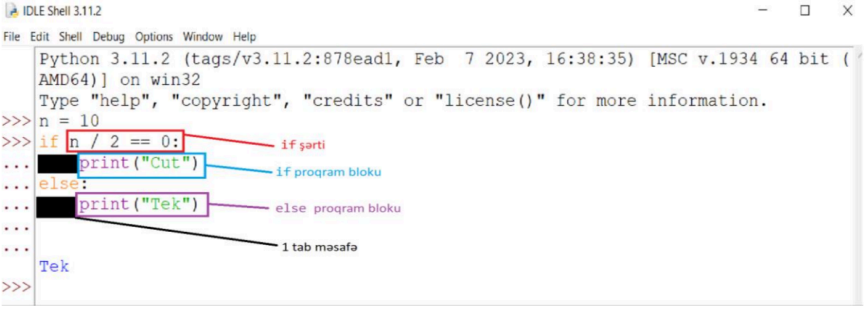
Blok-sxemdən göründüyü kimi, növbə ilə op dəyişənini yoxlayaraq, uyğun hesablama əməliyyatını aparıb, nəticəni çap edirik. Yuxarıdakı nümunədə icra olunan "əməliyyat blokunda" sadəcə olaraq bir ifadə - `R =a...` yazılıb, ancaq burada heç bir məhdudiyyət yoxdur, bu o deməkdir ki, ixtiyarı sayda hesablama, şərt və sairə bloklar ola bilər.

3.2. Şərti budaqlanma: if-else

İndi isə qayıdaq Pythona. Az öncə blok-sxemlərlə təsvir etdiyimiz alqoritmləri necə kodlaya bilərik ? Bu məqsədlə əksər proqramlaşdırma dillərində, o cümlədən də Pythonda da if-else operatorları mövcuddur. `if` – ingilis dilindən "əgər" kimi, `else` – "əks halda" kimi tərcümə olunur. Operatorların sintaksisi belədir:

```
if <şərt>:
    <doğru halda icra olunacaq proqram bloku>
else:
    <yanlış halda icra olunacaq proqram bloku>
```

`<şərt>` hissədə məntiqi ifadə yerləşir, ifadənin `True` qaytardığı halda `<doğru halda icra olunacaq proqram bloku>` hissəsində yazılan proqram kodu icra olunur, `False` qaytardığı halda isə `<yanlış halda icra olunacaq proqram bloku>` hissədə yazılan proqram kodu icra olunur. İndi isə, öncəki nümunədə verilmiş ədədin tək və ya cüt olduğunu çap edən alqoritmi Pythonda yazmağa çalışaq.



Sadə if-else bloku

Yuxarıdakı şəkildə diqqət edilməli bir neçə məqam var:

- `if` operatorunun məntiqi ifadəsindən sonra `:` işarəsi qoyulur, növbəti sətrdən etibarən artıq `if` operatorunun proqram bloku başlayır. Yəni, məntiqi ifadə `True` qaytardığı təqdirdə həmin blokda yerləşən bütün proqram kodu icra olunacaq. Proqram blokunun içərisində istənilən həcmdə proqram kodu yazıla bilər. Hətta, proqram blokunda ayrıca `if - else` istifadə etmək olar.
- Proqram blokunun hansı operatora aid olduğunu göstərmək üçün proqram bloku aid olduğu operatora nəzərən bir tab, yəni dörd boşluq sağda yerləşir. Həmin boşluqlar şəkildə qara rənglə verilib. Bəzi proramlaşdırma dillərində proqram bloklarını göstərmək üçün fiqurlu mötərizələrdən istifadə olunur, Pythonda isə mötərizələr deyil, sol tərəfdən buraxılan tablar istifadə olunur.:

```
    if (a > b){
        …
    }
    else {
        …
    }
```

3.3. Şərti budaqlanma: `if` - `elif` - `else`

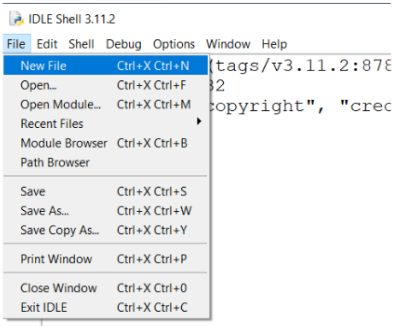
Sadə `if-else` strukturunda şərt ödəndikdə `if`, ödənmədikdə isə else icra olunur. Amma, bəzən bu bəs etmir. `if` şərti ödənmədikdə, əlavə şərtlər qeyd edə bilərik, bunu `elif` operatoru ilə edirik. `elif` – *elseif* sözündən yaranıb, yəni "əks halda əgər" kimi tərcümə etmək olar.

```
if <condition_1>:
    <block 1>
elif <condition_2>:
    <block 2>
elif <condition_3>:
    <block 3>
……
else:
    <block 4>
```

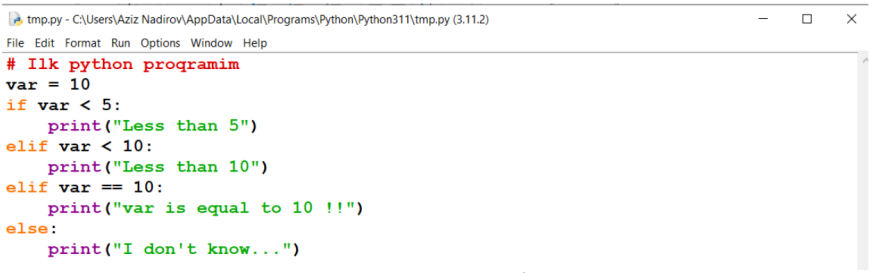
`elif`-lər həmişə `if`-dən sonra yazılır, çünki məna etibarilə `elif` opertaoru `if` operatorunun alternatividir. `If` ödənərsə, eliflər işə düşmür. `İf` ödənməzsə, yuxarıdan aşağıya doğru `elif`-lər yoxlanılacaq və şərti ödənən ilk `elif` operatoru işə düşür. Heç bir `elif` şərti ödənməzsə, `else` işə düşəcək. Yəni, əgər `else` operatorunun üzərindəki `if` və `elif`-lərin heç birinin şərti ödənməzsə, `else` bloku icra olunacaq. Bu `if-elif-else` zəncirində istənilən sayda `elif`-lər ola bilər.

3.4. Pythonda modullar - birinci hissə

İndiyə qədər yazdığımız bütün kodları interaktiv mühitdə yazmışıq. Bu isə, bir o qədər də rahat deyil. Çünki, orda yazılan kod "birdəfəlikdir". "Python yüklənməsi və quraşdırılması" barədə danışarkən bu mözuya ötəri toxunmuşduq. İndi isə bir qədər yaxından tanış olaq. Yazılan proqram kodlarımızı təkrar istifadə etmək istəyiriksə, onları bir fayl kimi yadda saxlamaq lazımdır. Özündə python kodunu saxlayan fayl – python modulu adlanır və .py genişlənməsinə sahib olur. Əslində bu fayl adi mətn faylıdır – sadə .txt faylıdır, sadəcə içərisindəki mətn python dilində yazılmış koddur. Bu o deməkdir ki, siz python modullarını adi notepad və ya hətta ən pis halda Word ilə də açа bilərsiniz. Amma, yaxşı xəbər ondan ibarətdir ki, bizə nə notepad, nə də Word lazım deyil, çünki bizim Python IDLE-ımız var! Elə IDLE içərisindən yeni modul yarada – `New File` (qısaca `CTRL + N`) və ya artıq mövcud olan faylı açа – `Open` (`CTRL+O`) bilərik. Fayl daxilindəki kodu işə salmaq üçün, modulu –



icra etmək lazımdır. Bunun üçün `Run` / `Run Module` və ya qısaca `F5` düyməsini sıxmaq(`F5` ilə düşmürsə, ola bilər ki kompüterdə funksional düymələr bağlanıb, belə olan təqdirdə `Fn+F5` sınayın) lazımdır. Aşağıda `tmp` adlandırılmış modulda sadə `if-elif-else` istifadəsi verilib:



Biz modullar barədə hələ danışacağıq, amma yerli-yerində. İndi isə şərti-budaqlanmaya davam edək.

3.5. Şərti budaqlanma, input funksiyası

İstifadəçidən məlumat almaq üçün `input` funksiyasından istifadə olunur. Funksiyaya arqument kimi – sətr ötürmək olar. Bu sətr, istifadəçiyə bir ismarıc kimi görünəcək. Funksiyanın geriyə qaytardığı dəyər – istifadəçinin daxil etdiyi verilənin sətr variantı olacaq. Bu o deməkdir ki, əgər istifadəçi 5 ədədini daxil edərsə, input bizə integer tipli 5 ədədini deyil str tipli "5" qaytaracaq. Ona görə də, əgər istifadəçidən ədəd gözləyirsinizsə, mütləq int funksiyası ilə tam ədədə çevirmək lazıdır.

```
name = input("what is your name: ")
print(f"Nice to meet you, {name}!")
age = int(input("How old are you? "))
if age < 10:
    print("You are pretty child ))")
elif age > 100:
    print("You are ancient as Monalisa ))")
else:
    print("okay!")
```

```
what is your name: Aziz
Nice to meet you, Aziz!
How old are you? 101
You are ancient as Monalisa ))
```

Gördüyünüz kimi, `input` funksiyasını bir başa olaraq `int` funksiyasını içərisində çağırmışam. Nəticədə - istifadəçi ədəd daxil edir, ədəd sətr kimi input funksiyası tərəfindən qaytarılır və bir başa düşür `int` funksiyasının içərisinə. `int` funksiyası da öz növbəsində sətri ədədə çevirməyə çalışır. Öncə də dediyim kimi, bu `if-else` konstruksiyalarını istənilən dərinlikdə qura bilərik.

```
answer = input("Sizi qeydiyyata alaq? y/n)
if answer == 'y':
    name = input("Ad: ")
    surname = input("Soy ad: ")
    answer_email = input("Size email yaradaq? ")
    if answer_email == 'y':
        email = name + "." + surname + "@gmail.com"
        print(f"emailiniz hazirdir: {email}")
    else:
        print("Emailsiz olaraq qeydiyyata alindiniz.")
else:
    print("Qeydiyyat leghv olundu…")
```

```
Sizi qeydiyyata alaq? y/n y
Ad: Aziz
Soy ad: Nadirov
```

```
Size email yaradaq? y
emailiniz hazirdir: Aziz.Nadirov@gmail.com
```

Sonda, bunları yadda saxlamaq lazımdır:

1. Tablara(soldan buraxılan boşluqlara) diqqət et. Yadda saxla ki python məhz boşluqlara əsasən blokların başlanma və bitmə nöqtələrini başa düşür.
2. `if` - `elif` - `else` üçlüyü ardıcıldır. Bu o deməkdir ki, `if` olmadan nə `elif`, nə də `else` istifadə edə bilməzsən. Çünki, `elif` və `else`, `if`-in altternativləridirlər.
3. Şərt hissəsində istənilən məntiqi ifadə yazmaq, `and`, `or`, `not` məntiqi operatorlarından istifadə etmək olar. Bəzən iç-içə yazılmış `if` şərtlərini ixtisar edərək, bir `if` şəklində yazmaq mümkündür:

```
if a > 5:
    if a < 10:
        print("a ededi 5-10 araligindadir")
```

yuxarıdakı proqram kodunu belə də yazmaq mümkündür:

```
if a > 5 and a < 10:
    print("a ededi 5-10 araligindadir")
```

Tapşırıqlar

1. İstifadəçi `input` funksiyası ilə ədəd daxil etməlidir. Ədəd 10-a tam bölünənə olarsa `"Onluq!"`, 5-ə bölünən olarsa `"Beşlik!"`, 3-ə bölünən olarsa `"Üçlük"`, 2 üçün `"Cütlük!"`, əks halda isə `"Nəsə başqa bir şey!"` çap edin.
2. istifadəçi iki ədəd daxil edir. Daha sonra istifadəçi `+`, `-`, `*` və ya `/` daxil edir. Daxil olunan əməliyyata uyğun olaraq ədədlərin cəmini, fərqini, hasilini və ya qismətini çap edin.
3. Daxil olunan sətrin polindrom olub-olmamasını çap edən proqram yazın. Polindrom sözlər, hər iki istiqamətdə - soldan-sağa və sağdan-sola eyni cür yazılan sözlərdir: ata, qapaq, mum və sairə. Link: https://github.com/AzizNadirov/python-road-book/blob/master/code/tasks/control_flow_3.ipynb

4. Dövrlər

Bəzən elə məsələlərlə üzləşirik ki, orada təkrar-bə-təkrar eyni əməliyyatı icra etməli oluruq. Belə əməliyyatlar – dövri əməliyyatlardır. Təsəvvür edin ki bir siyahımız var, bu siyahıda 9 ədədinin olub olmadığını yoxlamaq istəyirəm. Bunun üçün dövri olaraq siyahının hər bir elementini götürüb 9 ilə müqayisə edə bilərəm. Yaxud nə üçünsə 100 dəfə "hello dunya" çap etmək istəyirəm. Bu zaman bəlli məsələdir ki, 100 dəfə print yazmaq heç də həll yolu deyil. Bütün bu halları birləşdirən məqam - əməliyyatların dövri olmasıdır. Elə isə gəlin söhbəti uzatmad işə başlayaq!

4.1. while dövrü

Bu dövrü həm də ön şərtli dövr də adlandırırlar. Hərfi tərcüməsi "nə qədər ki" deməkdir. Bu dövr olduqca sadədir, dövrün başlığında məntiqi ifadə yazılır və nə qədər ki (fikir verin, while-ın tərcüməsi lap yerinə oturdu) məntiqi ifadə `True` qaytarır, dövrün proqram bloku icra ediləcək. Sintaksisi belədir:

```
while <şərt>:
    <proqram bloku>
```

`<shert>` hissəsinə istənilən məntiqi ifadə yazıla bilər. Proqram bloku istənilən sayda sətrdən ibarət ola bilər, içərisində şərti budaqlanma, dövrlər – istədiyiniz hər şeyi yaza bilərsiniz. Bütün bunlar hər bir dövr zamanı icra olunacaq. Belə bir listinqə baxaq:

```
n = 1
while n < 100:
    print(n)
```

Yuxarıdakı kodu işə salsaq, dayanmadan 1 çap edəcək. Çünki, dövrün şərti olan `n < 100` ifadəsi hər zaman ödənir. Dövr isə yalnız şərt ödənmədikdə dayanır. Buna görə də dövr sonsuzadək davam edir. Bu kodu işə salandan sonra sonsuzadək gözləməmək üçün `CTRL+C` sıxaraq proqramı məcburi şəkildə sonlandıra bilərsiniz. Ancaq, daha bir sətr əlavə edib kodu aşağıdakı kimi yazsaq, proqram 1-dən 100-ə qədər ədədləri çap etmiş olacaq:

```
n = 1
while n < 100:
    print(n)
    n = n + 1
```

Bu dəfə sonsuz dövr yaranmadı, çünki hər dövr zamanı n dəyişəninin qiyməti 1 vahid artır. Şərtə əsasən, `n` dəyişəninin qiyməti 100-ə çatan kimi dövrün şərti `False` qaytarır və dövr dayanır. Şərti budaqlanma mövzusunda olduğu kimi, burada da şərt daxilində məntiqi operatorlardan istifadə etmək olar. Gəlin indi `while` dövründən istifadə edərək, verilmiş ədədin rəqəmlərinin sayını hesablayan proqram yazaq.

```
num = int(input("Ədədi daxil edin: "))
count = 0
n = num
```

```
while n > 0:
    n = n // 10
    count = count + 1

print(f"{num} ədədinin rəqəmlərinin sayı: {count}")
```

```
ədədi daxil edin: 1234
1234 ədədinin rəqəmlərinin sayı: 4
```

Proqram necə işləyir: ixtiyari ədədin sonuncu rəqəmini atmaq üçün, ədədi 10-a tam bölmək lazımdı(ədəd div 10). Say üçün `count` adlı dəyişən yaradırıq və hər dövrdə `n` ədədinin soldan bir rəqəmini ataraq, `count` dəyişəninin qiymətini 1 vahid artırıram. Sonuncu rəqəmi atdıqdan sonra `n=10` olacaq. Bu isə dövrün şərtini ödəmədiyi üçün dövr sonlanır və count dəyişəninin son qiyməti ədədin rəqəmlərinin sayına bərabər olur. Alqoritmi bir qədər dəyişib, ədədin rəqəmlərinin cəmini də almaq olar:

```
num = int(input("ədədi daxil edin: "))
s = 0
n = num
while n > 0:
    digit = n % 10   # sonuncu rəqəmi alıram
    n = n // 10      # sonuncu rəqəmi atıram
    s = s + digit    # rəqəm ilə cəmləyirəm
print(f"{num} ədədinin rəqəmlərinin cəmi: {s}")
```

```
ədədi daxil edin: 134
134 ədədinin rəqəmlərinin cəmi: 8
```

4.2. break və continue

Adətən dövr şərt ödənməyənə qədər davam edir. Ancaq bəzi hallarda dövrü vaxtından öncə sonlandırmaq lazım gələ bilər. Bunun üçün `break` operatorundan istifadə olunur. `break` işə düşdükdə dövr sonlanır. Baxaq:

```
i = 0
while i < 100:
    print(i)
    if i == 5:
        break
    else:
        i = i + 1
```

Yuxarıdakı kodu işə salsaq, yalnzı ilk 5 ədəd çap olunacaq. Çünki, `i=5` olduqda, `if` operatorunun şərti ödənir və `break` işə düşür, dövr sonlanır. Gəlin bir az da irəli gedək, belə bir məsələ verilib: "Müəyyən bir array ardıcıllığı və item elementi verilir. item elementinin ardıcıllıqdakı indeksini çap edin, əgər element ardıcıllıqda yoxdursa, -1 çap edin ". Bu məsələni həll etmək üçün `while` dövrü ilə ardıcıllığın növbə ilə bütün elementlərini götürüb, axtarılan item ilə müayisə edəcəm.

```
array = [2, 7, 8, 9, 11]
item = 9
i = 0
while i < len(array):
    if array[i] == item:       # element tapıldı
        print(f"tapildi, indeks: {i}")  # çap et
        break        # dövrü sonlandır, əks halda sonsuz dövü yaranacaq
    else:
        i += 1     # tapılmasa, i = i+1,
else:
    print("-1")    # break işə düşməzsə
```

```
tapildi, indeks: 3
```

Yuxarıdakı nümunədə qəribəliyi gördünüz? Sonuncu sətrdəki else operatoru heç də `while`-ın içərisindəki `if`-ə aid deyil. İş orasındadır ki, dövr 2 üsulla sonlana bilər: normal – şərt ödənməyənə qədər və məcburən – `break` işə düşdükdə. Dövrlərlə birlikdə `else` operatorundan istifadə etmək olar, `else` o zaman işə düşəcək ki, dövr normal sonlanmış olsun. Yəni, əgər dövr məcburi şəkildə sonlanarsa, `else` işə düşməyəcək, normal şəkildə sonlanarsa, işə düşəcək. Qısası – `break` işə düşsə `else` işləmir, işə düşməsə `else` işləyəcək. Əslində, bir qədər qarışıq və yersiz məqamdı, bəziləri ümumiyyətlə, dövrlərlə `else` istifadə olunduğu barədə bilmir, bəziləri isə bunun yersiz bir şey olduğu barədə debatlar aparır. İndi qayıdaq yuxarıdakı nümunəyə, ardıcıllığın elementlərini indekslərinə görə almaq üçün indeks bildirəcək bir müvəqqəti dəyişənə ehtiyac var, bunun üçün `i=0` təyin edirəm. Dövrün şərtini `i < len(array)` olaraq təyin edirəm. Hər dövrdə `i` dəyişəninin qiymətini `1` vahid artırsam, `i` bu qiymətləri alacaq: `[0,1,2,3,4]`. O zaman, haqlı sual yarana bilər: nə üçün dövrün şərtini elə `i<5` olaraq yazmadıq, niyə məhz `i<len(array)` ? İş burasındadır ki, kodu çalışıb, maksimal dərəcədə "dinamik" tərzdə yazmaq lazımdır. Dinamik dedikdə, daha asan dəyişən, çevik nəzərdə tuturam. əgər kodun içərisinə bir başa olaraq 5 yazmış olsaydım, ola bilsin ki yaxın gələcəkdə özümə çətinlik yaratmış olardım. Çünki, əgər kodu yoxlamaq üçün array ardıcıllığını dəyişmiş olsaydım və ardıcıllıqdakı elementlərin sayı dəyişsəydi, məcbur olaraq "mismarladığım" 5 ədədini array ardıcıllığının uzunluğu ilə əvəz etməli olacaqdım. Ona görə də, bir başa olaraq, dəyişənin uzunluğunu hesablayan len funksiyası ilə təyin edirəm. `if` operatorunun şərtinin ödənməsi o deməksir ki, axtarılan element ardıcıllıqda tapılıb. Belə olan halda, özündə həmin elementin indeksini saxlayan i dəyişənini çap edib, dövrü sonlandırırıq. Əgər, ardıcıllıqdakı bütün `item`-a bərabər elementləri tapmağa çalışsaq, proqram kodunu bu cür yazmaq olar:

```
array = [1, 9, 9, 0, 7, 9]
item = 9
indexes = []
i = 0
```

```
while i < len(array):
    if array[i] == item:
        print(f"{i} indeksində.")
        indexes.append(i)
        i += 1
    else:
        i += 1
print(f"{array} ardıcıllığında {item} elementinin indeksləri: {indexes}")
```

```
1 indeksində.
2 indeksində.
5 indeksində.
[1, 9, 9, 0, 7, 9] ardıcıllığında 9 elementinin indeksləri: [1, 2, 5]
```

continue operatoru isə `break` -dən fərqli olaraq, dövrü dayandırmır, sadəcə cari dövrü yarıda dayandırıb, növbəti dövrə atladır. 1-10 aralığındakı cüt ədədləri çap edən proqram yazaq:

```
# 1-10 aralığındakı cüt ədədlərin kvadratını çap et
start = 1
stop = 10
i = start
while i <= stop:
    if i % 2 == 0:
        p = i ** 2
        print(p)
        i += 1
        continue
    else:
        i += 1
        continue
```

```
4
16
36
64
100
```

4.3. for dövrü

`for` dövrü `while` dövrünün bir növ bəzədilmiş formasıdır(buna ingilis dilində *syntax sugar* – sintaktik şəkər də deyilir). `for` dövrü müəyyən ardıcıllığın elementlərini *iterasiya etmək* üçün istifadə edilir. İterasiya etmək – ardıcıl götürmək kimi başa düşmək olar. Məsələn `sequence = [1,2,3,4,5]` ardıcıllığı verilib. Bu ardıcıllıqdakı elementləri bir-bir çap etmək üçün hər bir elementə indeksinə görə müraciət etməliyik, indeksləri də almaq üçün müvəqqəti bir dəyişən yaradıb hər dövrdə onu bir vahid artırmalıyıq:

```
sequence = [1,2,3,4,5]
i = 0      # müvəqqəti dəyişən
while i < len(sequence):
    print(sequence[i])
    i += 1     # sonsuz dövrə getməməkçün i+=1
```

Görürsünüz, adi bir ardıcıllığın elementlərini almaq üçün nə qədər əziyyət çəkməli oluruq. `while` əvəzinə, `for` dövründən istifadə etsəydik, bu məsələni iki sətrə həll edə bilərdik! `for` dövrünün sintaksisi:

```
for var_name in sequence:
    <code block>
```

Burda `var_name` – müvəqqəti dəyişən adı, `sequence` – iterə olunacaq ardıcıllıqdır. `<code block>` hissədə isə hər dövr icra olunacaq proqram kodudur. İndi isə, öncə `while` ilə yazdığımız kodu for dövrü ilə yazaq:

```
sequence = [1,2,3,4,5]
for i in sequence:
    print(i)
```

Qeyd etmək lazımdır ki, müvəqqəti dəyişənin adını istənilən dəyişən adı ilə adlandıra bilərsiniz, mütləqa ki `i` və ya `j` olmalı deyil. Məsələn, verilmiş ardıcıllıqda, verilmiş elementi axtaran proqramı `for` ilə yazsaq:

```
sequence = [1,2,3,4,5]
item = 3
for n in sequence:
    if n == item:
        print(sequence.index(n))
        break
```

İndi isə, məsələləri bir az da qəlizləşdirmək zamanıdır. Belə bir siyahı verilib:

```
authors = [{'name': 'Elkhan', 'surname': 'Zeynalli', 'age': 36},\
        {'name': 'Anar', 'surname': 'Rzayev', 'age': 85}, \
        {'name': 'Mikayil', 'surname': 'Mushviq', 'age': 29}]
```

Göründüyü kimi, `authors` lüğətlər siyahısıdır. Hər bir lüğət eyni adlı açarlardan ibarətdir. Deyək ki, bizdən yaşı(`age` açarı) 40-dan kiçik olan müəlliflərin adlarını çap etmək lazımdır. Bu zaman belə edə bilərik:

```
authors = [{'name': 'Elkhan', 'surname': 'Zeynalli', 'age': 36},\
        {'name': 'Anar', 'surname': 'Rzayev', 'age': 85}, \
        {'name': 'Mikayil', 'surname': 'Mushviq', 'age': 29}]
# yaşı 40-dan az olanları tap
author_list = []
for author in authors:       # author bir lüğətdir
```

```
    if author['age'] < 40:
        author_list.append(author['name'])
print(author_list)
```

```
['Elkhan', 'Mikayil']
```

Daha bir məsələ: verilmiş cümlədəki sözlərin sayını tapmalı. Fərz edək ki sözlər bir-birindən boşluq ilə ayrılmışdır. Belə olan hada, bir sətr olan cümləni split metodu ilə boşluqlar olan hissələrdən parçalayıb, `for` dövrü ilə sözləri iterə edək. Hər bir sözü boş lüğətə əlavə edək. Lüğətdə hər bir açar – bir söz, dəyəri isə sözün cümlədə neçə dəfə rast gəlindiyini göstərən ədəd olacaq. Əgər söz lüğətdə yoxdursa(o sözə bərabər olan açar yoxdursa) əlavə edib, dəyərini 1 kimi təyin edək, yox əgər sözə bərabər açar artıq varsa(deməli bundan öncə artıq əlavə etmişik) dəyərini bir vahid artıraq. Yəni, söz hər dəfə qarşımıza çıxdıqda sözün lüğətdəki dəyərini +1 edəcəyik. Baxaq:

```
sentence = "Aşbaz aşbaz aş asmış asmışsa da az asmış aş asmış"
sentence = sentence.lower()    # hərfləri kiçik et
words = sentence.split(' ')    # cümləni sözlərə parçala
word_count = {}            # boş lüğət
for word in words:        # sözlər siyahısını iterə edək
    if not word in word_count:
        word_count[word] = 1    # hələ əlavə edilməyibsə əla
    else:         # əks halda dəyərini al və
        word_count[word] = word_count[word] + 1   # 1 vahid artır

print(word_count)
```

```
{'aşbaz': 2, 'aş': 2, 'asmış': 3, 'asmışsa': 1, 'da': 1, 'az': 1}
```

`range` funksiyası. Bu funksiya verilmiş aralıqda ədədlərdən ibarət ardıcıllıq yaratmaq(və ya generasiya etmək) üçün istifadə olunur. Üç parametr alır:

```
range(start, stop, step)
```

Yəqin ki parametrlərin adları artıq sizə tanış gəldi, eynilə dilimləmə əməliyyatındakı kimi. Bir məqam var ki, `range` funksiyası nəticə olaraq siyahı və ya yazı yox, *generator* mənşəli xüsusi range obyekti qaytarır. Generatorlar barədə funksiyalarla tanış olduqdan sonra danışacam, hələki bilməli olduğumuz əsas məsələ - range geriyə xüsusi "əndrabadi" obyekt qaytarır. Bu əndrabadi obyekti çap edib, gözə yatan bir şey görmək istəyiriksə, bu obyekti `list` və ya `tuple` funksiyasının köməyi ilə uyğun olaraq siyahı bə ya yazıya çevirməliyik:

```
>>> range(1, 11)
range(1, 11)    # eləcə 'range(1, 11)' kimi çap olundu
```

```
>>> list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> tuple(range(1, 11))
(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

`range` funksiyası dövrlərlə tez-tez istifadə olunur. Çap olunarkən anlaşılmaz olmağına rəğmən, dövrlərlə istifadə edərkən range obyektini siyahıya filan çevirməyə ehtiyac yoxdur. Məsələn, bu yolla 1-5 aralığındakı ədədlərin kvadratlarını çap edə bilərəm, fikir verin, `range` funksiyasını `list` və ya `tuple` içərisində yazmağa ehtiyac yoxdur, səbəbini – generatorlarla tanış olduqdan sonra biləcəksiniz:

```
>>> for i in range(1, 6):
        print(i ** 2)
```

```
1
4
9
16
25
```

İç-içə `for` . Dövrləri iç-içə yazmaq mümkündür, dərinliyə heç bir məhdudiyyət yoxdur, proqramın bitməsini gözləyəcək vaxtınız varsa, lap 100 dövrü iç-içə yaza bilərsiniz, amma yazmasanız yaxşıdır... Növbəti məsələ kimi, matris təyin edib onun bütün elementlərini və koordinatları verilmiş elementini çap edən proqram kodu yazaq. Yadınıza salım ki, matris bizim üçün sadəcə ədədlərdən ibarət dördbucaqlı (sətr və sütunlar) formada yazılmış ədədlər toplusudur. Məsələn aşağıda `M` matrisi verilib.

$$M = \begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{matrix}$$

Siyahılardan artıq bildiyiniz kimi, bu matrisi siyahılar sıyahısı kimi yaza bilərik, yəni bir `M` siyahısı şəklində, hansının ki hər bir elementi matrisin bir sətri olacaq. Yəni belə: `M = [[1,2,3], [4,5,6], [7,8,9]]` . İndi isə matrisin bütün elementlərini çap edəcək kod yazaq:

```
M = [[1,2,3], [4,5,6], [7,8,9]]
for setr in range(len(M)):
    for sutun in range(len(M[setr])):
        print(M[setr][sutun], end='\t')
    print()
```

```
1    2    3
4    5    6
```

```
7    8    9
```

Son olaraq, onu qeyd etmək lazımdır ki, `break`, `continue` və `else` həm `while`, həm də `for` dövrü üçün eyni cür işləyir. Hansını istifadə etmək isə, ancaq məsələdən asılıdır, əgər konkret ardıcıllığın elementlərini iterə etmək, incələmək lazımdırsa – `for` dövrü, dəqiq dövr sayı bəlli deyilsə, sadəcə müəyyən şərt daxilində dövr işləməlidirsə - `while` dövrü seçilməlidir. Ancaq, yenə də - hər şey məsələdən asılıdır.

4.4. Siyahı və lüğət generatorları

Siahı və lüğət generatorları, bir sətrlik ifadə şəklində siyahı və ya lüğət yaratmağa imkan verir. Adətən bir sadə for dövrü ilə həll edilə bilən məsələləri bir sətrlik generaor ifadə şəklində yazmaq olar. Əvvəlcə, fərz edək ki `nums=[1,2,3,4,5]` siyahısı verilib və bu siyahının elementlərinin kvadratlarından ibarət yeni bir siyahı yaratmalıyıq. Belə olan təqdirdə, çox düşünmədən belə yazmaq olardı:

```
nums = [1,2,3,4,5]
sqrs = []
for i in nums:
    sqrs.append(i ** 2)
print(sqrs)    # [1, 4, 9, 16, 25]
```

Mahiyyət etibarilə, biz mövcud siyahı(`nums` siyahısı) əsasında yeni bir siyahı(`sqrs`) yaradırıq. Bu cür hallarda *siyahı generatorlarından*(ing. list comprehentions) istifadə etmək daha məqsədəuyğundur. Sadə siyahı generatorlarının sintaksisi belədir:

```
[expression for var_name in sequence]
```

Burada: `expression` – ixtiyari ifadə, `var_name` – müvəqqəti dəyişən adı, `sequence` – ardıcıllıqdır. Yazılışı belə şərh etmək olar: `sequence` ardıcıllığının növbə ilə elementlərini bir bir götür, onu `var_name` kimi adlandır və `expression` ifadəsini icra et. Siyahı generatoru yeni siyahı yaradır və `expression` ifadəsinin nəticəsi hər dəfə yeni bir element kimi bu siyahıya əlavə edilir. Bilirəm ki, ilk dəfə çox anlaşılmaz və mənasız səslənir, ilklərdə mənimsənilməsi çətin olan mövzulardandır və mövzunu tam qavramaq üçün müəyyən müddət tələb olunacaq, odur ki panikaya gərək yoxdur. Gəlin, ilk siyahı generatorumuzu yazaq və ilk donor kimi, az öncə yazdığımız ədədlərin kvadrata yüksəldilməsi alqoritmini seçək. Yuxarıdakı sintaktik yazılışa görə mənə 3 şeyi təyin etmək lazımdır: ardıcıllıq(`sequence`) – nəyin elementlərini iterə edəcəm; dəyişən adı(`var_name`) – hər iterasiyada elementin adı, bu lap `for` dövründəki müvəqqəti dəyişən kimidir; ifadə(`expression`) – element üzərində icra edəcəyimiz əməliyyatı ifadə şəklində bura yazmalıyıq. Bizim məsələmizdə ifadə - elementin kvadrata yüksəldilməsidir, yəni müvəqqəti dəyişən – `i` olsa, ifadəmiz `i**2` (və ya `pow(i, 2)` ,fərqi yoxdur) olmalıdır:

```
nums = [1,2,3,4,5]
sqrs = [i**2 for i in nums]
print(sqrs)    # [1, 4, 9, 16, 25]
```

Daha bir nümunə, verilmiş adlar siyahısındakı elementlərin birinci hərfi böyük yazılışlı, yeni siyahı yaradan proqram:

```
names = ['anar', 'ayan', 'arzu']
capitalised_names = [name.capitalize() for name in names]
print(capitalised_names)     # ['Anar', 'Ayan', 'Arzu']
```

İndi isə, bir addım da irəli ataq. Siyahı generatorları içərisində sadə şərti budaqlanmadan istifadə etmək olar. Bu zaman siyahı generatorunun sintaksisi bu cür olur:

```
[if_expression if condition else else_expression for var_name in sequence]
```

Burada `condition` – `if` operatorunun şərti, `if_expression` – bu şərt ödənərsə yerinə yetirilməli olan ifadə, `else_expression` – şərt ödənməyərsə yerinə yetirilməli olan ifadədir. İfadəni həm də bu cür yadda saxlamaq olar:

```
    [if_expression if condition else else_expression for var_name in sequence]
   [bunu_et əgər əgər bu_doğrudursa əks_halda bunu_et for var_name in sequnece]
```

İndi isə, bütün bunlardan istifadə edərək, ədədlər ardıcıllığının cüt elementlərinin kvadratlarından, tək elementlərinin isə kublarından ibarət olan yeni bir siyahı yaradan kod yaza bilərik:

```
nums = range(2, 12)
pow_nums = [i**2 if i%2==0 else i**3 for i in nums]
print(pow_nums)
# [4, 27, 16, 125, 36, 343, 64, 729, 100, 1331]
```

Bəs əgər else şərti bizə lazım deyilsə? Əgər yalnız şərti ödəyən elementlər əlavə olunmalıdırsa necə etməliyik ? Bu zaman natamam şərti budaqlanmadan istifadə etmək olar. Bu zaman else operatoru və onun şərtini yazmırıq, üstəlik operatorların da yeri bir az dəyişir və ifadə bu cür sintaksis alır:

```
[expression for var_name in sequence if condition]
```

Bu cür generatorda ancaq `condition` şərtini ödəyən elementlər siyahıya əlavə ediləcək:

```
nums = range(2, 12)
pow_nums = [i**2 for i in nums if i%2==0]
print(pow_nums)    # [4, 16, 36, 64, 100]
```

Lüğət generatorları. Lüğət generatorları da siyahı generatorları kimidirlər, ancaq adından da göründüyü kimi, siyahı deyil – lüğət yaradır. Üstəlik lüğətlər siyahılardan fərqləndiyi üçün yazılışlarında da fərqlər görəcəksiniz.

```
{key: value for var_name in sequence}
```

Gördüyünüz kimi, fərq – siyahı generatorlarında ifadə(`expression`) yazılan hissədə ifadə deyil açar: dəyər cütünün yazılması və fiqurlu mötərizələrdən istifadə edilməsidir. `key: value` – açar: dəyər cütü yeni yaradılacaq lüğətə əlavə ediləcək. Bir neçə nümunə üzərində baxaq, aşağıdakı proqram ədədlər siyahısından - `ədəd: onun_kvadratı` lüğətini yaradacaq:

```
nums = [1,2,3,4,5]
sqrs = {i: i**2 for i in nums} # i-açar, i**2-dəyər
print(sqrs)     # {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

`ədəd: onun_rəqəmlərinin_sayı` lüğətini yaradan proqram kodu:

```
nums = [1,21,300,4,5111]
sqrs = {i: len(str(i)) for i in nums} # i-açar, i**2-dəyər
print(sqrs)    # {1: 1, 21: 2, 300: 3, 4: 1, 5111: 4
nums = [1,21,300,4,5111]
sqrs = {i: len(str(i)) for i in nums} # i-açar, i**2-dəyər
print(sqrs)     # {1: 1, 21: 2, 300: 3, 4: 1, 5111: 4
```

Siyahı generatorlarında olduğu kimi, burada da şərti budaqlanmadan istifadə etmək olar. Deyək ki, bir ad-email lüğətimiz var, email-ləri olmayan istifadəçilərə yeni email yaratmaq istəyirik:

```
users = {'anar': 'anar@fakemail.com', 'faiq': 'faiq12@fakemail.com', 'elxan':
''}
generated = {user: f"{users[user]}@fakemail" for user in users if not
bool(users[user])}
users.update(generated)
print(generated)    # {'elxan': '@fakemail'}
print(users)
{'anar': 'anar@fakemail.com', 'faiq': 'faiq12@fakemail.com', 'elxan':
'@fakemail'}
```

Lüğət generatorları. Lüğət generatorları da siyahı generatorları kimidirlər, ancaq adından da göründüyü kimi, siyahı deyil – lüğət yaradır. Üstəlik lüğətlər siyahılardan fərqləndiyi üçün yazılışlarında da fərqlər görəcəksiniz.

```
{key: value for var_name in sequence}
```

Gördüyünüz kimi, fərq – siyahı generatorlarında ifadə(`expression`) yazılan hissədə ifadə deyil `açar: dəyər` cütünün yazılması və fiqurlu mötərizələrdən istifadə edilməsidir. `key: value` – açar: dəyər cütü yeni yaradılacaq lüğətə əlavə ediləcək. Bir neçə nümunə üzərində baxaq, aşağıdakı proqram ədədlər siyahısından - `ədəd: onun_kvadratı` lüğətini yaradacaq:

```
nums = [1,2,3,4,5]
sqrs = {i: i**2 for i in nums} # i-açar, i**2-dəyər
print(sqrs)    # {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

`ədəd: onun_rəqəmlərinin_sayı` lüğətini yaradan proqram kodu:

```
nums = [1,21,300,4,5111]
sqrs = {i: len(str(i)) for i in nums} # i-açar, i**2-dəyər
print(sqrs)                # {1: 1, 21: 2, 300: 3, 4: 1, 5111: 4
nums = [1,21,300,4,5111]
```

```
sqrs = {i: len(str(i)) for i in nums} # i-açar, i**2-dəyər
print(sqrs)          # {1: 1, 21: 2, 300: 3, 4: 1, 5111: 4
```

Siyahı generatorlarında olduğu kimi, burada da şərti budaqlanmadan istifadə etmək olar. Deyək ki, bir ad-email lüğətimiz var, email-ləri olmayan istifadəçilərə yeni email yaratmaq istəyirik:

```
users = {'anar': 'anar@fakemail.com', 'faiq': 'faiq12@fakemail.com', 'elxan':
''}
generated = {user: f"{users[user]}@fakemail" for user in users if not
bool(users[user])}
users.update(generated)
print(generated)    # {'elxan': '@fakemail'}
print(users)
# {'anar': 'anar@fakemail.com',
#  'faiq': 'faiq12@fakemail.com',
#  'elxan': '@fakemail'}
```

4.5. Ardıcıllıqların açılması və çoxdəyişənli generatorlar

Yazılar haqqında danışanda sizinlə ardıcıllıqların açılması barədə danışmışdıq, python mənimsətmə zamanı solda birdən çox dəyişən gördükdə, sağdakı obyekti "açmağa" çalışır:

```
>>> a, b = [1,2]   # a=1; b=2
```

Bu onun sayəsində baş verir, python sağdakı ardıcıllığı açır və oradakı elementləri mövqelərinə görə sağdakı dəyişənlərə mənimsədir. Bu "açılma" prosesi ingilis dilində unpacking(anpəkinq) adlanır və qablaşdırılmış bir şeyin geriyə açılması anlamını verir. Yaxşı, bəs ardıcıllıqdan çıxacaq element sayı ilə, sağdakı dəyişənlərin sayı eyni olmazsa nə baş verər:

```
>>> a, b = (1,2,3)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: too many values to unpack (expected 2)
```

Solda dəyişən sayı iki, sağdakı ardıcıllıq içərisində isə üç element var. Ona görə də ardıcıllığı açdıqda `a->1, b->2, ?->3` alınır, yəni 3 elementini mənimsətməyə heç bir dəyişən yoxdu, 3 "qalır yerdə". Bundan başqa, `*` işarəsindən istifadə etməklə, açılmanın "tərzini" təyin edə bilərik. Belə hallarda hansı dəyişənin qarşısına `*` yazılsa – "yerdə qalan bütün elementləri buna mənimsət" deməkdir. Çoxlu sayda elementi bir dəyişənə necə mənimsətmək olar – texniki olaraq heç necə, amma o elementlərin hamısını bir yazıya əlavə edib mənimsətmək olar. Əgər `a, *b = (1,2,3)` kimi yazsam, `a` dəyişəninə `1`, yerdə qalan `(2, 3)` elementləri isə `b` dəyişəninə mənimsədiləcək. Ulduzlu dəyişən sonda olmaya da bilər. Məsələn:

```
a, b, *c -> "birincini a-ya, ikincini b-yə, qalanlarını isə c-yə mənimsət";
*a, b, c -> "sağdan birincini c-yə, sağdan ikincini b-yə, qalanlarını isə a-
ya";
a, *b, c -> "soldan birincini a-ya, sağdan birincini c-yə, qalanları isə b-yə"
```

`zip` funksiyası. Mənzərəni tamamlamaq üçün `zip` funksiyası ilə tanış olaq. Funksiya parametr kimi bir və ya daha çox ardıcıllıq alır və onların indekslərinə görə uyğun elementlərini qaytarır. Məsələn, əvvəlcə bütün ardıcıllıqların 0 indeksli elementləri, daha sonra 1-ci və beləcə ən qısa ardıcıllığın son elementinə qədər. Baxaq:

```
>>> zip(['a', 'b', 'c'], ['x', 'y', 'z'], [1, 2, 3])
<zip object at 0x000001C03B29E840>
```

Gördüyünüz kimi, `zip` geriyə "zip obyekti" qaytarır. Bu obyekt bir növ generatordu və onu rahatlıqla iterə etmək olar:

```
>>> z = zip(['a', 'b', 'c'], ['x', 'y', 'z'], [1, 2, 3])
>>> for i in z:
...     print(i)
('a', 'x', 1)
('b', 'y', 2)
('c', 'z', 3)
>>>
```

Bütün bunları üst-üstə qoyub, siyahı və ya lüğət generatorlarını daha da funksional edə bilərik. Aşağıdakı nümunədə, email ünvanlar siyahısından yalnız 'email' dəyəri boş olmayan elementləri seçirik.

```
>>> accounts = {'a': 'aa@mail.com', 'b': '', 'c': 'ccc@gm.com'}
>>> accounts.items()
dict_items([('a', 'aa@mail.com'), ('b', ''), ('c', 'ccc@gm.com')])
>>> not_nulls = {k: v for k, v in accounts.items() if bool(v)}
>>> not_nulls
{'a': 'aa@mail.com', 'c': 'ccc@gm.com'}
```

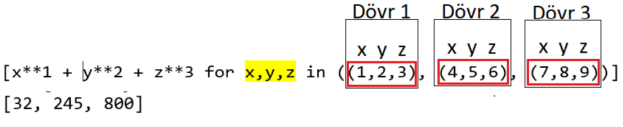
Bu yerdə bir neçə şeyi yada salmaq lazımdır:

- lüğətin `items` metodu (açar, dəyər) cütlərindən ibarət olan yazılar yazısı qaytarır.
- `bool` funksiyası boş olan hər şey üçün `False`, əks halda `True` qaytarırı, boş sətrlər üçün `False` alırıq. Göründüyü kimi, `accounts.items()` bizə `[('a', 'aa@mail.com'), ('b', ''), ('c', 'ccc@gm.com')]` kimi bir siyahı qaytarır. Bu siyahının hər bir elementi iki elementdən ibarət yazıdır. Yəni əgər bu siyahını açsaq:

```
>>> a, b, c = [('a', 'aa@mail.com'), ('b', ''), ('c', 'ccc@gm.com')]
>>> a
('a', 'aa@mail.com')
>>> b
('b', '')
>>> c
('c', 'ccc@gm.com')
```

Əgər, generator ifadədə(siyahı və ya lüğət generatoru) birdən çox müvəqqəti dəyişən varsa, ardıcıllığın hər bir elementi bu sayda elementlər şəklində açılmağa çalışır. Məsələn:

```
>>> [x**1 + y**2 + z**3 for x,y,z in ((1,2,3), (4,5,6), (7,8,9))]
[32, 245, 800]
```



Yuxarıdakı ardıcıllıq 3 dənə 3 elementli yazıdan ibarət bir yazıdır. Müvəqqəti dəyişənlərin sayı üç(`x` , `y` və `z`) olduğu üçün python bu atdıcıllığın hər bir elementini üçlük şəklində açmağa çalışır. Əgər dəyişənlərin sayı və ardıcıllıq açılan zaman alınan dəyərlərin sayı fərqli olarsa:

```
>>> [x**1 + y**2 + z**3 for x,y,z in ((1,2), (4,5,6), (7,8))]
Traceback (most recent call last):
ValueError: not enough values to unpack (expected 3, got 2)
```

Az öncə haqqında danışdığımız zip funksiyasını da yada salaq. Fərz edək ki adlar və soyadlar siyahıları var. Bu siyahılardan istifadə edərək ad: email lüğətini yaratmaq tələb olunur. Sadə lüğət generatoru və `zip` funksiyası vasitəsilə belə yazmaq olar:

```
names = ['Ali', 'Anar', 'Hikmet']
surnames = ['Mammadli', 'Rzayev', 'Nesibov']
emails = {name: f"{name}.{surname}@fakemail.com".lower() for name, surname in
zip(names, surnames)}
print(emails)
# {'Ali': 'ali.mammadli@fakemail.com', 'Anar': 'anar.rzayev@fakemail.com',
# 'Hikmet': 'hikmet.nesibov@fakemail.com'}
```

İlk iki sətrdə siyahılar təyin edilir. Bütöv lüğət generatoru ifadəmiz:

```
{name: f"{name}.{surname}@fakemail.com".lower() for name, surname in zip(names,
surnames)}
```

İfadəmiz 3 hissədən ibarətdir: ifadə hissəsi(`name: f"{name}.{surname}@fakemail.com".lower()`), müvəqqəti adlar hissəsi(`name, surname`) və ardıcıllıq hissəsi(`zip(names, surnames)`). İlk öncə ardıcıllıq hissəsinə nəzər yetirək: zip(names, surnames) bizə hər iterasiyada bir ad və bir soyad qaytaracaq, yəni qaytarılan veriləni cütlük kimi açmaq olar. Hər dövrdə qaytarılan verilən (ad, soyad) cütlüyü olduğu üçün iki ədəd müvəqqəti dəyişəndən name və surname istifadə edirik. İfadə hissəsi isə bu müvəqqəti dəyişənlər əsasında sətr formatlaşdırır: `name: f"{name}.{surname}@fakemail.com".lower()` Lap əvvəldə də dediyim kimi, generator ifadələr ilk tanışlıqda çətin və anlaşılmaz görünür, ancaq vaxt keçdikcə, tətbiq ediməyə çalışdıqca, hər şey öz yerinə oturur. Odur ki, əgər yuxarıdakı nümunələri ilk dəfədən bütövlükdə başa düşə bilməmisinizsə, sadəcə bəsit nümunələrlə oynamağa çalışın və yenidən cəhd edin.

Tapşırıqlar

1. Tələbələr siyahısı velib: `["Rəşad", "Əziz", "Ülvi", "Hikmət"]` . Hər bir tələbənin adını çap edin.

```
Rəşad
...
Hikmət
```

2. Bu dəfə hər bir tələbənin adını və sait və ya samitlə başlamasını çap edin.

```
Rəşad - samit
Əziz - sait
...
Hikmət - samit
```

3. Siyahı generatorlarından istifadə edərək, ancaq samitlə başlayan tələbələrdən ibarət olan `students_consonants` siyahısını yaradın.

```
# kodunuz
# students_consonants = ["Rəşad", "Hikmət"] olmalıdır
```

4. Siyahı generatorlarından istifadə edərək, tələbələr üçün email ünvanları siyahısı yaradın. Email domeni qismində `"@school.edu"` istifadə edin. Adları standartlaşdırmaq üçün adlardakı hərfləri bu cür əvəzləyin: `ə-e, ü-u, ö-o, ş-sh, ç-ch, ğ-gh, ı-i`

```
# sonda nəticə:
["reshad@school.edu", "eziz@school.edu", "ulvi@school.edu",
"hikmet@school.edu"]
```

5. Tələbələrin alınmış adlar və email ünvanlarından istifadə etməklə, tələbənin ad və email ünvanlarını saxlayacaq lüğətlər siyahısını hazırlayın. Siyahı generatorundan və `zip` funksiyasından istifadə edə bilərsiniz.

```
# sonda nəticə
[
    {"name": "Rəşad", "email": "reshad@school.edu"},
    {"name": "Əziz", "email": "eziz@school.edu"},
    {"name": "Ülvi", "email": "ulvi@school.edu"},
    {"name": "Hikmət", "email": "hikmet@school.edu"},
]
```

6. İndi isə, tələbələrin adlarından, email ünvanlarından və id nömrələrindən istifadə etməklə tələbələr lüğətini yaradın. Tələbənin id nömrəsi - adlar siyahısındakı indeksi olsun. Sonda lüğətin açarları id nömrələri, dəyərləri isə - ad və email üçvanlarını saxlayan lüğətlər olacaq:

```
    # sonda nəticə
    {
        "0": {"name": "Rəşad", "email": "reshad@school.edu"},
        "1": {"name": "Əziz", "email": "eziz@school.edu"},
        "2": {"name": "Ülvi", "email": "ulvi@school.edu"},
        "3": {"name": "Hikmət", "email": "hikmet@school.edu"},
    }
```

Link: https://github.com/AzizNadirov/python-road-book/blob/master/code/tasks/loops_4.ipynb

5. Funksiyalar

5.0. İndiyə qədər funksiyalar haqqında bildiklərimiz.

İndiyə qədər çoxlu sayda funksiyalardan istidadə etmişik, buraya: `print`, `input`, `pow`, `divmod`, `range`, `zip` və sairə. Gəlin, funksiyalar haqqında bildiklərimizi yerbəyer edək:

- **Funksiyalar çağrılır**: funksiyadan istifadə etmək, işə salmaq üçün – onu çağırmaq lazımdır. Hər hansı obyekti çağırmaq – onun qarşısında adi yumru mötərizələr açmaq deməkdir. Belə ki, pow özü-özlüyündə funksiyadır, mahiyyət etibarilə "funksiya tipli" obyektdir. `pow()` isə bu funksiya obyektini işə salır.

```
>>> pow
<built-in function pow>
>>> pow(2,2)
4
```

- **Bəzən funksiyanın işləməsi üçün müəyyən verilənlər – arqumentlər lazım gəlir**: yuxarıda göstərilən `pow` funksiyasının işləməsi üçün ən azı iki arqument(əsas və qüvvət) lazımdır, ancaq üçüncü (`mod`) da iştirak edə bilər. Bu o deməkdir ki, bəzən arqumentləri "arada buraxmaq", yazmamaq da olar.

```
>>> pow(5,2)
25
>>> pow(5,2,3)
1
```

- **Funksiya geriyə verilən qaytara bilir**: funksiyanı lazım olan arqumentlərlə çağırdıqdan sonra, funksiya geriyə nəticə qaytara bilər. Geriyə qaytarılan nəticəni dəyişənə mənimsətmək, hesablamada istifadə etmək və ya çap etmək olar:

```
>>> iyirmibesh = pow(5,2)
>>> doqquz = pow(3,2)
>>> iyirmibesh + doqquz
34
```

Bunlar, sadəcə indiyə qədər funksiyalardan çıxarda bildiyimiz məqamlardı. Bütün bu funksiyalar, Pythonın öz funksiyaları, daxilinə qurulmuş – qurma funksiyaları idi. İrəlidə isə biz özümüz öz funksiyalarımızı yaratmağa başlayacağıq və yuxarıda təyin etdiyimiz o xüsusiyyətlər bizə funksiyalar başa düşməyə kömək edəcək.

5.1. Funksiyaların mahiyyəti

Nə isə kodlaşdırmağa başlamazdan əvvəl, sizə funksiyaların təyinatından danışmaq istəyyirəm. Nə üçün onlar bu qədər vacibdirlər və onlarsız hansı çatışmamazlıqlar yaşayardıq. Funksiyalar – ixtiyari həcmdə proqram kodunu bir ad altında yazmağa, birləşdirməyə imkan verir. Həmin proqram kodundan istifadə etmək lazım gəldikdə, sadəcə olaraq, həmin adı çağırmaq lazım gəlir. Təsəvvür edin ki, pow funksiyasının arxasında 50-60 sətrlik kod dayanır. Amma, siz sadəcə funksiyanı

çağıraraq, o 50-60 sətrlik kodun "gücündən" istifadə edirsiniz, üstəlik – ixtiyari sayda istifadə edə bilərsiniz. Başqa sözlə, funksiyalar vasitəsilə proqram kodunu təkrar istifadə etmək olduqca rahatdır. Mənasına varmadan belə bir proqram kodu yazıb işə salaq:

```
def add(n, m):
    res = n + m
    return res
```

Yuxarıda - iki ədəd alaraq onların cəmini qaytaran funksiya yazılıb. İndi, Python IDLE mühitində proqramı `F5` ilə işə salaraq, aşağıdakı kimi edə bilərik:

```
>>> add(1, 3)
4
>>> add(35, 5)
40
>>> add('hello_', 'world')
'hello_world'
```

Gördüyünüz kimi, `add` adlı funksiya bir dəfə təyin olunur(yaradılır) və üç dəfə istifadə olunur.

5.2. Funksiyaların sintaksisi

Funksiyanı təyin etmək üçün `def` - *define*(təyin etmək) açar sözündən istifadə edilir. Daha sonra funksiyanın adı, yumru mötərizələr və bu mötərizələr daxilində parametrlər sadalanır:

```
def func_name(param1, param2):
    <func_body>
```

Yuxarıdakı nümunədə `func_name` – funksiyanın adıdır, `<funk_name>` hissəsində isə, funksiyanın gövdəsi yerləşir. `param1` və `param2` – funksiyanın aldığı parametrlərdir. Funksiya çağırılarkən bu parametrlər hökmən ötürülməlidir və funksiya daxilində istifadə oluna bilərlər. İlk nümunə kimi, iki ədəd alıb, onların cəmini çap edəcək `add` adlı funksiya yaradaq. Funksiya iki ədəd cəmləməli olduğu üçün, bu iki ədədi funksiyaya parametr kimi ötürməliyik, gəlin onları `num1` və `num2` kimi adlandıraq:

```
def add(num1, num2):
    print(num1 + num2)
```

Yuxarıdakı proqram kodunu işə salsaq, heç nə baş verməyəcək. Çünki, öncə deyildiyi kimi, təyin olunmuş funksiyanı işə salmaq üçün, onu – çağırmaq lazımdır. Funksiyanı çağırmaq üçün, funksiyanın adının qarşısında yumru mötərizələr açıb, orada funksiyanın işləməsi üçün zəruri olan parametrləri sadalamalıyıq.

```
# funksiyanın təyin olunması
def add(num1, num2):
    print(num1 + num2)
```

```
# funksiyanın çağrılması
add(3,5)
```

Kod işə salındıqda `8` çap edəcək. Funksiyanı çağırarkən, ötürdüyümüz parametrlər mövqeyinə görə funksiyanın başlığındakı arqumentlərə mənimsədilir. Yəni, `num1=3` , `num2=5` olur.

Arqumentlər və Parametrlər. Bu iki söz çox vaxt yan-yana istifadə olunur və bəzən mənaları səhv salınır. *Parametr* – funksiyanın başlığında qeyd etdiyimiz addır, məsələn: `num1` , `num2` . *Arqument* isə - funksiyanı çağırarkən ötürdüyümüz real dəyərlərdir – məsələn: 3, 5. Funksiya çağrıldıqda, arqumentlər funksiyaya ötürülür və parametrlərə mənimsədilir. İndi isə, üç ədəd alıb, onların ədədi ortasını çap edəcək funksiya yazaq. Ədədlərin ədədi ortası – ədədlərin cəminin, ədədlərin sayına olan nisbətidir:

```
def average(a, b, c):
    s = a + b + c
    print(s / 3)

average(3, 3, 6)    # 4.0
```

5.3. Qaytarmaq ya çap etmək?

İndi isə, düşünək: bizim funksiyalar nəticə olaraq nə edir – nə isə çap edir. Çap etmək – sadəcə olaraq görmək üçündür. İstifadə etmək üçün isə çap etmək bəs etmir! Məsələn, qurma pow funksiyası qüvvət çap etmir, qüvvəti qaytarır. Bunun sayəsində biz funksiyanın nəticəsi olan ədədi istifadə edə bilirik. Məsələn, `pow` funksiyasının köməyi ilə, `23+1` ifadəsini hesablamaq lazımdır. Bunu belə yazmaq olar:

```
>>> pow(2, 3) + 1     # 9
```

`pow` funksiyası nəticəni qaytardığı üçün, həmin qaytarılan nəticəsini üzərinə vahid əlavə edə bildik. Başqa sözlə, çap olunan dəyəri istifadə edə bilmirik. Funksiyadan sonda nəticə almaq istəyiriksə, elə etməliyik ki, funksiya həmin dəyəri bizə qaytarsın. Funksiyadan geriyə dəyər qaytarmaq üçün `return` açar sözündən istifadə edilir. Funksiya icra olunarkən, `return` açar sözünün qarşısındakı dəyəri(ifadə yazılarsa – ifadənin nəticəsini) geriyə qaytarır və sonlanır. İndi isə, öncə yazdığımız `average` funksiyamıza yenidən baxaq. funksiya ədədlərin ədədi ortasını çap edirdi. Funksiyanı dəyişərək, ədədi ortanı qaytarmasını təmin edək:

```
def average(a, b, c):
    s = a + b + c
    return s / 3
average(3, 3, 6)    # heç nə çap etmir
```

Kodu işə saldıqda nəticə görünmür, çünki `average(3, 3, 6)` bizə `4` qaytarsa da, biz bu nəticəni istifadə etmədiyimiz üçün görmürük. Görmək üçün `print(average(3, 3, 6))` yazmaq olar. İndi isə, həmin ədədlərin ədədi ortalarının 10 mislini tapmaq tələb olunnur. Belə olan təqdirdə, funksiyamız vasitəsilə ədədi ortanı tapıb, nəticəni 10-a vura bilərik – çünki, funksiyamız geriyə ədədi orta ədədini qaytarır:

```
def average(a, b, c):
    s = a + b + c
    return s / 3
print(average(3, 3, 6) * 10)    # 40
```

Əlimiz soyumamış nümunələr yazmağa davam edək: ədədi orta funksiyamızı bir az təkmilnləşdirək – parametr kimi ayrı-ayrı ədədlər deyil, ədədlərdən ibarət tək sıra alaq, elementlərin sayını tapmaq üçün `len` , cəmini tapmaq üçün isə `sum` funksiyasından istifadə edək:

```
def average(nums):
    s = sum(nums)
    ln = len(nums)
    return s / ln

average([1,2,3,4,5])    # 3.0
```

Əslində, bu kodu daha qısa da yazmaq olar:

```
def average(nums):
    return sum(nums) / len(nums)
average([1,2,3,4,5])   # 3.0
```

Hər hansı alqoritmi ərsəyə gətirmək üçün funksiya əla vasitədir. Çünki, burada giriş(parametrlər) və çıxış(`return` ilə qaytarılan) verilənləri aydındır. Sadəcə götürüb, lazım olanları verib – cavabını alırsan! Gəlin öncə yazmış olduğumuz bəzi nümunələri "funksiyalaşdıraq"!

- Ötürülmüş sırada, ötürülmüş elementin indeksini axtaran funksiya:

```
def find(array, item):
    i = 0
    while i < len(array):
        if array[i] == item:  # element tapıldı
            return i          # qaytar
        else:
            i += 1   # tapılmasa, i = i+1, əks halda sonsuz dövü yaranacaq
    else:
    return -1

array = [2, 7, 8, 9, 11]
print(find(array, 2))    # 0
print(find(array, 8))    # 2
```

- Ötürülmüş sətrlər siyahısındakı bütün sətrlərin ilk hərfini böyük edib qaytaran funksiya:

```
def capitalise_all(words):
    capitalised_names = [word.capitalize() for word in words]
    return capitalised_names

names = capitalise_all(['anar', 'ayan', 'arzu'])
```

```
cities = capitalise_all(['baku', 'ganja', 'barda'])
print(names)   # ['Anar', 'Ayan', 'Arzu']
print(cities)  # ['Baku', 'Ganja', 'Barda']
```

- Cümlədəki sözləri sayıb, lüğət şəklində qaytaran funksiya:

```
sentence_1 = "Aşbaz aşbaz aş asmış asmışsa da az asmış aş asmış"
sentence_2 = "bir bir gəlin bir bir gedin"

def count_words(sentence):
    sentence = sentence.lower()  # hərfləri kiçik et
    words = sentence.split(' ')  # cümləni sözlərə parçala
    word_count = {}   # boş lüğət
    for word in words:    # sözlər siyahısını iterə edək
        if not word in word_count:
            word_count[word] = 1   # hələ əlavə edilməyibsə əlavə et
                                   # dəyərini 1 kimi təyin et
        else:                # əks halda dəyərini al və
        word_count[word] = word_count[word] + 1    # 1 vahid artır
    return word_count

print(count_words(sentence_1))
# {'aşbaz': 2, 'aş': 2, 'asmış': 3, 'asmışsa': 1, 'da': 1, 'az': 1}
print(count_words(sentence_2))
# {'bir': 4, 'gəlin': 1, 'gedin': 1}
```

5.4. İç-içə - nested yazılış

Bir funksiya içərisində çox rahatlıqla başqa bir funksiya təyin etmək olar:

```
def salam_sagol(ad):
    # iç funksiya 1
    def salamla(ad):
        print(f"Salamlar olsun - {ad} ")
    # iç funksiya 2
    def sagollas(ad):
        print(f"Görüşənədək - {ad}")

    # salamlaş
    salamla(ad)
    # sağollaş
    sagollas(ad)

salam_sagol("əziz")
# Salamlar olsun - əziz
# Görüşənədək - əziz
```

`salam_sagol` içərisində iki ədəd funksiya təyin olunur: `salamla` və `sagollas`. Əsas `salam_sagol` funksiyası içərisində bu funksiyalar növbə ilə çağrılır və müvafiq ismarıclar çap olunur. İç-içə funksiyaların əsas üstünlüyünü *qapanmalar* və *dekoratorlar* barədə danışarkən görəcəksiniz. İndi isə nümunələrlə öyrənməyə davam edək. Sadə bir hesablayıcı funksiyası yazaq.

Funksiya üç ədəd arqument alacaq: `ədəd 1`, `ədəd 2` və `əməliyyat`. Əməliyyat - ədədlər arasında aparılmalı olan əməliyyatı təyin edəcək. Məsələn, `calc(5, 5, '+')` arqumentləri ilə çağırdıqda `10` almalıyıq.

```
def calc(num_1, num_2, operation):
    # mövcud əməliyyatlar
    operations = ('+', '-', '*', '/')
    # hesablayıcı funksiyalar
    def add(num_1, num_2):
        return num_1 + num_2
    def sub(num_1, num_2):
        return num_1 - num_2
    def mul(num_1, num_2):
        return num_1 * num_2
    def divide(num_1, num_2):
        return num_1 / num_2

    # əməliyyatın düzgünlüyünü yoxla
    if not operation in operations:
        print(f"Yanlış əməliyyat: '{operation}'. Düzgün əməliyyatlar: \
        {operations}")
        return None
    else:
        if operation == '+':
            res = add(num_1, num_2)
        elif operation == '-':
            res = sub(num_1, num_2)
        elif operation == '*':
            res = mul(num_1, num_2)
        elif operation == '/':
            res = divide(num_1, num_2)
    return res

print(calc(5, 5, '+'))    # 10
print(calc(5, 5, '-'))    # 0
print(calc(5, 5, '*'))    # 25
print(calc(5, 5, '/'))    # 1.0
```

Öncə deyildiyi kimi, pythonda hər şey obyektdir, funksiya da həmçinin:

```
>>> def foo():
    ... print("hi!")
>>> foo()
hi!
>>> foo
<function foo at 0x000001C98D670A40>
>>>
```

Biz bu obyektlə indiyə qədər digər tip verilənlərlə etdiyimiz əksər əməliyyatları edə bilərik. Məsələn, funksiyalar siyahısı yaratmaq, funksiya obyektini başqa bir funksiyaya arqument kimi ötürmək, dəyişənə mənimsətmək və sairə. Rahatlıqla dəyərləri funksiyalar olan lüğət yaratmaq olar. Budan istifadə edərək, öncəki calc funksiyasını bir az təkminləşdirə bilərik:

```
def calc(num_1, num_2, operation):
    # hesablayıcı funksiyalar
    def add(num_1, num_2):
        return num_1 + num_2

    def sub(num_1, num_2):
        return num_1 - num_2

    def mul(num_1, num_2):
        return num_1 * num_2

    def divide(num_1, num_2):
        return num_1 / num_2

    # mövcud əməliyyatlar: açar - əməliyyat, dəyər - əməliyyata cavabdeh
funksiya
    operations = {'+': add, '-': sub, '*': mul, '/': divide}
    # əməliyyatın düzgünlüyünü yoxla
    if not operation in operations.keys():
        print(f"Yanlış əməliyyat: '{operation}'. Düzgün əməliyyatlar:\
        {operations}")
        return None
    else:
        func = operations[operation]
        res = func(num_1, num_2)
        return res

print(calc(5, 5, '+'))   # 10
print(calc(5, 5, '-'))   # 0
print(calc(5, 5, '*'))   # 25
print(calc(5, 5, '/'))   # 1.0
```

5.5. Parametrlər və Arqumentlər

Parametrlər vs Arqumentlər.

Gəlin yada salaq: Funksiyanı təyin edərkən sadaladıqlarımız - funksiyanın *parametrləridir*, çağırdıqda funksiyaya ötürdüyümüz dəyərlər isə - *arqumentlər*. Məsələn, `foo` funksiyası təyin olunub:

```
def foo(a, b, c):    # parametrlər
    print(f"a={a}, b={b}, c={c}")

foo(1,2,3)     # arqumentlər
```

Yuxarıdakı `foo` funksiyasını üç ədəd parametr qəbul edir: `a`, `b` və `c`. Funksiyanı çağırarkən, ona dəyərlər ötürürük, bu dəyərlər müvafiq olaraq uyğun parametrlərə mənimsədilir, foo üçün a=1, b=2, c=3 olur.

Susmaya görə dəyərlər

Yadınızdadırsa, bizim sevimli `pow` funksiyamızın iki cür "davranışı" vardı: iki ədəd arqumentlə çağırdıqda - onlardan birincisi əsas, digəri qüvvət olur və geriyə "əsas üstü qüvvət" qaytarırdı. Əlavə olaraq, `pow` üçüncü arqument də ala bilirdi - bu arqument ötürüldükdə, ilk iki arqumentlər vasitəsilə tapılan qüvvətin üçüncə ədəd ilə modu(bölmədən alınan qalıq) tapılır.

```
>>> pow(2, 4)   # 2 üstü 4 -> 16
>>> pow(2, 4, 5)  # 2 üstü 4 mod 5 -> 16 % 5 -> 1
```

Bizi maraqlandıran sual: necə olur ki funksiya həm iki, həm də üç ədəd arqument ilə işləyir? Əgər özümüz elə bir şey yazmağa çalışsaq:

```
def my_pow(base, power, mod):
    return (base ** power) % mod

print(my_pow(2, 4, 5))   # 1
print(2, 4)   # error!
# TypeError: my_pow() missing 1 required positional argument: 'mod'
```

Məsələ orasındadır ki, funksiyamızın çalışması üçün hər üç parametr tələb olunur. Python da daxil olmaqla, əksər proqramlaşdırma dillərində parametrlərə *susmaya görə dəyər* (default value) təyin etmək mümkündür. Parametrin susmaya görə dəyəri - həmin parametr ötürülmədiyi təqdirdə parametrin malik olduğu dəyərdir. Belə çıxır ki, əgər hər hansı parametrin susmaya görə dəyəri varsa, funksiyanı çağırarkən, həmin parametrə dəyər ötürməyə də bilərik. Parametrə susmaya görə dəyər təyin etmək üçün elə funksiyanın başlığındaca `<parametr_adı>=dəyər` kimi yazmaq lazımdır. Öncəki `my_pow` funksiyamızın mod parametrinə susmaya görə dəyər - `None` təyin edək.

```
def my_pow(base, power, mod=None):
    if mod is None:
        return base ** power
    else:
        return (base ** power) % mod

print(my_pow(2, 4, 5))   # 1
print(2, 4)              # 16
```

Yuxarıdakı nümunədə nəticəni hesablamazdan əvvəl `mod` parametrinin dəyərini yoxlayırıq: əgər `None` olarsa - deməli istifadəçi `mod` üçün heç nə ötürməyib (ya da elə `None` dəyərini ötürüb) və mode hesablamağa ehtiyac yoxdur, əks təqdirdə isə `mod` ilə ifadəmizi hesabıyırıq. Daha bir nümunə: verilmiş ad və soy adlı istifadəçi üçün email generasiya edən funksiya:

```
def generate_email(name, surname, separator='', domain='fake.com'):
    return f"{name}{separator}{surname}@{domain}"

print(generate_email("aziz", "nadirov")) # 'aziznadirov@fake.com'
print(generate_email("aziz", "nadirov", '_')) # 'aziz_nadirov@fake.com'
print(generate_email("aziz", "nadirov", '_', 'ml.com')) # 'aziz_nadirov@ml.com'
```

⚠ **Diqqət**

Funksiyanın təyini zamanı, susmaya görə dəyərləri olan parametrlər sonda qeyd olunur. Əks təqdirdə sintaktik xəta almış olacağıq. Susmaya görə dəyəri olmayan bir parametri, susmaya görə dəyəri olan parametrdən sonra yazsaq:

```
def generate_email(surname, separator='', domain='fake.com', name):
   return f"{name}{separator}{surname}@{domain}"

SyntaxError: non-default argument follows default argument
```

Nə üçün belə bir məhdudiyyət var - bu haqqda "Arqumentlərin funksiyaya ötürülmə üsulları" bəhsində baxacağıq.

Arqumentlərin funksiyaya ötürülmə üsulları

info

Bəzən hansısa proqram bloku yazılmalı olan yerdə sadəcə bir söz - `pass` görəcəksiniz. `pass` bir növ "heç nə", "bura boşdur" deməkdir. Əgər funksiyanın gövdəsinə `pass` yazılıbsa, deməli funksiya heç nə etmir, heç nə etməyən funksiya isə geriyə "heç nə", yəni `None` qaytarır. Əlavə olaraq, bəzən funksiyalarda `return` operatorundan sonra heç nə yazılmır, belə olan təqdirdə - geriyə `None` qaytarılır.

Fərz edək ki

```
def foo(a, b):
    pass
```

kimi təyin olunmuş funksiyamız var. Funksiyanı `a=1, b=2` qiymətlərini ötürmək üçün `foo(1, 2)` kimi ötürürdük. Bu cür arqumentlər *mövqeli arqumentlər* (positional arguments) adlanır. Niyə mövqeli, çünki arqumentləri ötürərkən parametrlərin ardıcıllığına - mövqelərinə görə ötürürük, birinci arqument - birinci parametrə, ikinci arqument-ikinci parametrə və sairə. İndi isə təsəvvür edək ki funksiyamız 15 ədəd parametr alır və belə çıxır ki funkasiyanı çağırarkən bütün bu parametrlərin ardıcıllığını yadda saxlamaq lazımdı... Çıxış yolu kimi - *adlı arqumentlərdən* (keyword argument) istifadə etmək olar. Arqumenti adlı etmək - onu parametrin adını yazaraq ötürmək deməkdir. `foo` funksiyasını adlı arqumentlərlə çağırsaq: `foo(a=1, b=2)` kimi çağırırıq. Arqumenti adlı şəkildə ötütdükdə ardıcıllıq önəm daşımır, çünki arqument bir başa parametrin adına uyğun mənimsədilir: `foo(b=2, a=1)` . Əlavə olaraq, adlı və mövqeli arqumentləri eyni çağırışda istifadə etmək olar, ancaq bir şərtlə: **əvvəlcə mövqeli arqumentlər, daha sonra adlı arqumentlər sadalanır**! Nəticədə `foo(1, b=2)` yazılışı da doğrudur, ancaq `foo(a=1, 2)` yazılışı doğru **deyil**!

```
def foo(a, b):
    pass

foo(a=1, 2)  # adlı arqumentdən sonra mövqeli olmaz!
# SyntaxError: positional argument follows keyword argument
```

Sadə bir `calc` funksiyası verilib. Funksiya `a` və `b` üçün `operator` əməliyyatını icra edir, susmaya görə `operator='+'` :

```
def calc(a, b, operator='+'):
    if operator == '+':
        return a + b
    elif operator == '-':
        return a - b
    elif operator == '*':
        return a * b
    elif operator == '/':
        return a / b
    else:
        return None

print(calc(2, 3))                   # 5
print(calc(2, 3, operator='*'))     # 6
```

Fərz edək ki, içərisində işçilər olan `employees` siyahısı var, siyahının hər bir elementi bir lüğətdir və hər bir lüğət - bir işçinin id, ad, soy ad və yaşından xüsusiyyətlərindən ibarətdir. `search_employee` funksiyası ötürülmüş xüsusiyyətlərə sahib işçiləri qaytarır. Axtarış üçün həm hər üç xüsusiyyətdən, həm də ancaq lazım olan birisindən istifadə etmək olar. `only_id` parametri `True` kimi ötürülərsə, geriyə tapılan işçilərin lüğətləri deyil, ancaq `id` -ləri qaytarılacaq.

```
employees = [
       {'id': 0, 'first_name': 'Aziz', 'last_name': 'Nadirov',
       'age': 25, 'salary': 1500},
       {'id': 1, 'first_name': 'Aziz', 'last_name': 'Nadirli',
       'age': 26, 'salary': 1500},
       {'id': 2,'first_name': 'Den', 'last_name': 'Bruk',
       'age': 30, 'salary': 2000},
       {'id': 3,'first_name': 'Leyla', 'last_name': 'Rahimova',
       'age': 26, 'salary': 1400},
   ]

def search_employee(employees, name=None, age=None, salary=None,
only_id=False):
   if name == age == salary == None:
       print("All parameters are None")
       return None

   found = []
   for employee in employees:
       if name and employee['first_name'] != name:
           continue
       if age and employee['age'] != age:
           continue
       if salary and employee['salary'] != salary:
           continue
       found.append(employee)

    # belə
```

```
    if only_id:
        return [employee['id'] for employee in found]
    else:
        return found

    # ya da qısaca belə:
    # return [employee['id'] for employee in found] if only_id else found


print(search_employee(employees, name="Aziz"))
print(search_employee(employees, name="Aziz", only_id=True))
print(search_employee(employees, name="Aziz", age=25))
```

```
[{'id': 0, 'first_name': 'Aziz', 'last_name': 'Nadirov', 'age': 25, 'salary':
1500},
 {'id': 1, 'first_name': 'Aziz', 'last_name': 'Nadirli', 'age': 26, 'salary':
1500}]

[0, 1]

[{'id': 0, 'first_name': 'Aziz', 'last_name': 'Nadirov', 'age': 25, 'salary':
1500}]
```

5.6. Adlar fəzaları və LEGB qaydası

Geriyə - funksiyanın təyinatına qayıtsaq, hər bir funksiya mümkün qədər müstəqil – çöldəki kodla ancaq arqumentlər vasitəsilə əlaqəli olmalıdır. Bu nöqteyi nəzərdən, funksiyaların adlar fəzaları çöldəki proqramdan təcrid olunur. *Adlar fəzası* (ing - namespace) - dəyişənin yaradılarkən, proqramın hansı hissəsindən əlçatan olacağını təyin edir. Hələ ki dörd fərqli adlar fəzası haqqında danışacağıq: `lokal`, `qapadan`, `qlobal` və `qurma` adlar fəzası.

Local adlar fəzası(ing local namespace) – müəyyən funksiyanın içərisində təyin edilmiş adların yazıldığı adlar fəzasıdır. Funksiyanın müstəqil bir altproqram olduğu üçün, funksiyanın daxilində təyin olunan adlar yalnız həmin funksiyanın içərisində görünür və həmin funksiyanın *lokal dəyişənləri* adlanır.

Qlobal adlar fəzası(ing global namespace) – hər hansı funksiya deyil, modul səviyyəsində təyin olunmuş adların yazıldığı adlar fəzasıdır. Qlobal adlar fəzasında təyin olunmuş adlar modul daxilində hər yerdə əlçatandır. Hələki, bu iki adlar fəzası ilə yaxından tanış olaq.

```
y = 11    # qlobal
def f():
    x = 10    # lokal
    print(f"x from f: {x}")
    print(f"y from f: {y}")
f()
```

Yuxarıdakı nümunədə `f` funksiyası daxilində iki ədəd dəyişən – `y` və `x` təyin edilmişdir. `y` dəyişəni heç bir funksiya daxilində deyil, modul səviyyəsində olduğu üçün qlobal dəyişəndir. `x` dəyişəni isə, `f` funksiyası daxilində təyin edildiyi üçün `f` funksiyasının lokal dəyişənidir.

Qapadan adlar fəzası(ing enclosing namespace) – öz daxilində `f2` funksiyasını saxlayan `f` funksiyası varsa, `f` funksiyasının adlar fəzası `f2` üçün *qapadan adlar* fəzası adlanır. Baxaq:

```
x = 1   # qlobal
def f():
    y = 2   # f-ə görə lokal, f2-ə görə qapadan
    def f2():
        z = 3   # lokal
        print(f"x={x}; y={y}; z={z}")   # x=1; y=2; z=3

    f2() # f2 çağrılır
f()
```

Funksiyanın özü üçün, öz içərisində təyin edilən adlar lokaldır. Əgər adlar çöl-valideyn funnksiyada təyin olunubsa, o zaman adlar qapadan adlar fəzasına yazılır. Buna görə də `f` içərisində təyin olunan `y` dəyişəni `f` üçün lokal(çünki, `f` -in özünün daxilində təyin olunub), `f` içərisində olan `f2` üçün isə qapadan sayılır. `f` funksiyası `f2` funksiyasını əhatə etdiyi – qapadığı üçün, adlar fəzası da bu cür adlanır.

Qurma adlar fəzası(ing builtin namespace) – Python dilinin özünə daxil olan, öncədən təyin olunmuş adlardır. Belə adlara hazır funksiyaları – `print`, `pow`, `input`, `range`, `map`, `filter` və sairə göstərmək olar. Qeyd edim ki, söz operatorlar – `for`, `while`, `in`, `and`, `or` və sairə bura daxil **deyil**, onlar açar sözlər hesab olunur.

Adların adlar fəzalarında axtarış ardıcıllığı – LEGB qaydası.

Öyrəndik ki Pythonda dörd tip adlar fəzası var – lokal, qapama, qlobal və qurma. Bəs biz hər hansı addan istifadə etməyə çalışarkən, Python tələb olunan adı necə axtarır, hansı adlar fəzalarını və hansı ardıcıllıqda yoxlayır? Hər şey adın istifadə olunduğu yerdən asılıdır:

- əgər ad lokal proqram blokunda – hər hansı funksiyanın içərisində istifadə olunmağa çalışırsa, o zaman axtarış əvvəlcə həmin funksiyanın içərisində, daha sonra çöl funksiyada(əgər varsa), daha sonra isə qlobal və sonda qurma adlar fəzasında aparılır. Yəni, bir növ içdən-çölə doğru! Əgər ad son adlar fəzası olan qurma adlar fəzasında da olmazsa, NameError atılır.
- əgər ad qapama adlar fəzasındadırsa(özündə başqa funksiya saxlayan bir funksiyanın içərisində), o zaman içəridən-çölə prinsipinə əsasən axtarış əvvəlcə funksiyanın öz içərisində, sonra isə çöldəki adlar fəzalarında(qlobal və qurma) aparılır.
- ümumiyyətlə, istəyir lap 100 ədəd bir-biriinin içərisində yazılmış funksiyamız olsun, axtarış həmişə içəridən(lokaldan), qurma adlar fəzasına qədər davam edəcək.
- ```
  ![[Pasted image 20231024214152.png|450]]
  ```

Təsəvvür edin ki, dörd ədəd `f1`, `f2`, `f3`, `f4` funksiyarı var və yuxarıdakı şəkildəki kimi – `f4` - `f3` -ün, `f3` – `f2` - nin, `f2` isə `f1` -in daxilində təyin olunur. Bu zaman `f4` daxilində dəyişəndən istifadə etməyə çalışsaq(məsələn, `print(x)` yazsaq), o zaman python dəyişəni əvvəlcə `f4` -ün özünün daxilində, daha sonra ox boyu `f3-> f2-> f1-> Qlobal-> Qurma` adlar fəzalarında
  ```

ataracaq. Və yaxud da, `f2` içərisində istifadə etməyə çalışsaq, `f2-> f1-> Qlobal-> Qurma` ardıcıllıqlı axtarış olacaq – içəridən çölə doğru. Baxaq:

```
x = 'global'
def f1():
    x = 'F1'
    print(f"variable x in f1: {x}")

    def f2():
        print(f"variable x in f2: {x}")
        def f3():
            x = 'F3'
            print(f"variable x in f3: {x}")
                def f4():
                    x = 'F4'
                    print(f"variable x in f4: {x}")
            f4()
        f3()
    f2()
f1()
print(f"variable x in global: {x}")
```

```
variable x in f1: F1
variable x in f2: F1
variable x in f3: F3
variable x in f4: F4
variable x in global: global
```

Yuxarıdakı listinqdən göründüyü kimi, `f2` içərisində `x` dəyişəninin qiyməti – `f1` -də təyin olunmuş qiymətdir, çünki `f2` -nin öz içərisində `x` təyin olunmayıb, içəridən-çölə prinsipinə görə axtarış `f2` içərisindən `f1` -ə keçir və orada `x='F1'` tapılır. Əgər bütün `f` funksiyalarındakı `x = ...` sətrlərini silsək, bütün `print` əmrləri "`variable x in global: global` " çap edəcək. Çünki, `x` adı yalnız qlobal adlar fəzasında mövcud olur və python içəridən-çölə doğru axtarışında `x` adını məhz qlobal adlar fəzasındakı `x='global'` qiymətini tapacaq. Bu cür içəridən-çölə doğru axtarış qaydası – **LEGB qayası** adlanır və axtarışın aparıldığı adlar fəzalarının ilk hərflərinin abbriviaturasıdır(**L**-local, **E**-enclosing, **G**-global, **B**-builtin).

LEGB qaydasından istifadə etməklə qurma adların təyinatlarının dəyişdirilməsi. Yuxarıda deyildiyi kimi, qurma adlar fəzasında qurma funksiyalar(`print` , `input` və s.) yerləşir. Əlavə olaraq LEGB qaydasına görə - ən son növbədə axtarılan adlar fəzası da məhz qurma adlar fəzasıdır. Bu o deməkdir ki, əgər biz məsələn, input adında dəyişən yaratsaq və ondan istifadə etməyə çalışsaq, python bizim təyin etdiyimiz qiyməti tapacaq:

```
input = "bad practice"
def foo():
    n = input("Type something: ")
```

```
foo()     # TypeError: 'str' object is not callable
```

Yuxarıda input adlı dəyişənə `bad practice` sətrini mənimsətdikdən sonra, `input` artıq istifadəçidən dəyər qəbul edən qurma funksiya deyil, adi sətr oldu. Çünki, qurma adlar fəzasında input adının dəyəri – tanıdığımız input funksiyasıdır, qlobal adlar fəzasında isə adi sətr! Axtarış ardıcıllığında qlobal adlar fəzası, qurma adlar fəzasından öndə olduğu üçün(LEGB qaydası), qlobal adlar fəzasındakı dəyər götürülür. Beləliklə, `input("Type something: ")` deməklə əslində dəyəri sətr olan dəyişəni funksiya kimi çağırmağa çalışırıq və yerili-yerində `TypeError` xətası alırıq. Hansı ki, xətanın ismarıcında bizə `'str' object is not callable` – sətr obyekti çağrılan deyil deyir. Ümumiyyətlə, bu cür qurma adlarla eyni adda dəyişənlər yaratmaq yaxşı hal deyil. Çünki, çox güman ki sonda gətirib bu cür xətalara çıxardacaq. Odur ki, öz adlarınızı python qurma adlarından fərqləndirmək üçün, ən azından öz adlarınızın önünə `_` əlavə edin, `input` yox, `_input` kimi.

5.7. Lambda ifadələr

Bəzən yazılması tələb olunan funksiya çox kiçik - bir neçə sətrlik ola bilir və bəlkə də bu funksiyadan kod içərisində yalnız bir yerdə istifadə edəcəyik. Belə olan təqdirdə *lambda ifadədən* istifadə etmək daha əlverişlidir. lambda ifadə - hesablama bloku bir ifadə şəklində yazıla bilən funksiyaları qısa - bir sətrlik ifadələrə çevirməyə imkan verir. Lambda ifadələr də adi funksiyalar kimi parametrlər alır və arqumentlərlə çağrıla bilirlər. Nümunə:

```
add = lambda a, b: a+b
```

Lambda ifadələr `lambda` açar sözü ilə başlayır, daha sonra vergül ilə qəbul edilən parametrlər sadalanır. Parametrlərdən sonra `:` qoyulur və nəticəsi qaytarılacaq ifadə yazılır. İfadə ixtiyari uzunluqda ola bilər, əsas məsələ budur ki - tək ifadə olsun. Lambda ifadə özü - funksiyaya bənzər obyektdir. Yuxarıdakı nümunədə `a` və `b` qəbul edib, `a+b` ifadəsinin nəticəsini qaytaran lambda ifadə yaradılır və add dəyişəninə mənimsədilib. Dəyişənə mənimsətdiyimiz obyektin tipinə baxsaq:

```
print(type(add))    # <class 'function'>
```

Deməli, həqiqətən də ifadə özü-özlüyündə adi funksiyadır və aşağıdakı ilə eynidir:

```
def add(a, b):
    return a+b
```

Lambda ifadəmizi rahatlıqla adi funksiya kimi çağıra bilərik:

```
print(add(1, 3))   # 4
```

Öncə dediyim kimi, hesablanan hissədəki ifadə ixtiyari uzunluqda ola bilər. Aşağıdakı nümunədə təyin olunmuş lambda ifadə bir `numbers` parametri alır və ancaq cüt elementlərdən ibarət siyahı qaytarır.

```
get_evens = lambda numbers: [i for i in numbers if i % 2 == 0]
```

```
print(get_evens([1, 2, 3, 4]))  # [2, 4]
```

Bəzi maraqlı qurma funksiyalar.

Mövzunu davam etdirməzdən əvvəl, bir neçə maraqlı qurma funksiya barədə danışmalıyam.

- `map` funksiyası - hər hansı bir ardıcıllığın bütün elementlərinə ardıcıl bir-bir bir funksiyaya ötürür və alınan cavablardan ibarət ardıcıllıq(generator - daha doğrusu) qaytarır. Baxaq:

```
    def sqr(x):
        return x**2

    print(map(sqr, [1,2,3]))  # <map object at ...>
```

Əvvəlcə, parametrlərə baxaq: map funksiyası iki parametr qəbul edib, birinci parametr - bütün elementlərə tətbiq olunacaq funksiyadır, bizim nümunədə bu `sqr` funksiyasıdır. Diqqətlə baxın, `sqr` funksiyasını çağırmırıq, bir obyekt kimi kimi `map` -ə ötürürük. İkinci parametr isə, bütün elementlərini funksiyaya ötürülməsini istədiyimiz ardıcıllıqdır, bizim nümunədə bu `[1,2,3]` ardıcıllığıdır. Hazırda, funksiyadan gözləntimiz budur ki, funksiya ardıcıllığın bütün elementlərini bir-bir götürüb, `sqr` -ə ötürüb, alınan nəticələri(kvadratları) bir ardıcıllıqda geriyə qaytaracaq. Amma, map bizə generator obyekti qaytarır, bu obyekti tanıdığımız siyahı tipinə çevirmək üçün `list` funksiyasına "bükə" bilərik:

```
    def sqr(x):
        return x**2

    print(list(map(sqr, [1,2,3])))   # [1, 4, 9]
```

map daha mürəkkəb quruluşa malik ola bilər: əgər funksiyamız 1-dən çox sayda parametr alısa, o zaman hər bir parametr üçün ayrıca ardıcıllıq ötürülür. Əgər funksiya 3 ədəd parametr qəbul edirsə, o zaman map içərisinə 3 ədəd ardıcıllıq ötürülməlidir. Üstəlik, ardıcıllıqlar eyni uzunluqda olmalıdır. Aşağıdakı nümunədə funksiya kimi `pow` istifadə edilib, funksiya bildiyiniz kimi iki ədəd parametrlə çalışa bilir, odur ki iki ədəd ardıcıllıq ötürürük.

```
    bases = [1000, 100, 10, 5]
    exponents = [0, 1, 2, 3]

    # 1000^0, 100^1, 10^2, 5^3
    print(list(map(pow, bases, exponents)))
    # [1, 100, 100, 125]
```

- `filter` funksiyası - map funksiyasına bənzəyir - funksiya və bir ardıcıllıq alır, amma ondan fərqli olaraq funksiyanı süzgəc-filter kimi istifadə edir. Funksiya adətən məntiqi tipli bir nəticə qaytarmalıdı və ardıcıllığın yalnız o elementləri geriyə qaytarılır ki - o elementlər üçün funksiya məntiqi cəhətdən `True` qiyməti qaytarmış olsun:

```
def is_even(n):
    return n % 2 == 0
```

```
list(filter(is_even, [1,2,3,4]))   # [2, 4]
```

Yuxarıdakı nümunədə ardıcıllığın elementləri bir-bir `is_even` funksiyasına ötürülür və yalnız o elementlər geriyə qaytarılacaq ardıcıllığa əlavə olunur ki onlar üçün `is_even` `True` qaytarmış olardı. Əslində, şərt deyil ki funksiya məhz *boolean*(məntiqi) cavab qaytarsın, belə ki, filter əslində funksiyanın qaytardığı nəticəni `bool` funksiyasına ötürür və onun qaytaracağı məntiqi cavaba görə hərəkət edir. Bildiyiniz kimi, `bool` boş olan hər şey üçün `False` , əks təqdirdə isə `True` qaytarırdı. Beləliklə, funksiyanız `0` və ya `1` qaytarsa belə, onlar `True` və ya `False` -a çevriləcəklər. Daha bir nümunə: yenə `employees` lüğətlər siyahısı var və `salary_morequal_than` adlı funksiyamız var. Funksiya bu lüğətlərdən birini alır və `amount` dəyişəninin qiymətindən böyük-bərabər olmasını yoxlayır - `True` və ya `False` qaytarır.

```
employees = [
       {'id': 0, 'first_name': 'Aziz', 'last_name': 'Nadirov',
       'age': 25, 'salary': 1500},
       {'id': 1, 'first_name': 'Aziz', 'last_name': 'Nadirli',
       'age': 26, 'salary': 1200},
       {'id': 2,'first_name': 'Den', 'last_name': 'Bruk',
       'age': 30, 'salary': 2000},
       {'id': 3,'first_name': 'Leyla', 'last_name': 'Rahimova',
       'age': 26, 'salary': 1400},
   ]

def salary_morequal_than(employee):
    amount = 1500
    return employee['salary'] >= amount

print(list(filter(salary_morequal_than, employees)))
# [{'id': 0, 'first_name': 'Aziz', 'last_name': 'Nadirov', 'age': 25, 'salary':
1500},
# {'id': 2, 'first_name': 'Den', 'last_name': 'Bruk', 'age': 30, 'salary':
2000}]
```

Lambda ifadələri çox zaman elə məhz `map` və `filter` ilə birlikdə istifadə edirlər. Çünki, biz lambda ifadəni bir-başa `map` və ya `filterin` çağırışında təyin edə bilərik:

```
employees = ... # öncəki kimi
filtered = filter(lambda data: data['salary'] >= 1500, employees)
print(list(filtered))
# [{'id': 0, 'first_name': 'Aziz', 'last_name': 'Nadirov', 'age': 25, 'salary':
1500},
# {'id': 2, 'first_name': 'Den', 'last_name': 'Bruk', 'age': 30, 'salary':
2000}]
```

Sonda, kiçik bir nümunə `map` ilə: ardıcıllığın cüt elementlərinin kvadratlarından, tək ədədlərinsə kublarından ibarət olan siyahı:

```
powereds = map(lambda n: n ** 2 if n % 2 == 0 else n ** 3, [1, 2, 3, 4, 5])
```

```
print(list(powereds))  # [1, 4, 27, 16, 125]
```

- `enumerate` funksiyası - ardıcıllığı "sadalamağa" xidmət edir. Tutaq ki, `t = ['a', 'b', 'c']` siyahısı var, `enumerate(t)` bizə `[(0, 'a'), (1, 'b'), (2, 'c')]` kimi say-element cütünü qaytarır. Gördüyünüz kimi, funksiya saymağa 0-dan başlayır, `start` parametri ilə istənilən ədəddən başlamaq olar. Funksiya geriyə generator qaytardığı üçün onu list funksiyasına bükmək lazımdır:

```
>>> list(enumerate(['A', 'B', 'C'], start=1))
[(1, 'A'), (2, 'B'), (3, 'C')]
```

- `all` funksiyası - ötürülmüş ardıcıllıqdakı dəyərlərin hamısı "doğru" olduqda doğru qaytarır. Yadınıza salım ki məntiqi nöqteyi nəzərdən boş olan hər şey "yanlış", əks halda isə "doğru" hesab olunur.

```
>>> all([True, True, False])
False
>>> all([True, True, True])
True
>>> all([1, 2, 3])
True
>>> all([1, 2, 0])
False
```

- `any` funksiyası - ötürülmüş ardıcıllıqda heç olmasa bir "doğru" qiymət olarsa, doğru qaytarır:

```
>>> any([1,1,0])
True
>>> any(['', (), 0])
False
>>> any(['', (), True])
True
```

5.8. Koll-stek və Rekursiya

Koll-stek

Bildiyiniz kimi, funksiya çağrıldıqda funksiyanın proqram kodu icra edilməyə başlayır. Deyək ki, 5-ci sətrdə funksiya çağrılır, proqram icrası keçir funksiyanın içərisinə və yalnız funksiya işini bitirdikdən sonra proqram icrası 5-ci sətrdən 6-cı sətrə keçir:

```
def foo():
    print("\tin foo")

print("Birinci sətr")
foo()
print("ikinci sətr")
```

```
Birinci sətr
    in foo
ikinci sətr
```

Şərti olaraq, əgər üç funksiyamız - `f1`, `f2` və `f3` varsa və `f1 - f2`-ni, `f2` isə `f3`-ü çağırarsa, eyni hal baş verər: `f1 - f2`-ni, `f2` isə `f3`-ü "gözləyəcək". Yəni, `f3` icrası bitəndən sonra `f2` davam edəcək, `f2` bitəndən sonra isə `f1`. Gəlin daha yaxşı təsəvvür üçün bütün bu mücərrədiyyəti koda, daha sonra isə şəklə çevirək:

```
def f3():
    print("in f3")

def f2():
    print("in f2")
    print("calling f3")
    f3()
    print("in f2: f3 is over")

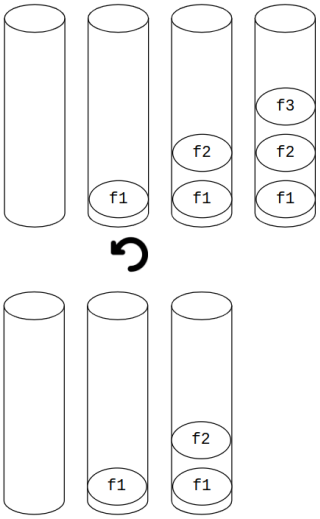
def f1():
    print('in f1')
    print("calling f2")
    f2()
    print("in f1: f2 is over")

print("line 1 in global")
print("calling f1...")
f1()
print("in global. f1 is over")
```

```
line 1 in global
calling f1...
in f1
calling f2
in f2
calling f3
in f3
in f2: f3 is over
in f1: f2 is over
in global. f1 is over
```

Gördüyünüz kimi, hər bir dəfə, bir funksiya çağrıldıqda - həmin funksiya tamamlanmamış proqramın icrası irəliləmir. Hər yeni bir çağırış bir növ "növbədə" irəliyə keçir və geridəkilər öndəkilərin bitməsini gözləyir. Bu quruluş *stek*(stack) adlanan verilən strukturunu yaradır. Stek strukturu bir növ ardıcıllıqdır, ancaq bu ardıcıllığa element əlavə etdikdə ancaq sona əlavə etmək, element silmək istədikdə də həmçinin ancaq sondan silə bilərik. Bu cür prinsipi bəzən *LIFO - Last In, First Out* da

adlandırırlar. Steki vizual olaraqyalnız üst hissəsi açıq olan silindr kimi təsəvvür etmək olar.Hər bir element silindrə atılmış bir topdur, silindrin eni topun enindən çox da böyük olmadığı üçün toplar bir-birinin üzərində dayanır. Yəni ən altda yerləşən topu çıxartmaq üçün məcburən bütün toplarlı çıxartmalısınız. Funksiyaların çağırışları da həmçinin stekə - silindrə salınmış toplardır. Bu stek - *çağırışlar steki*, və ya - koll-stek (call stack) adlanır.



Yuxarıdakı şəkil vizual olaraq öncəki kodun icrası zamanı çağırış stekini göstərir. Şəklin yuxarl hissəsində funksiyalar bir-birlərini çağırlar. Ən son çağrılan `f3` işini bitirdikdən sonra stek geriyə "yığılmağa" başlayır - şəklin aşağı hissəsi.

Rekursiya

Bir funksiya, digərini çağırdıqda proqram icrasının necə getdiyini, koll-stekin özünü necə apardığını bilirik. Bəs əgər funksiya özü-özünü çağırarsa nə baş verər? Təyin olunmuş hər hansı bir `f` funksiyası nə zaman icra olunur - çağrıldığı zaman. Funksiya icra olunarkən özünü çağırsa - özünü çağırdığı an stekə özü-özünü əlavə edəcək, yəni yenidən işə düşəcək. Bəs nə vaxta qədər ? Belə bir funksiya verilib:

```
1. def bad_foo():
2.     print("in foo")
3.     bad_foo()   # funksiya özünü çağırır
4.
5. bad_foo()   # ilk çağırış

# RecursionError xətası
```

Yuxarıda 5-ci sətrdə `bad_foo` çağrılır və proqram icrası funksiyaya keçir. İşə düşən funksiya nə edir - "in foo" çap edərək yenidən özünü çağırır. Bəs funksiya yenidən özünü çağırdıqda nə baş verir - eyni çey! - çap edib, özünü çağırır və beləcə sonsuzadək. Amma, yox, sonsuzadək yox. Çünki, funksiya hər dəfə özü-özünü çağırdıqda bütün normal çağırışlar kimi, bu çağırışlar da koll-stekə

yığılır, stek isə yaddaş tələb edir, sadə sonsuz dövrlərdən fərqli olaraq. Odur ki, funksiya özünü təqribən 1000 dəfə çağırdıqdan sonra - koll-stekə özünü 1000 dəfə yazandan sonra proqramımız `RecursionError` - rekursiya xətası ilə sonlanır. Funksiyanın özü-özünü çağırması - *rekursiya*, bu cür funksiyalar isə *rekursiv funksiyalar* adlanır. Yaxşı, bəs rekursiyanı necə səmərəli şəkildə tətbiq etmək olar? İstənilən səmərəli rekursiv funksiyanın *baza şərti* olmalıdır. *Baza şərti* - funksiyanın elə bir şərtidir ki, bu şərt ödəndikdə funksiya geriyə nə isə qaytarır. Bildiyiniz kimi, funksiyanın əsas "qabiliyyəti" geriyə nə isə qaytarmaqdır və rekursiv funksiya tutaq ki özünün 5-ci çağırışında geriyə nə isə qaytarsa - bu koll-stekin yığılmasına səbəb olacaq. Belə bir nümunəyə baxaq:

```
1. def foo(n):
2.   print(f"foo with n={n}")
3.   if n >= 5:    # baza şərti
4.       print('enough! Go back!')
5.       return None   # bəsdir, daha geri yığıl.
6.    else:
7.       foo(n+1)
8.       print(f"foo with n={n+1} is done!")
9.       return None
10.
11. foo(0)
```

```
foo with n=0
foo with n=1
foo with n=2
foo with n=3
foo with n=4
foo with n=5
enough! Go back!
foo with n=5 is done!
foo with n=4 is done!
foo with n=3 is done!
foo with n=2 is done!
foo with n=1 is done!
```

foo rekursiv funksiyasının baza şərti 3-cü sətrdə yazılan `n >= 5` şərtidir. Bu şərt ödəndikdə funksiyamız geriyə `None` qaytarır. Əks halda, 7-ci sətrdə funksiya özü-özünü çağırır, ancaq 1 vahid daha böyük `n` arqumenti ilə. Bu o deməkdir ki hər dəfə 7-ci sətrdə funksiya özü-özünü çağırdıqda bu çağırışlar koll-stekə yığılır, funksiya nəhayət baza şərtiti ödədikdə geriyə yığılmağa başlayır. `n=5` olduqda baza şərtimiz ödənir və funksiyamız geriyə `None` qaytarır. Bu o deməkdir ki `foo(5)` hardan çağrılıbsa, proqram icrası da ordan davam edəcək. Bəlli məsələdir ki, `foo(n=5)` ilə çağıran `foo(n=4)`-dir. `foo(n-5)` işini bitirdikdən sonra, proqram icrası qayıdır - `foo(n=4)`-ə, 8-ci sətrdən davam edir, daha sonra `foo(n=4)` də bitib qaytarır `foo(n=3)`-ə və beləcə ilk çağırışa qədər.

Faktorial nümunəsi. Rekursiyanı praktiki olaraq göstərmək üçün ən ideal məsələ - faktorialın rekursiv hesablanmasıdır. Yadınıza salım ki, ədədin faktorialı - 1-dən həmin ədədə qədər(ədədin özü də daxil olmaqla) bütün ədədlərin hasilidir və riyaziyyat aləmində `!` kimi işarə edilir. Məsələn, $5! = 5 * 4 * 3 * 2 * 1 = 120$. Diqqətlə baxsaq, görmək olar ki, bu nümunəni həm də

$5! = 5 * 4 * 3 * 2 * 1 = 5 * 4! = 120$ kimi yazmaq olar. `4!` özünü də $4 * 3!$ kimi yazmaq olardı. Burdan belə bir düstur yazmaq olar: $n! = n * (n - 1)!$ Yəni, istənilən müsbət `n` ədədinin(1-dən böyük olması məntiqli olar) faktorialı - ədədin özü, vurulsun (ədəd çıxılsın 1) faktoriala. Beləliklə, $2! = 2 * 1!$, $1!$ isə 1-ə bərabərdir. Bu riyazi funksiyanı rekursiv funksiya şəklində yazmaq olar, bunun üçün ! işarəsi əvəzinə `fact()` yazılışından istifadə edək:

```
fact(5) = 5 x fact(4)
    fact(4) = 4 x fact(3)
        fact(3) = 3 x fact(2)
            fact(2) = 2 x fact(1)
                fact(1) = 1    # <--| burdan yuxarı, qiymətləri yerlərinə qoyaq
            fact(2) = 2 x 1 = 2
        fact(3) = 3 x fact(2) = 3 x 2 = 6
    fact(4) = 4 x fact(3) = 4 x 6 = 24
fact(5) = 5 x fact(4) = 5 x 24 = 120
```

İndi isə, bütün bu söhbəti çevirək rekursiv funksiyaya. İlk öncə baza şərtimizi təyin edək. `fact(n)` funksiyasının baza şərti `n==1` olacaq, bu zaman `1` qaytaracağıq.

```
def recursive_fact(n):
    print(f"n={n}")
    if n == 1:
        return 1
    res =  n * recursive_fact(n - 1)
    print(f"calc {n} * recursive_fact({n-1}) = {res}")
    return res

recursive_fact(5)
```

```
n=5
n=4
n=3
n=2
n=1
calc 2 * recursive_fact(1) = 2
calc 3 * recursive_fact(2) = 6
calc 4 * recursive_fact(3) = 24
calc 5 * recursive_fact(4) = 120
120
```

> ⚠ **Diqqət**
>
> Faktorial funksiyasının bu cür - sadə rekursiv quruluşu işləsə də, heç də optimal deyil və nisbətən böyük rəqəmlər üçün işləməyəcək. Praktikada faktorial hesablamaq lazım gəldikdə `math.factorial` funksiyasından istifadə edə bilərsiniz. Üstəlik, kitabın "Əlavələr" bölməsində

izahlı alqoritmlər sırasında factorialın rekursiv hesablanmasının daha səmərəli üsulu göstərilib.

5.9. Qapanmalar və Dekoratorlar

Qapanmalar

Adlar fəzaları və LEGB qaydası ilə tanış olduğunuz üçün sizə Pythonda geniş istifadə olunan və adlar fəzalarına əsaslanan bir "fənd" göstərmək istəyirəm. Bu fənd həm də növbəti başlıqda öyrənəcəyimiz dekoratorların işləməsinə səbəb olan bir şeydir. Yəqin ki, adlar fəzalarından qapadan(enclosing) adlar fəzası yadınızdadır. Fərz edək ki, bir `outer` adlı funksiyamız var, hansı ki öz içərisində təyin olunmuş `inner` adlı funksiyanı qaytarır:

```
def outer():
       print("Hello from outer")
       def inner():
           print("Hello from inner")

       return inner

   outer()
```

Gəlin kodun nəticəsinə baxmadan düşünək - `outer` funksiyası çağrıldlqda nə baş verir? Əvvala ""Hello from outer"" çap olunur. Sonra isə funksiya geriyə öz içərisindəki `inner` **funksiyasını** qaytarır. Təkrar edirəm - `inner` funksiyasını çağırmır - funksiya obyektinin özünü qaytarır! Bu o deməkdir ki, biz `outer` funksiyasını çağırdıqda, o bizə inner funksiyasını verəcəkdir və əgər biz qaytarılan funksiya obyektini hər hansı dəyişənə mənimsətsək, ondan istifadə edə bilərik:

```
def outer():
    print("Hello from outer")
    def inner():
        print("Hello from inner")

    return inner

ifoo = outer()
print("Type of ifoo: ", type(ifoo))
ifoo()
```

```
Hello from outer
Type of ifoo:  <class 'function'>
Hello from inner
```

`outer()` çağrıldıqda geriyə funksiya qaytarır və biz bu funksiyanı `ifoo` adlı dəyişənə mənimsətdik. Faktiki olaraq, `ifoo` dəyişəninə `outer` içərisindəki `inner` funksiyasını mənimsətmişik. Belə olan təqdirdə `outer`-in işi bir növ funksiya yaradıb qaytarmaq olur. İndi isə,

belə bir maraqlı təcrübə aparaq: gəlin edə edək ki, bizim `outer` funksiyamız bir parametr alsın və `inner` o parametrdən istifadə etsin. Belə olan təqdirdə, həmin parametr `inner` üçün qurma adlar fəzasında olacaq.

```
def outer(x):
    print("Hello from outer")
    def inner():
        print(f"Hello from inner, x={x}")

    return inner

ifoo = outer(x=1)
ifoo()
```

```
Hello from outer
Hello from inner, x=1
```

`outer(x=1)` bizə elə bir `inner` funksiya qaytarır ki, o funksiya üçün `x` dəyişəninin dəyəri `1` olaraq "yadda saxlanılıb". Yəni, geriyə qaytarılan daxili funksiya - qaytarıldığı anda qurma adlar fəzasındakı - içərisində yerləşdiyi funksiyanın lokal adlar fəzasındakı bütün adları yadda saxlayır. Buna görə də `ifoo2 = outer(x=2)` icra etsək, `ifoo2` - a mənimsədilən `inner` funksiya üçün `x=2` olacaq. Belə bir nümunəyə baxaq:

```
def make_discounter(percent):
    def discounter(price):
        return price - (price * (percent/100))

    if percent < 0:
        print(f"negaitive value for percent.")
        return None
    # if ödəndiyi təqdirdə funksiya bitir, odur ki 'else' yazmasaq da olar
    return discounter

# 'endirimçi' funksiyalar
do_10 = make_discounter(percent=10)     # 10 faiz
do_50 = make_discounter(percent=50)     # 50 faiz
do_100 = make_discounter(percent=100)   # 100 faiz - pulsuz apar

print(do_10(1000))  # 900.0
print(do_50(1000))  # 500.0
print(do_100(1000)) # 0.0
```

`make_discounter` faiz ifadə edən `percent` arqumenti alır və geriyə - ötürülmüş qiymətə `percent` faiz endirim tətbiq edən funksiya qaytarır. `make_discounter` ilə uyğun olaraq 10, 50 və 100 faiz endirim tətbiq edən `do_10`, `do_50` və `do_100` funksiyaları yaradırıq. Bu funksiyaların hər birinin özünün yadda saxladığı bir `percent` dəyəri var. Daxili funksiya qaytarıldığı an qurma adlar fəzasındakı adları yaddaşına həkk edərək bir növ qapanır - ona görə də bu hadisəni "qapanma"

olaraq tərcümə etmişəm. İngilis dilli ədəbiyyatda isə *closures*, *factory functions* kimi də hallanır. Niyə "factory functions" (funksiya fabriki) , çünki çöl funksiya bir növ funksiya "istehsal" edən fabrik rolunu oynayır.

Dekoratorlar

Ötən dəfə sizinlə qapanmalar haqqında danışmışdıq. İndi isə, qapanmalara söykənən bir nümunəyə baxaq: təsəvvür edək ki `products` adlı məhsullar siyahısı var:

```
products = [
    {'id': 1, 'name': 'smartfon iphone 99 pro',  'category': 'smartphone',
'price': 10000},
    {'id': 2, 'name': 'smartfon xiaomi poco M5 128',  'category': 'smartphone',
'price': 400},
    {'id': 3, 'name': 'lenovo legion 5',  'category': 'laptops', 'price':
2000},
    {'id': 4, 'name': 'laptop dell inspiron 15',  'category': 'laptops',
'price': 1200},
    {'id': 5, 'name': 'airpods 99',  'category': 'headphones', 'price': 300},
]
```

İndi isə `get_price` adlı funksiya yazaq, hansı ki `id` və `products` siyahısını alıb, bizə lazım olan məhsulun qiymətini (`price` açarının dəyərini) qaytaracaq:

```
def get_price(id, products):
    # id-ə uyğun elementlər siyahı şəklində qaytarılır
    return [p['price'] for p in products if p['id'] == id]

get_price(id=2, products=products)   # [400]
```

İndi isə, funksiyamızın üzərinə bir qədər də "öhdəlik" atmaq istəyirik - istəyimizdən asılı olaraq, `50` faiz endirimli qiyməti qaytarmasını istəyirik. Məhz bu məsələ üçün o qədər də bəsid bir həll yolu olmaya bilər, amma biz bunun üçün qapanmalardan istifadə edə bilərik. Yadınızdadırsa, qapanmalar haqqda danışdıqda hətta, buna bənzər `make_discounter` funksiyasını da yazmışdıq. Bu dəfə isə belə edək: `with_discount(func)` adlı funksiya yazaq. Funksiya parametr kimi başqa bir funksiyanı - `func` alacaq, qapanma effektindən istifadə etmək üçün `with_discount` içərisində bir daxili funksiya `wrapper` təyin edək. `wrapper` bizim ötürdüyümüz `foo` funksiyasını çağıracaq, nəticəni alaraq ona 50 faiz endirim tətbiq edəcək və geriyə qaytaracaq. Baxaq:

```
def with_discount(foo):
    def wrapper(id, products):
        res = foo(id, products)
        p = 0.5      # 50%
        res = [r - (p*r) for r in res]    # hər bir nəhsul üçün endirim
        return res

    return wrapper
```

`with_discount` içərisinə `foo=get_price` ötürəcəyik, geriyə qaytarılan `wrapper` bizim `foo` - yəni `get_price` funksiyamızı çağırır və nəticəni `res` dəyişəninə mənimsədir. Daha sonra hər bir qiymət

üçün endirim tətbiq edərək geriyə qaytarır. Gəlin get_price funksiyamızı `with_discount` -a ötürək:

```
def with_discount(foo):
    def wrapper(id, products):
        res = foo(id, products)
        p = 0.5      # 50%
        res = [r - (p*r) for r in res]    # hər bir nəhsul üçün endirim
        return res

    return wrapper

def get_price(id, products):
    # id-ə uyğun elementlər siyahı şəklində qaytarılır
    return [p['price'] for p in products if p['id'] == id]

# endirimləşdirilmiş 'get_price' əldə edək
get_price_50 = with_discount(get_price)
# 2 id-li məhsul üçün
get_price_50(id=2, products=products)    #[200.0]
```

`get_price_50` dəyişəninə əslində `wrapper` funksiyasını mənimsətmişik, o da `get_price` kimi `id` və `products` arqumentlərini alır. Odur ki, `get_price` və `get_price_50` arasında fərqi demək olar ki hiss etmirik. Bir az da irəli gedib, `get_price = with_discount(get_price)` kimi yazsaq, hər dəfə `get_price` çağırdıqda endirimli qiymətləri alacağıq. Bu yazılışla sanki - "ilkin" `get_price` funksiyasını `with_discount` vasitəsilə `wrapper` ilə əvəzləyirik və yenidən get_price dəyişəninə mənimsədirik. Funksiya "üzdən" orijinala bənzəyir, di gəl ki artıq "o funksiya deyil". Bununla biz bir növ `get_price` funksiyasına "əlavə rəng", "əlavə funksionallıq" qatmış olduq, yazılış isə belə oldu:

```
get_price = with_discount(get_price)
```

`with_discount` kimi - bu cür funksiyalar *dekoratorlar* adlanır. Onlar ilk baxışdan bir `foo` funksiyasını alıb, `foo` yenidən geriyə qaytarırlar, ancaq bəzi "əlavələr" ilə. Bizim nümunədə bu əlavə endirim idi. "Dekoratorlaşdırmaya" məruz qalmış funksiyanı - *dekorlanmış* funksiya adlandıracam. Gördüyünüz kimi bütün möcüzə qapanmalara söykənir, iş orasındadır ki `wrapper` bizim ötürdüyümüz `get_price` funksiyasını `with_discount` funksiyasının `foo` parametri kimi yadda saxlayır və çağırır. Burda diqqət edilməli bir neçə məqam var: İlkin funksiyamızı çağırdığımızı zənn edərkən `wrapper` çağırmış oluruq. Odur ki, `wrapper` da ilkin funksiyanın aldığı bütün parametrləri ala bilməlidir. Əks halda dekorlanmış funksiya ilkin funksiyanın parametrləri ilə çalışmayacaq. Adətən `wrapper` ardıcıllıqların yığılması üçün `*` və `**` istifadə edir beləliklə ötürülən ixtiyari sayda adlı və ya adlı arqumentlərin hamısını ilkin funksiyaya ötürür. Növbəti nümunələrdə biz də elə edəcəyik.

Qısa yazılış.

Funksiyanı dekorlaşdırmağın sintaksis cəhətdən daha qısa yolu var: dekorlamalı olan funksiyanı təyin etməzdən əvvəl `@<dekorator_funksiya>` yazmaqla funksiyamızı elə "bələkdəcə" dekorlamış olduq. Öncəki nümunədəki `get_price` funksiyasını `with_discount` ilə dekorlamaq üçün `get_price` funksiyanı belə təyin etmək olardı:

```
@with_discount
def get_price(id, products):
```

```
    # id-ə uyğun elementlər siyahı şəklində qaytarılır
    return [p['price'] for p in products if p['id'] == id]
```

Bununla da daha `get_price = with_discount(get_price)` yazmağa ehtiyac qalmır.

Daha bir ssenari - təsəvvür edin ki, bütün mallara 50% endirim etmək əvəzinə, hər bir mala ayrıca endirim tətbiq edilməlidir. Deyək ki bir `discount = {'id': percent, 'id_2': percent}` kimi lüğətimiz var və bu lüğətdə `id` malın id kodu, `percent` isə tətbiq edilməli olan endirim faizidir. Indi isə, endirimləri bu lüğət əsasında tətbiq edəcək `with_discount_2` dekoratorunu yazaq:

```
# dekorator
def with_discount_2(foo):
    discounts = {1: 0.50, 2: 0.50, 3: 0.20, 4: 0.10}
    def wrapper(id_, products):
        res = foo(id_, products)
        p = discounts.get(id_, 0)     # 'id' endirimdə olmasa, 'get' 0 qaytacaq
        res = [r - (p*r) for r in res]    # hər bir nəhsul üçün endirim
        return res

    return wrapper

# dekorlanmış funksiya
@with_discount_2
def get_price(id_, products):
    # id-ə uyğun elementlər siyahı şəklində qaytarılır
    return [p['price'] for p in products if p['id'] == id_]

print(get_price(id_=1, products=products))      # [5000.0]  -50%
print(get_price(id_=3, products=products))      # [1600.0]  -20%
print(get_price(id_=5, products=products))      # [300.0]   -0%
```

Bu dəfə, "@" sintaksisindən istifadə etdik və bütün mallara 50 faiz endirimlər tətbiq etmək əvəzinə, "endirimlər lüğətindən" istifadə etdik.

Parametrli dekoratorlar.

"@" suntaksisindən istifadə etdikdə dekorator funksiya yalnız bir parametr ala bilir, o da - dekorlanan funksiya olur. Dekoratorun əlavə arqumentlər ala bilməsini təmin etməyin ən yaxşı yolu - onun özünü ayrı bir funksiya daxilində yazmaq lazımdı. Bu çöl funksiya lazım dekorlanmalı funksiyadan başqa, lazım olan bütün arqumentləri alır və geriyə dekorator funksiya obyektini qaytarır. Qapanma effektinə görə, dekorator - içərisində yerləşdiyi funksiyanın daxilindəki adları - ötürdüyümüz arqumentləri yadda saxlayır. Bu dolaşıq məsələyə aydınlıq gətirməkçün, belə bir şey edək: öncə bizdə `with_discount_2` adlı dekorator vardı, onun adını dəyişib `inner` edək və bu `inner` -in özünü yeni yaradacağımız `apply_discount` funksiyasının içərisinə yazaq. `apply_discount` bizdən `discounts_dict` - endirimlər lüğətini alacaq və geriyə `inner` adlandırdığımız dekorator funksiyamızı qaytaracaq:

```
# dekorator
def apply_discount(discounts_dict):
    def inner(foo):
        def wrapper(id_, products):
```

```
            res = foo(id_, products)
            p = discounts_dict.get(id_, 0)    # 'id' endirimdə olmasa, 'get' 0
qaytacaq
            res = [r - (p*r) for r in res]    # hər bir nəhsul üçün endirim
            return res

        return wrapper

    return inner

@apply_discount(discounts_dict={1: 0.50, 2: 0.50, 3: 0.20, 4: 0.10})
def get_price(id_, products):
    # id-ə uyğun elementlər siyahı şəklində qaytarılır
    return [p['price'] for p in products if p['id'] == id_]

print(get_price(id_=1, products=products))      # [5000.0]  -50%
print(get_price(id_=3, products=products))      # [1600.0]  -20%
print(get_price(id_=5, products=products))      # [300.0]   -0%
```

Nəticədə, biz artıq funksiyamızı dekorlayarkən, istifadə olunmalı endirimlər lüğətini ötürə bilirik. Bu cür quruluş daha səmərəlidir. Çünki, endirimlər dinamik bir şeydir və tez-tez dəyişə bilər. Hər dəfə dekorator içərisində əl ilə lüğətə düzəliş etmək yaxşı fikir deyil, həmçinin, yaxşı funksiya fəlsəfəsinə görə, lazım olan bütün verilənlər (ələxsus onlar dinamikdirsə) xaricdən arqumentlər şəklində daxil olmalıdır. Üstəlik, çox güman ki endirimlər bir başa lüğətdən deyil, hər hansı bir excel cədvəlindən oxunub lüğətə çevriləcək(bu haqqda hələ söhbət gedəcək), odur ki, lüğəti "hardcode" olaraq funksiyanın içərisinə mismarlamaq lazım deyil. Öncə dediyim kimi, məsələni yüz üsulla həll etmək olar, əsas məsələ - ən rahat və ən effektiv üsulla həll etməkdir.

5.10. İteratorlar və Generatorlar

İteratorlar

İndiyə qədər dəfələrlə "iterasiya" sözündən istifadə etmişəm. Hər hansı bir ardıcıllığı iterasiya etmək, iterə etmək - onun elementlərini bir-bir götürmək, incələmək deməkdir. Məsələn, `lst=[1,2,3]` siyahısı var, `for i in lst` kimi yazdıqda `lst` siyahısını irerə etmiş oluruq. Bəs bütün obyektlər "iterasiya olunabiləndir"(iterable) ? Əlbəttə ki, xeyir. Məsələn, götürək ədədləri - biz `for i in 2.5` kimi yazsaq, bu nə anlama gələ bilər? Deməli, ancaq ardıcıllıqları iterə edə bilirik. Yaxşı, iterasiya olunan və oluna bilməyən obyektlər bir-birindən necə fərqlənir? H'ər bir iterasiya oluna bilən obyektin - *iterator* adlanan xüsusiyyəti olur. Biz ardıcıllığı iterə etdikdə, onun özünü yox, iteratorunu iterə etmiş oluruq. Yalnız iterasiya oluna bilən obyektlərin iterator xüsusiyyəti olur. Obyektin iteratorunu almaq üçün `iter` funksiyasından və ya obyektin `__iter__` metodundan istifadə edə bilərik. Metod yalnız iterəolunan obyektlərdə olur, odur ki, iterəolunmayan obyektin `__iter__` metodunu çağırmağa cəhd etsəz, xəta alacaqsınız.

```
print([1,2,3].__iter__())   # <list_iterator object at ...>
print(iter([1,2,3]))        # <list_iterator object at ...>
print(iter(666))        # TypeError: 'int' object is not iterable
```

Arxa fonda obyekti iterə etmək üçün məhz `__iter__` metodu istifadə olunur. Baxaq:

```
for i in [1,2,3].__iter__():
    print(i, end=',')
# 1,2,3,
```

`for` dövrü də əslində arxa fonda bu cür davranır. İterasiya prosesinin özü - iterator obyektinin hər dəfə "növbəti element" qaytarması hesabına baş verir. Iteratorların `__next__` metodu bizə iteratorun növbəti elementini qaytarır. İlk çağırışda - birinici, ikinci çağırışda - ikinci və beləcə sonuncu elementə qədər. Əgər sonuncu element artıq qaytarılıbsa və geriyə qaytarmağa daha heç nə yoxdursa `StopIteration` xətası baş verir. Baxaq:

```
>>> lst = [1,2,3]       # siyahı
>>> iter_lst = iter(lst)  # onun iterator obyekti
>>> print(iter_lst)
    <list_iterator object at ...>
>>> iter_lst.__next__()  # ilk çağırış - ilk element
    1
>>> iter_lst.__next__()   # ikinci
    2
>>> iter_lst.__next__()   # üçüncü və sonuncu
    3
>>> iter_lst.__next__()   # daha element yoxdur
    Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
    StopIteration
```

İteratorda neçə elementin qaldığını bilmək üçün `__length_hint__` metodundan istifadə etmək olar. Üstəlik, `__next__` metodundan əlavə olaraq, `next` funksiyasından da istifadə etmək olardı, heç bir fərqi yoxdur, çünki `next(i)` çağırışı arxa fonda `i.__next__()` icra edəcək.

Generatorlar

Nə üçün lazımdır. Mövzuya bir qədər uzaqdan yaxınlaşaq - gəlin bir dəfə də `range` funksiyasına nəzər salaq. Funksiya üç arqument qəbul edir: `start` - burdan, `stop` - bura qədər və `step` - bu addım ilə ədədi aralıq verirdi. Bilirsiniz ki, hər bir obyekt əməli yaddaşda yerləşən bir veriləndir və o deməkdir ki baytlarda müəyyən həcmdə yer tutur. Obyektin yaddaş həcmini öyrənmək üçün `__sizeof__` metodundan istifadə olunur, metod geriyə obyektin baytlarla ifadə olunmuş həcmini qaytarır. Təsəvvür edin ki `r = range(1, 10_000_000)` kimi yazmaqla bir milyonluq ardıcıllıq almaq istəyirik. Belə bir "ardıcıllığın" həcmi nə qədər olar?

```
>>> r = range(1, 10_000_000)
>>> r.__sizeof__()
48
>>> r
range(1, 10000000)
```

Yəqin ki hiss etdiniz ki nəsə düz deyil, 10 milyonluq ardıcıllığın cəmisi 48 bayt həcm tutması heç inandırıcı deyil, üstəlik də `r` bizə normal ardıcıllıq yox hansısa `range(1, 10000000)` qaytarır. Gəlin `r` dəyişəninin dəyərini siyahıya çevirib baxaq:

```
>>> r_list = list(r)
>>> r_list.__sizeof__()
80000040
```

`80_000_040` bayt! `r_list` dəyişənini çap etməyi heç cəhd belə etmirəm, çünki içərisində 10 milyon ədəd var... Yaxşı, bəs onda `range` funksiyası bizə nə verir? Niyə həcmi bu qədər az olur? İş burasındadır ki, `range` geriyə *generator* qaytarır. *Generator* - hər "növbəti element" tələb olunanda onu "yaradan", "generasiya" edən bir funksiyadır. Növbəti elementi almaq bir növ - obyektin iteratorunu `next` funksiyasına ötürməkdir. Bu o deməkdir ki, generator-funksiya yalnız biz tələb etdikdə həmin elementi yaradır və bizə qaytarır. Buna görə də `range` funksiyasının geriyə qaytardığı obyekt bu qədər "yüngül" idi, çünki içərisində heç bir element filan yox idi - sadəcə olaraq "növbəti element necə hesablanmalıdır" qaydası vardı. Bununla da generatorlar bir növ sxemə bənzəyirlər, bu sxem əsasında ardıcıllıq "toplanır", "yığılır". Biz generatoru siyahıya çevirdikdə generatorun bütün elementləri alındı(təsəvvür edin ki, `StopIteration` xətası çıxana qədər `next` çağrılır) və siyahıya əlavə olundu. Yaxşı, bəs generator-funksiyalar necə yaradılır - adi funksiyalar kimi, sadəcə olaraq `return` əvəzinə `yield` açar sözü yazılır. Əgər adi funksiyada `return` icra olunan anda funksiya bitirdisə, generatorlarda yalnız bir iterasiya bitmiş olur, yəni `yield` -in önündəki ifadənin nəticəsi `next(<generator>)` çağırışının nəticəsi kimi qaytarılır, növbəti dəfə `next(<generator>)` çağırdıqda `yield` -dan sonrakı hissə icra olunur. Baxaq:

```
def sqrs(start, stop):
    for i in range(start, stop):
        yield i ** 2

gen = sqrs(1, 5)
print(type(gen))    # <class 'generator'>

print(next(gen))    # 1
print(next(gen))    # 4
print(next(gen))    # 9
print(next(gen))    # 16
print(next(gen))    # xəta - StopIteration
```

`sqrs` generatoru `start` - `stop` aralığındakı ədədlərin kvadratlarını qaytarır. Adətən `yield` dövrlərin içərisində yazılır, funksiya sona çatdıqda `StopIteration` baş verir. Belə bir nümunəyə baxaq - aşağıda "sonsuzadək" fibonaçi ardıcıllığını generasiya edəcək `get_fibo` generatoru yazılıb. Fibonaçi ardıcıllığında hər bir element - özündən öncəki iki elementin cəminə bərabərdir:

```
def get_fibo():
    # ilk 2 elementi əl ilə doldurmuşam
    sequence = [1, 1, ]
    for i in sequence:
        yield i

    # sonrakıları dövr daxilində
    i=1
    while True:     # sonsuz dövr
        num = sequence[i] + sequence[i-1]
        sequence.append(num)
```

```
        i += 1
        yield sequence

fibo = get_fibo()   # generator
for i in range(1, 11):
    print(f"[{i}]: {next(fibo)}")
```

```
[1]: 1
[2]: 1
[3]: [1, 1, 2]
[4]: [1, 1, 2, 3]
[5]: [1, 1, 2, 3, 5]
[6]: [1, 1, 2, 3, 5, 8]
[7]: [1, 1, 2, 3, 5, 8, 13]
[8]: [1, 1, 2, 3, 5, 8, 13, 21]
[9]: [1, 1, 2, 3, 5, 8, 13, 21, 34]
[10]: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Yuxarıdakı `get_fibo` generator funksiyası hər next çağırışında fibonaçi ardıcıllığının növbəti elementini qaytarır. Əslində `for i in fibo` da yaza bilərik, sadəcə bu zaman sonsuz for dövrü yaranacaq, çünki bizim `fibo` generatorumuzun dayanma şərti yoxdur - sonsuza qədər ardıcıllıq generasiya edəcək. İndi, generatoru ilk `n` həddi qaytarmağa proqramlaşdıraq:

```
def get_fibo(n):
    # n elementli fibo ardıcıllığı
    # ilk 2 elementi əl ilə doldurmuşam
    if n < 1: return None   # yanlış n
    if n == 1: return 1
    elif n == 2: return [1, 1, ]
    sequence = [1, 1, ]
    for i in sequence:
        yield i
    # sonrakıları dövr daxilində
    for i in range(1, n-2): # iki iterasiya artıq yuxarıda edilib
        num = sequence[i] + sequence[i-1]
        sequence.append(num)
        yield sequence

for i, fibo_i in enumerate(get_fibo(11), start=1):
    print(f"[{i}]: {fibo_i}")
```

```
[1]: 1
[2]: 1
[3]: [1, 1, 2]
[4]: [1, 1, 2, 3]
[5]: [1, 1, 2, 3, 5]
```

```
[6]: [1, 1, 2, 3, 5, 8]
[7]: [1, 1, 2, 3, 5, 8, 13]
[8]: [1, 1, 2, 3, 5, 8, 13, 21]
[9]: [1, 1, 2, 3, 5, 8, 13, 21, 34]
[10]: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

`yield from`.

Təsəvvür edin ki, generator funksiya daxilində başqa bir generator funksiyanı iterə etməlisiniz. Belə olan təqdirdə ilk ağıla gələn bunu `for` dövrü ilə etməkdir. Bu cür hallarda sintaksisi sadələşdirmək və prosesi optimallaşdırmaq üçün python 3.3-dən etibarən `yield from` istifadəyə verilmişdir. Baxaq:

```
def generator1():
    for i in [1,2,3]:
        yield i

def generator2():
    for i in ['a', 'b', 'c']:
        yield i

def mega_generator():
    yield from generator1()
    yield from generator2()

for i in mega_generator():
    print(i, end=',')     # 1,2,3,a,b,c,
```

`yield from generator1` əvəzinə `for j in generator1: yield j` yazmaq olardı, ancaq `yield from` daha optimal variantdır.

5.11. Annotasiyalar və Type Hinting

Python 3.5-dən başlayaraq Pythonda annotasiyalar və type hinting(tip göstəricisi) gətirildi. Əslində, bu mövzu heç də funksiyalarla məhdudlaşmır, sadəcə funksiyalarla tam tanış olduqdan sonra, arsenalımızı zənginləşdirə biləcəyimiz ən yaxşı və yerli alətlərdən biri olduğu üçün məhz bu məqamda sizi tanış etmək istəyirəm. Gəlin qısaca bunlara baxaq.

Annotasiyalar

Annotasiyalar(annotation, doc string) - funksiya, metod və hətta modulların başlığında yazılmış, funksiya, metod və ya modul barədə qısa tanıtım mətnidir: funksiya/metod/modul - nə iş görür funksiyadırsa - parametr kimi nə qəbul edir və nə qaytarır və sairə. Mətn adətən bir sətrdən böyük olduğu üçün, üçqat dırnaqlar içərisində yazılır.

```
def add(a, b):
    """ ötürülən iki ədədi cəmləyir.
        Params:
            a: int - birinci cəm
            b: int - ikinci cəm
        Returns:
```

```
            a və b ədədlərinin cəmi
    """
    return a + b

add(2,3)    # 5
```

Gördüyünüz kimi, bu mətn funksiyanın işinə heç də təsir etmir. İlk dəfə üçün əsas mətni Azərbaycanca yazmışam, amma praktikada həmişə ingilis dilində yazılır. Göründüyü kimi, ilk öncə ümumi təyinat yazılır, daha sonra parametrlər təsvir edilir və sonda qaytarılan dəyər. Annotasiyanın yazılmasının tək bir qaydası yoxdur, hər kəs "+-" ortaq standarta bənzədib yazır. Bəs praktikada annotasiyanın faydası nədir? Bir tərəfdən kodu oxuyarkən əlavə məlumat almış oluruq, həm özümüz yadımıza salmış oluruq, həm də digər bir tərtibatçı baxarsa başa düşmək asan olar. Digər tərəfdən, fayda heç də bununla yekunlaşmır: pythonda olan `help` funksiyası obyekt haqqında məlumat qaytararkən yazılmış annotasiyanı oxuyur və çap edir(qaytarmır).

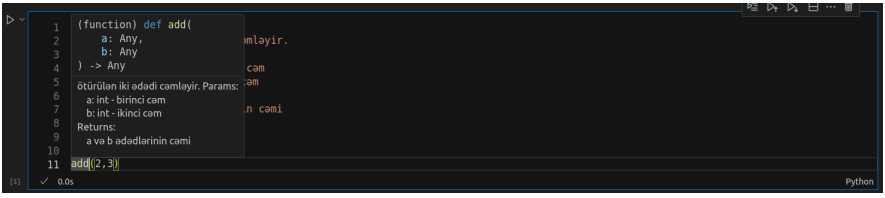
```
print(help(add))
```

```
Help on function add in module __main__:

add(a, b)
    ötürülən iki ədədi cəmləyir.
    Params:
        a: int - birinci cəm
        b: int - ikinci cəm
    Returns:
        a və b ədədlərinin cəmi

None
```

Bu da son deyil, belə ki, bəzi ağıllı mühitlər(VS Code, Pycharm və s.) bu annotasiyaları avtomatik oxuyur və biz deyək ki kursoru funksiyanın adının üzərinə gətirdikdə həmin məlumatı bizə göstərir(şəkildə VS Code göstərilib):



Tip Göstəriciləri

Type hinting - tip göstəriciləri annotasiyalar kimi yardımçı xarakter daşıyır və konkret dəyişənin və ya funksiya və ya metodun geriyə qaytardığı dəyərin tipini göstərmək üçün istifadə olunur. Dəyişənin tipini göstərmək üçün - dəyişənin adından sonra `:` qoyulur və tip yazılır. Tip dedikdə, tip funksiyaları nəzərdə tutulur(sadə hallarda), məsələn `int`, `str`, `float`, `list`, `bool`, `dict` və

sairə. Üstəlik, funksiyanın/metodun geriyə qaytardığı dəyərin tipini də göstərə bilərik, bunun üçün funksiyanın adından sonra `->` işarəsi qoyulur. Gəlin yuxarıdakı `add` funksiyamıza tip göstəriciləri əlavə edək: a və b parametrləri `float` tipli dəyişənlər olmalıdır və funksiyanın geriyə qaytaracağı nəticənin tipini də `float` kimi göstərək.

```
def add(a: float, b: float)->float:
    """ ötürülən iki ədədi cəmləyir.
        Params:
            a: int - birinci cəm
            b: int - ikinci cəm
        Returns:
            a və b ədədlərinin cəmi
    """
    return a + b

add(2,3)   # 5
```

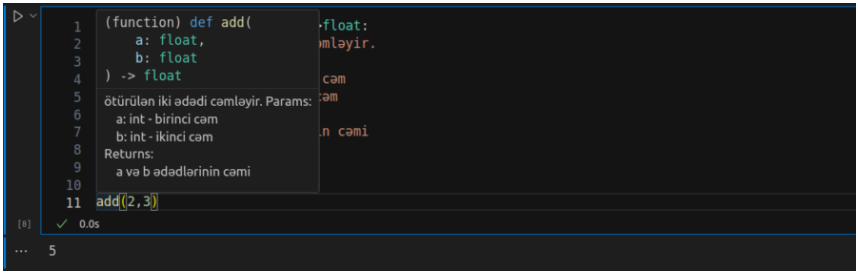
`help` funksiyasında bir də baxaq:

```
help(add)
```

```
Help on function add in module __main__:

add(a: float, b: float) -> float
    ötürülən iki ədədi cəmləyir.
    Params:
        a: int - birinci cəm
        b: int - ikinci cəm
    Returns:
        a və b ədədlərinin cəmi
```

VS Code - un göstərdiyinə də baxaq:



Göründüyü kimi, tip göstəriciləri də annotasiyalar kimi `help` və mühit yardımçı mətnlərində istifadə olunur. Qeyd etmək lazımdır ki, tip göstəricisi heç bir məhdudiyyət tətbiq etmir, sadəcə göstərmək üçün istifadə olunur. Odur ki, əgər biz `a` dəyişəninin tipini `float` kimi göstərib, realda qeyri `float` ötürsək, heç nə olmaz. Yuxarıdakı funksiya üçün `a` və `b` üçün iki sətr ötürsək, geriyə bu sətrlərin

konkatenasiyasını qaytaracaq. Çünki, python verilən kodu icra edir və əgər icra zamanı proqramı sındıran heç nə baş vermirsə, proqram icrası davam edir:

```
print(add("hello ","world"))    # 'hello world'
```

> **Diqqət**
>
> Biz burada yalnız ən bəsit tip göstəriciləri nümunələrinə baxdıq. Tip göstəriciləri `int`, `str` kimi standart tiplərdən savayı daha mürəkkəb göstəricilər ala bilər, biz dəyişənin hansı tipli elementlərdən ibarət ardıcıllıq olacağını, hansı tip açar və dəyərlərdən ibarət olacaq lüğət olacağını və sairə göstərə bilərik. Əlavə olaraq, əgər dəyişən bir neçə tipə aid ola bilərsə, o zaman o tiplər `Union[]` içərisində sadalanır və ya Python 3.11-dən sonra aralarında `|` işarəsi yazılmaqla sadalanır. `Union` və buna bənzər digər alətlər `typing` modulunda yerləşir. Digər tərəfdən də göründüyü kimi, bu mövzu hazırda Pyhtonda aktiv inkişaf etdirilir, dəyişdirilir. Odur ki, bu mövzu barədə kitabda Python 3.11 üçün doğru olan, Python 3.10 üçün yanlış ola bilər. Ona görə də nümunələrdə tip göstəriciləri barədə nəsə yeni bir şey göstərdikdə, onun təsvirini də verəcəm, belə ki, bu mövzunu bir yerdə tam əhatə etmək xeyirdən çox ziyan verəcək.

Tapşırıqlar

1. Bir `ascii_az(string)` funksiyası yaradın. Funksiya, Azərbaycan əlifbası hərflərini - onların ingilis dili qarşılığı ilə əvəz edəcək. Əvəzləmə üçün cütlər: `ə-e, ü-u, ö-o, ş-sh, ç-ch, ğ-gh, ı-i`

   ```
   def ascii_az(string: str) -> str:
       # ...

   ascii_az("Python öyrənirəm!")
   # Python oyrenirem!
   ```

2. Sətri qəbul edib, onun tərsini qaytaran lambda ifadə yazıb `reverser` dəyişəninə mənimsədin.

   ```
   reverser = lambda ... # sizin lambda ifadə
   reverser("hi!")   # "!ih"
   ```

3. Elə bir `cleaner` lambda ifadəsi yazın ki: sətr qəbul edir, sətrdəki kənar(sağ və sol kənarlardan) boşluqları silsin, aşağı registrə(kiçik hərflə) gətirib qaytarsın. Əgər boşluqlar silindikdən sonra sətr boş olursa, o zaman `None` qaytarsın.

   ```
   cleaner = lambda ...  # sizin lambda ifadə
   cleaner(" Salamlar olsun! ")  # "salamlar olsun!"
   cleaner("   ")  # None
   ```

4. Fibonaçi ardıcıllığının n-ci həddini qaytaran funksiya yazın. Fibonaçi ardıcıllığında hər bir element, özündən öncəki iki elementin cəminə bərabərdir. Ardıcıllığın ilk 7(0 da sayılmasa 6) elementi belədir: `[0,1,1,2,3,5,8]`.

```
def fibonacci(n: int)->int:
    # Base cases: fib(0) = 0, fib(1) = 1
```

```
    # Recursive case: fib(n) = fib(n-1) + fib(n-2)
    pass

fibonacci(6)   # 8
```

5. Qapanmalardan istifadə edərək `create_counter` sayğac qaytaran funksiyasını yaradın. Funksiya geriyə sayğac funksiyası qaytarmalıdır, bu funksiyanı hər dəfə çağırdıqda öncəki çağırışdan bir vahid böyük ədəd qaytarmalıdır. `create_counter` fabrik funksiyası `start` arqumentini almalıdır, bu arqument sayğacın neçədən başlamalı olduğunu təyin edəcək, susmaya görə dəyəri `0` olmalıdır.

```
def create_counter(start: int=0):
    # Create a closure that maintains a counter
    # Each call should increment and return the current count
    def counter()->int:
        # Your code here
        pass
    return counter

# Test cases
counter1 = create_counter()
counter2 = create_counter(10)

print(counter1())  # Should output: 1
print(counter1())  # Should output: 2
print(counter2())  # Should output: 11
print(counter1())  # Should output: 3
print(counter2())  # Should output: 12
```

6. `counter` adlı dekorator funksiya yaradın. Dekorator, bürünən funksiyanın hər çağırılışı zamanı, neçənci dəfə çağrıldığını çap etməlidir.

```
def counter(func):
    # Your implementation here
    pass

@counter
def greet(name):
    return f"Hello, {name}!"

greet("Ali") # greet function called 1 time(s)
greet("Ali") # greet function called 2 time(s)
greet("Ali") # greet function called 3 time(s)
```

7. Generatorlardan istifadə edərək, öz `range` funksiyanızı yazmağa çalışın. Funksiya `start`, `stop` və `step` arqumentlərini alacaq. Susmaya görə `start=0`, `step=1` olacaq.

```
def custom_range(start, stop, step=1):
   """
   Custom range generator function

   Args:
       start: Starting value (inclusive)
       stop: Ending value (exclusive)
```

```
        step: Step size (default: 1)
    """
    # Funksiyanın generator olması üçün yield istifadə edin.
    # `step` parametri həm mənfi, həm də müsbət ola bilər.
    pass

print(custom_range(0, 4))
# <generator object custom_range at ...>
print(list(custom_range(0, 4)))
# [0, 1, 2, 3]
```

Link: https://github.com/AzizNadirov/python-road-book/blob/master/code/tasks/functions_5.ipynb

6. 1. Xəta, ya istisna ?

İndiyə qədər proqramımızı sonlandıran, sındıran hadisəyə "xəta" deyirdik. Əslində isə, xarici ədəbiyyatda bunun üçün iki termin istifadə olunur: *error* - xəta və *exception* - istisna. Bildiyiniz kimi, python interpretə olunan dildir və proqramı işə saldıqda heç bir kompliyasiya prosesi baş vermir(Python VM və JIT haqqında sonra danışacağıq), proqram işə düşür, "ilkin" yoxlamadan sonra kod işə düşür. Bəzən elə olur ki, kod heç işə düşmədən "çökür". Buna səbəb kodun yazılış qaydasının - sintaksisinin səhv olmasında ola bilər. Yəni, ən azından kod düzgün şəkildə - dilin sintaksisinə uyğun yazılmalıdır ki icra edilsin. Bu xəta idi. Əlavə olaraq, kod işə düşdükdən sonra nəsə düz getməyə bilər - sıfıra bölməyə cəhd, təyin olunmamış dəyişəndən istifadəyə cəhd, ardıcıllığın mövcud olmayan indeksinə müraciyət və sair ilaxır. Bunlar hamısı məntiqi xətalardır. Yəni, bir növ bizim bir proqramçı kimi düşündüyümüz kod - alqoritm daxilində gözdən qaçırdığımız, gözləntilərimiz üçün "istisna" olan hallardır. Görünür məhz buna görə onları - exception, yəni istisna adlandırırlar. Mən isə, çaşqınçılıq yaratmamaq üçün "xəta" kimi adlandırmağa davam edəcəm.

6.2. Pythonda xətalar

İndiyə qədər n qədər xətalarla üzləşmisiniz. Xəta baş verdikdə etməli olduğumuz şey - xətanın harada və nə üçün baş verdiyini aydınlaşdırmaqdır. Ona görə də müdrik insanlar proqramlaşdırma dillərində xəta baş verdiyi zaman biz - istifadəçilərə maksimum dərəcədə kömək etmək üçün baş verən xətalar haqqında məlumat təqdim edirlər. Mənim kompüterimdə `/home/anadirov/Documents/LTA/exc.py` ünvanında `exc.py` alı bir python skripti var, skriptin içərisindəki kod:

```
def foo(n):
    print(n / 0)    # sıfıra bölmə

foo(10)
```

Bəlli məsələdir ki kod işə düşdükdən sonra çökəcək, çünki funksiya daxilində sıfıra bölmə əməliyyatı var, bildiyimiz kimi 0-a bölmənin nəticəsi riyazi olaraq təyin edilməyib. Ola bilər ki skriptin yolu sizi çaşdırsın, səbəb windows deyil, linux əsaslı ƏS istifadə etməyimdir. İndi, kopu işə salıram və belə bir xəta ismarıcı alıram:

```
Traceback (most recent call last):
  File "/home/anadirov/Documents/LTA/exc.py", line 4, in <module>
    foo(10)
  File "/home/anadirov/Documents/LTA/exc.py", line 2, in foo
    print(n / 0)    # sıfıra bölmə
ZeroDivisionError: division by zero

Process finished with exit code 1
```

Burada, bizim üçün faydalı olacaq 4 şey var:

1. **Xəta baş verənə qədər proqramın keçdiyi yol** - `Traceback` (treysbək): bizim halda traceback budur:

```
File "/home/anadirov/Documents/LTA/exc.py", line 4, in <module>
 foo(10)
File "/home/anadirov/Documents/LTA/exc.py", line 2, in foo
 print(n / 0)    # sıfıra bölmə
```

Bu yazılanların ilk sətrindən belə qənaətə gəlmək olar ki, ilk icra olunan `exc.py` skriptində, 4-cü sətrdəki `foo(10)` çağırışıdır. Daha sonra, treysbəkin ikinci sətrin dediyinə görə, daha sonra elə həmin skriptin 2-ci sətrindəki `print(n / 0)` çağırışı olubdur və bundan sonra proqram çöküb. Yəni bütün bunlar bizə proqramın keçdiyi yolu izləməyə imkan verir. Ola bilər ki, bizim kodda onlarla funksiyalar, modullar var, birinci-ikincini, ikinci-üçüncünü çağırır və sairə. Belə olan təqdirdə stəktreys bizə xeyli kömək etmiş olacaq, çünki proqramın hardan-hara və necə gəldiyini görə biləcəyik. 2. **Xətanın adı və ismarıcı** - Pythonda hər bir xətanın adı və ismarıcı var. Xətanın adına görə xətanın nə tip bir şey olduğunu başa düşürük. Bizim halda, xətanın adı - `ZeroDivisionError` (hərfi mənada tərcümədə - SıfıraBölməXətası), ismarıcı isə - `division by zero` (sıfıra bölmə) olacaq. Xətanın ismarıcı - xətanı bir növ izah edir, əlavə təsviredici məlumat verir. 3. **Proqramın sonlanma kodu** - bizim yazdığımız kod terminalda və ya konsolda icra olunur. Proqramımız uğurla və ya uğursuz(qəza səbəbilə) sonlana bilər - başqa sözlə - çökə bilər. Uğurla sonlandıqda proqram terminala `0` kodu, əks təqdirdə isə `1` qaytarır. Ümumilikdə, indiyə qədər yazdığım nümunə kodlar daxilində kifayət qədər növ xətalar vardı. Onların bəzilərinə cədvəldə baxaq:

Xətanın adı	nə zaman ortaya çıxır
`ZeroDivisionError`	sıfıra bölmə zamanı
`IndexError`	ardıcıllığın mövcud olmayan indeksinə müraciyət etdikdə
`AssertionError`	`assert` şərti ödənmədikdə (bu bölmədə tanış olacağıq)
`AttributeError`	obyektin mövcud olmayan attributuna müraciyət etdikdə
`ImportError`	yanlış import zamanı
`KeyError`	lüğətin mövcud olmayan açarına müraciət zamanı
`NameError`	təyin olunmamış dəyişənin adından istifadə etməyə çalışdıqda
`TypeError`	funksiya və ya operator uyğun olmayan tipdə verilən aldıqda
`ValueError`	funksiya və ya arqument uyğun olmayan dəyər qəbul etdikdə

Son iki xəta olduqca oxşar gəlir ilk baxışdan. `TypeError` - tip xətasıdır, məsələn `float("[0.5]")` işləməyəcək, çünki `float` funksiyası sətr və ədəd tipindəki verilənləri kəsr ədədə çevirə bilir, yəni burda əsas problem tip oldu. Əgər `float("one")` kimi yazsaq, baxmayaraq ki tip cəhətdən sətr olması uyğundur, ancaq dəyər baxımından - söz olması kodun işləməməsinə səbəb olacaq. Əslində bu tip məsələlər praktikada çox az adamı düşündürür, amma yenə də müəyyən təsəvvürün olması pis deyil.

6.3. Xətaların tutulması

Proqram məntiqini yazdıqda maksimal dərəcədə xətalardan qaçmağa çalışırıq, funksiyadırsa - parametrləri yoxlayırıq, istifadəçidən hər hansı bir dəyər alırıqsa - onun dəyərini və sairə yoxlayırıq. Məsələn, sadə hesablayıcı proqramında əgər a və b ədədləri və hər hansı c əməmliyyatı varsa,

yoxlayırıq - c bölmədirsə, b=0 deyil ki... Çünki, bilirik ki, belə olan halda proqramımız çökür, çünki xəta baş verdikdə proqram dayanır. Ondansa, elə `print` ilə sadə və aydın ismarıc çap edib proqramı sonlandırmaq daha yaxşıdır. Ümumiyyətlə, istifadəyə verilmiş proqram heç bir halda çökməməlidir. Bəs buna necə nail olmaq olar - bütün mümkün parametr və giriş verilənlərini yoxlamaqla - çətin olsa da mümkündür, ancaq bu da tam çıxış yolu deyil. Çünki, problen heç də həmişə giriş verilənlərindən qaynaqlanmır. Şəbəkə bağlantısı tələb edən proqramda internet bağlantısının itməsi və ya böyük fayllar ilə işləyən proqramlarda əməli yaddaşın tükənməsi və sair ilaxır. Qısası - hər şeyi nəzarətdə saxlamaq olmur, üstəlik bu nəzarətə cəhd proqram kodunun oxunmasını, başa düşülməsini qəlizləşdirirsə, deməli bu qətiyyən çıxış yolu deyil. Əvəzində xətaları "tuta" bilərik. Təsəvvür edin ki, bir blok kod var və bu blok şəbəkədən asılıdır və bilirik ki burada şəbəkə xətası baş verə bilər. Belə olan halda, bu blok kodu "göz altında" - `try` içərisində yazırıq. `try` -dan sonra `except` bloku gəlməlidir - `try` içərisində hər hansı proqram blokunu icra etməyə çalışırıq, əgər bu blokda göstərəcəyimiz xəta baş verərsə, `except` içərisindəki blok icra olunacaq. Yəni bir növ `try` ilə sınayırıq, `except` -də isə xətanı tuturuq(baş verərsə). Sadə bölmə ilə nəzərdən keçirək:

```
def divide(n, m):
    try:    # cəhd et
        return n / m
    except ZeroDivisionError:   # ZeroDivisionError baş verərsə
        print("You are dividing by zero!! ")
        return None

print(divide(25, 5))    # 5.0
print(divide(25, 0))    # 'You are dividing by zero!! '
```

`divide` funksiyasını ilk dəfə düzgün arqumentlərlə çağırdıq və heç bir xəta baş verməd, ikinci çağırışda isə `ZeroDivisionError` baş verdi və bu xəta gözlənilən olduğu üçün `except` daxilindəki proqram bloku icra olunur. `except` önündə birdən çox xəta adı da göstərə bilərik, bunun üçün xəta adlarını yumru mötərizələr daxilində yazırıq və sadalanan xətaları tuturuq. Əgər `except` ilə tutulmayan xəta baş verərsə, bu zaman proqram çökür.

```
>>> try:
...     i = [1,2,3][99]
... except (ZeroDivisionError, ValueError):
...     print('error !!!')
...
   Traceback (most recent call last):
     File "<stdin>", line 2, in <module>
   IndexError: list index out of range
>>>
```

Yuxarıdakı nümunədə biz `except` önündə `ZeroDivisionError` və `ValueError` xətalarını qeyd etməklə, onları gözlədiyimizi bildirdik, lakin baş verən xəta - mövcud olmayan indeksi almağa cəhd etdikdə yaranan `IndexError` xətası oldu.

`finnaly` və `else`. Əslində, bu konstruksiya ancaq `try` və `except` ilə bitmir. Əlavə olaraq, `finally` və `else` də var.

- `try` - icra etməyə çalışdığımız proqram kodu
- `except` - gözlədiyimiz xəta və ya xətalar
- `finally` - istənilərn halda - xəta baş versə də, verməsə də icra olunacaq proqram kodu
- `else` - xəta baş verməzsə icra olunacaq proqram kodu Gördüyünüz kimi, `else` burda da qabağımıza çıxdı... `try` və `except` mütləq hissələrdir, çünki `try` ilə kodu icra etməyə çalışırıqsa, dəqiq xəta gözləntimiz də olmalıdır. Ancaq, `finally` və `else` optimaldırlar, istifadə edə də bilərik, etməyə də. `finally` daxilində istənilən halda icrası lazım olan, vacib olan əməliyyatlar olur. Məsələn, açıq olan olan bağlantını bağlamaq, hansısa faylları silmək və sairə. Bu nöqteyi nəzərdən, tətbiq sahələri zaman və təcrübə ilə özləri gəlirlər. Növbəti bölmədə fayllar haqqında danışdıqda, try-except-else-finally "dörd dostu" bir daha yad edəcəm.

Hər xəta üçün ayrıca gözlənti. Yuxarıdakı nümunədə iki xəta gözləntimiz vardı və xəta baş verərkən tək `except` bloku icra olunurdu. Bəs əgər hər bir xəta üçün ayrıca icra olunmalı olan proqram kodu varsa, əgər hər bir xəta üçün ayrıca `except` tələb olunarsa, o zaman necə davranmalı? Belə olan halda `except: <xəta_adı>` alt-alta sadalayırıq vəssalam, bir növ alt-alta `elif` -lər kimi. Baxaq:

```
>>> def div(n, m):
...     try:
...          return n / m
...     except ZeroDivisionError:
...             print("zeeero!")
...     except TypeError:
...             print("be carefull, type!")
...     else:
...             print("cool! no errors!")
...     finally:
...             print("i am always here...")
...
>>> div(10, 0)     # ZeroDivisionError
zeeero!
i am always here...
>>> div(10, 'hi')   # TypeError
be carefull, type!
i am always here...
>>> div(10, 2)      #  No Errors
i am always here...
5.0
```

Əsas xəta Müxtəlif adlı xətalar gördük, bu xətaların əcdadı - *əsas xəta* və ya *ümumi xəta* var : `Exception` . Əgər bu xətanı except ilə gözləntiyə alsaq, ixtiyari xəta baş verərsə except-in `Exception` blokunda tutulacaq. Ancaq, nəzərə alsaq ki, kod yuxarıdan-aşağı icra olunur, o zaman bir neçə except olarsa və yuxarıda duran except-lərdən hansısa biri işə düşərsə - sonra gələn `except Exception` işə düşməyəcək. Adətən, əvvəlcə konkret xətaları tutmağa çalışırlar və yalnız ən son halda ümumi `Exception` qeyd olunur. Çünki, əgər `except Exception` ilk olaraq qeyd olunsa, ixtitarı xəta baş verdikdə işə düşmüş olacaq, bu isə o deməkdir ki proqram məntiqimizi o qədər də yaxşı idarə etmirik və bütün istisna halları sadəcə bir nöqtəyə ataraq "yola veririk". Burada artıq hər şey təcrübə və yanaşmadan ibarətdir:

```
>>> try:
...     print(10 / 0)    # ZeroDivisionError
... except Exception:    # ümumi xəta
...     print("code crashed")
... except ZeroDivisionError:
...     print("You are trying to divide to 0 ")
...
code crashed
```

Gördüyünüz kimi, ilk dayanan ümumi xəta gözləntisi olduğu üçün, o işə düşür və konkret bizim xətamızı təsvir edən `ZeroDivisionError` gözləntisi "gözdən qaçır". Odur ki, ümumi gözləntini istifadə edirsinizsə belə, sonda qeyd etmək lazımdır.

6.4. Xətaların atılması

İlk öncə, "xətaların tutulması" necə qəribə səslənirdisə, "xətaların atılması" da bir o qədər qəribə səslənir. Xətanı atmaq - proqramda həmin xətanın baş verməsini təmin etməkdir. Bunu üçün `raise` açar sözündən istifadə edilir, yazılışı belədir: `raise <xəta_adı>`. İnsanın doğum ilinə əsasən yaşını təyin edən funksiya yazaq, doğum ili 1900-dən kiçik olarsa xəta ataq:

```
>>> def calc_age(birth_date: int)->int:
...     if birth_date < 1900:
...         raise ValueError("You are a liar old man!")
...     else:
...         return 2023 - birth_date
...
>>> calc_age(1998)
25
>>> calc_age(1898)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in calc_age
ValueError: You are a liar old man!
>>>
```

Yuxarıda diqqət edilməli bir neçə məqam var:

1. `birth_date < 1900` şərti ödəndikdə xəta atıram və proqram bu xəta ilə sonlanır. Belə bir sual ortaya çıxa bilər - xətanı özüm ataraq, tuta bilərik? Yəni bir tərəfdən `raise ValueError` deyib, digər tərəfdən də `except ValueError` desək işləyərmi - bəli, işləyər.
2. Yəqin ki fikir verdiniz, sadəcə olaraq `raise ValueError` yazmadım, `ValueError` obyektini çağıraraq, ona xəta ismarıcı ötürdüm. Bununla da xəta atılarkən bizim təyin etdiyimiz ismarıcla atılır.

6.5. Assert ilə yoxlama nöqtələrinin qoyulması

raise operatoruna bəznzər daha bir operator var, bu operator da xəta atır ancaq müəyyən şərt daxilində. Söhbət `assert` operatorundan gedir. Bu operator müəyyən bir məntiqi ifadə alır və əgər bu ifadə doğru **olmazsa**, xəta baş verir, özü də konkret `AssertionError` - yəni assert ilə öz istədiyimiz xətanı ata bilmirik. Sintaksisi belədir:

```
assert <şərt>, <xəta ismarıcı>
```

İlk öncə shell-də bir neçə sadə nümunəyə baxaq:

```
>>> assert 10 == 10
>>> assert 11 == 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
>>> assert 11 == 10, "upps!!"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: upps!!
>>> email="email@mail.com"
>>> assert '@' in email, 'bad email!'
>>> email2="email.mail.com"
>>> assert '@' in email2, 'bad email!'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: bad email!
```

Assert operatorunun əsas təyinatı - bir növ qoruyucu, yoxlayıcı nöqtə rolunu oynamaqdır. Deyək ki, proqram məntiqi yazılarkən, əminik ki bu sətrdə hər hansı `n` dəyişəninin qiyməti müsbət bir ədəd olacaqdır. Ancaq yenə də hər ehtimala qarşı bunu yoxlayacaq şərt ilə bir assert yaza bilərik. Beləliklə, hər hansı səbəbdən parametr bizim gözləmədiyimiz qiyməti alarsa, proqram ortaya `AssertionError` xətasını atacaq. Məqsəd nədir - proqram daxilində baş verəcək qeyri-müəyyənlikləri, sürprizləri aradan qaldırmaqdır. Çünki, adətən yazdığımız proqramı kiminsə üçün, müəyyən sifarişçi üçün yazırıq və hər hansı `n` parametrinə görə proqramımız gizli şəkildə səhv işləsə, bu heç də yaxşı olmaz. Proqramın elə bizim əlimizdə çökməsi daha yaxşıdır, çünki səhvi tapıb aradan qaldıra bilərik, amma yox, kod bizim əlimizdə ikən, bütün səhvləri aşkar etməsək, sonra beyin məhsulumuz olan kodu sifarişçiyə təhvil verdikdən sonra ortaya çıxacaq problemlər heç yaxşı hisslər yaşatmayacaq. Odur ki, istənilən "dumanlı" yerlərdə yoxlamalar qoymaq lazımdır. İndi isə, nəhayət nümunəyə keçək:

```
from typing import Union

def to_meter(value: Union[int, float], unit: str):
    # 'unit' kəmiyyəti və 1 unitin metrlərlə qiyməti
    valid_units = {'km': 1000, 'dm': .1, 'mm': .001, 'm': 1,
                   'mile': 1609.344, 'yard': 0.9144, 'foot': 0.3048, 'inch':
0.0254}

    assert value > 0, "length must be a positive number."  # assert ilə yoxlama
    if not unit in valid_units.keys():  # adi şərt ilə yoxlama
        raise ValueError(f"unsupported unit - '{unit}'!\n" +
                         f"Possible units are: {list(valid_units.keys())}")

    one_in_m = valid_units[unit]    # 'one_in_m' - kəmiyyətin vahid metrdə
dəyəri
    value_in_m = value * one_in_m
```

```
    return value_in_m


print(to_meter(100, 'km'))  # 100000
print(to_meter(100, 'dm'))  # 10.0
print(to_meter(10, 'yard'))  # 9.144
```

shell mühitində davam etsək:

```
>>> to_meter(-10, 'km')  # assert xətası
    AssertionError: length must be a positive number.
>>>
>>> to_meter(10, 'kg') # atdığımız xəta
    ValueError: unsupported unit - 'kg'!
    Possible units are: ['km', 'dm', 'mm', 'm', 'mile', 'yard', 'foot', 'inch']
```

Sual yarana bilər - bəs nə vaxt if-else, nə vaxt da `assert` istifadə etməliyəm? `assert` daha çox kodun test edilməsi - kodun hansı hallarda, özünü necə aparması və debaq(ing *debug*) - kodda aşkarlanmış xətanın və ya nöqsanın səbəbinin araşdırılması zamanı istifadə edilir. Başqa sözlə, proqramçı olaraq özümüz kodu incələyəndə istifadə edirik. Hər bir assert - kodda gizlənmiş mənfur nöqsanı tutmaq üçün bir tələdir. Odur ki, sadəcə mümkün xətaların qarşısını almaq üçün ya `if-else` ilə verilənləri yoxlayırıq, ya da `try-except` ilə gözlədiyimiz xətaları tutub emal edirik.

Tapşırıqlar

1. Təhlükəsiz şəkildə sətrləri ədədlərə çevirəcək `str_to_num` funksiyası yazın. Funksiya ötürülmüş sətri ədədə çevirməlidir. Əgər ədədə çevrilmə zamanı xəta baş verərsə, geriyə `None` qaytarılsın.

```
def str_to_num(string_sum: str)-> float|None:
    pass

str_to_num("3.1415")   # 3.1415
str_to_num("-3")   # -3
str_to_num("3a")   # None
```

2. Düzbucaqlının sahəsini hesablayacaq `rectangle_area` funksiyasını yazın. Funksiya - dördbucaqlının eni və uzunluğunu ifadə edən `length` və `width` parametrləri almalıdır. Sadə `assert` yoxlamaları ilə əmin olun ki bu parametrlər müsbət ədədlərdir.

```
def rectangle_area(length: float, width: float)->float:
    # Your implementation here
```

3. Istifadəçi parolunu yoxlamaq - validasiya etmək üçün `validate_password` funksiyasını yazın. Funksiya özündən parol ehtiva edən sətr tipli `password` arqumentini alır və aşağıdakı şərtlərdən ən azı biri ödənməsə "Invalid password!" ismarıclı `ValueError` xətası atmalıdır, hər şey qaydasındadırsa - `True` qaytarmalıdır.
   - Ən azı 8 simvol olmalır;
   - boşluq simvolları olmamalıdır;
   - İçərisində həm hərflər, həm də rəqəmlər olmalıdır;

```
def validate_password(password: str)->bool:
    pass

validate_password("12345678pass")  # True
validate_password("12345678")  # ValueError
validate_password(" 12345678pass")  # ValueError
```

4. Yuxarıdakı `validate_password` funksiyasını elə dəyişin ki parol yoxlamanı keçmədiyi təqdirdə xəta atmaq əvəzinə geriyə `False` qaytarılsın. Təsəvvür edin ki belə bir istifadəçilər bazamız var:

```
users_dict = [
    {"name": "Rəşad", "email": "reshad@school.edu", "pass": "12345678pass"},
    {"name": "Əziz", "email": "eziz@school.edu", "pass": "pass"},
    {"name": "Ülvi", "email": "ulvi@school.edu", "pass": "123pass"},
    {"name": "Hikmət", "email": "hikmet@school.edu"},
]
```

Istifadəçilərin parollarını yoxlamaq üçün `validate_users_pass` adlı funksiya yazın. Funksiya bu arqumentləri almalıdır:

- `users: dict` - yuxarıdakı nümunəyə bənzər istifadəçilər lüğəti

- `pass_field_name: str="pass"` - lüğətdə özündə istifadəçi parolunu saxlamalı olan açar, susmaya görə `"pass"` olacaq.

`validate_users_pass` funksiyası bütün istifadəçilərin parollarını yoxladıqdan sonra geriyə lüğət qaytarmalıdır: lüğətin açarları - yoxlanılan istifadəçinin(siyahıdakı lüğətin) indeksi, dəyərləri isə `validate_password` funksiyasının bu istifadəçinin parolu üçün nəticəsi olacaq. Onu da nəzərə alın ki, bəzi istifadəçilərin parolları olmaya da bilər. Belə olan halda, həmin istifadəçi üçün nəticə `None` olacaq.

```
def validate_password(password: str)->bool:
    pass

def validate_users_pass(users: list[dict],
                        pass_field_name: str="pass")-> dict[int, bool]:
    pass

res = validate_users_pass(users_dict)
print(res)  # {0: True, 1: False, 2: False, 3: None}
```

Link: https://github.com/AzizNadirov/python-road-book/blob/master/code/tasks/exceptions_6.ipynb

7.0. Fayllar haqqında bir az

Fayllar.

Şübhəsiz ki informasiyanı uzun müddətli saxlamağın ən geniş yayılmış üsulu - informasiyanı fayl kimi saxlamaqdır. Gündəlik həyatımızda müxtəlif tip fayllarla qarşılaşırsınınz - musiqi faylları, şəkillər, videolar və sairə. Bütün bu faylları birləşdirən əsas məqam - onların bayt ardıcıllığı olmasıdır. Yəni, bütün fayllar əslində baytlardan(yadınızdadırsa, 1 bayt=8 bit, 1 bit isə, bir dənə 0 və ya 1 demək idi) ibarətdir, hər bir faylın sonunda - onun genişlənməsi(ing. *extention*) olur. Faylın genişlənməsi - faylın tipini göstərir. genişlənmə əsasında əməliyyat sistemi - faylı nə ilə, hansı proqram ilə açmalı olduğunu başa düşür. Məsələn, `.pdf` genişlənməsi elektron sənədlərə aid olduğu üçün, siz bu faylı klikləyərək açmağa çalışdıqda, ƏS bu tip faylları açmaq üçün nəzərdə tutulmuş proqramlardan istifadə edir. Fayllar əməli yaddaş qurğusunda(RAM-da) deyil, sətr diskdə(ing. hard drive) yerləşdiyi üçün, sərt diski zədələməsək, ordan itəsi deyil.

Faylın yolu, müxtəlif əməliyyat sistemlərində yollar

Fayllar sərt diskdə necə gəldi yox, qovluqlarda(ing *directory* və ya *folder*) yerləşir və hər birinin öz ünvanı - yolu(ing *path*) var. Faylın yolu - faylın harada - hansı qovluqda yerləşdiyini bildirir. Windows ƏS-də yolun əvvəlində sərt diski ifadə edən hərf daha sonra faylın yolu qeyd olunur:

```
D:\my_books\python ilə proqramlaşdırma.pdf
```

Yuxarıdakı nümunədə fayl - `D` diskində `my_books` qovluğunda yerləşir, adı `python ilə proqramlaşdırma`, genişlənməsi isə - `pdf` -dir. Gördüyünüz kimi, Windowsda yolu təşkil edən hissələr `\` simvolu ilə ayrılır. Ancaq heç də bütün dünya windows üzərində fırlanmır, başqa bir əməliyyat sistemləri ailəsi olan UNİX ailəsində yollar tamamilə fərqlənir, burada yolun əvvəlində diskin adı deyil, ƏS-nin içərisində yerləşdiyi disk olur. Məsələn, təsəvvür edin ki içərisində SSD tipli sərt disk olar noutbukunuz var və bu sərt diskə Linux ailəsinə mənsub ƏS yazılıb. ƏS yazılan disk özü qovluqlara bölünür, içərisində `bin`, `boot`, `etc`, `home`, `mnt` və sairə adlı qovluqlar olur. ƏS demək olar ki bu qovluqlara səpələnmiş fayllardan ibarətdir. Kənardan qoşulan disklər isə, windows ƏS-dəki kimi fayl sisteminin kökündə deyil, hər hansı qovluğa bağlanır, Linuxda bu qovluq adətən `/mnt` olur. Windows ƏS-də istifadəçinin "My Documents" qovluğunun ünvanı belə ola bilər: `C:\Users\Aziz\Documents`, Linux üçün isə bu belə olar: `/home/Aziz/Documents`. Burada bizi maraqlandıran əməliyyat sistemləri deyil, onların fayl yollarındakı fərqlərdir. Diqqətlə baxsaq, görmək olar ki Windows üçün ayırıcı simvol kimi `\` istifadə edilir, UNIX-də isə `/`. Buna görə də müxtəlif əməliyyat sistemlərində yollar fərqli görünür.

Tam (mütləq) yol və nəzəri yol

Tam yol(absolute path) - özündən fayl sisteminin kökündən başlayıb, lazımi fayla və ya qovluğa qədər olan hissəni ehtiva edir. Məsələn, `C:\Users\Aziz\Documents\LTA\docs\table.xlsx` yolunda `table` faylının tam yolu verilib, çünki faylın yolu fayl sisteminin lap əvvəlindən(kökündən) başlayır. İndi isə, təsəvvür edin ki biz `C:\Users\Aziz\Documents\LTA` qovluğunun içərisindəyik və bu qovluğun içərisində `docs` qovluğundan əlavə, `images` qovluğu da var, hansı ki öz işərisində `logo.jpg` adlı fayl saxlayır. Əgər biz özümüz `LTA` qovluğunun içərisindəyiksə, elə həmin qovluqda olan faylın yolunu göstərmək üçün fayl sisteminin kökünə qayıtmağa ehtiyac yoxdur, elə içərisində olduğumuz `LTA` qovluğundan başlayıb yazmaq olar, başqa sözlə, faylın yolunu içərisində

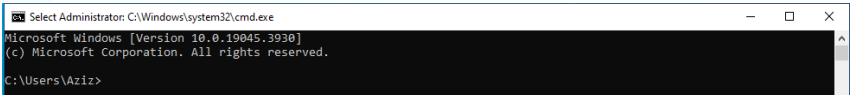
yerləşdiyimiz qovluğa "nəzərən" yaza bilərik. Elə olan halda, `logo.jpg` faylının yolunu `C:\Users\Aziz\Documents\LTA\images\logo.jpg` əvəzinə `images\logo.jpg` kimi yazmaq olar, çünki zatən `C:\Users\Aziz\Documents\LTA` içərisindəyik. Bu cür, hansısa qovluğa nəzərən göstərilmiş yollar *nəzəri yollar*(relative path) adlanır. O zaman, `LTA` qovluğuna nəzərən `table.xlsx` , faylının yolu - `docs\table.xlsx` olacaq.

Konkret bir qovluğun içərisindən işləyən zaman nəzəri yollar olsuqca rahatdırlar. Ancaq, tam yollar faylın dəqiq, birmənalı yolunu ifadə edir. Əgər mən sizə kompüteri verib desəm ki `dir_a\dir_b\pic.jpg` şəklini tapın, çətin olacaq, çünki yol tam yol deyil, `dir_a` qovluğunun harada olması barədə heç bir təsəvvür yoxdur. Ancaq `C:\Users\Aziz\Documents\LTA\dir_a\dir_b\pic.jpg` kimi tam yolu vermiş olsaydım, birmənalı şəkildə faylın harada olduğu görünür.

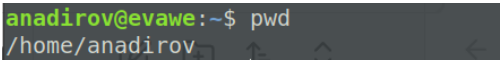
Unix tipli ƏS-lərində yol və fayllara aid bəzi əmrlər

Unix tipli sistemlərdə kompüterin demək olar ki bütün "nöqtələrinə" terminaldan - xüsusi əmrlər vasitəsilə çatmaq mümkündür. Windows və Mac istifadəçiləri adətən GUI(qrafik istifadəçi interfeysi-qrafik komponentlərdən-düymələrdən və müxtəlif pəncərələrdən ibarət olan interfeys) istifadə edirlər. Düzdür, Mac da Unix tipli ƏS üzərində qurulduğu üçün terminala ordan da əl atmaq olar, ancaq Windows istifadəçilərində elə bir imkan yoxdur. Windows-da adətən cmd(command line) və ya power shell idtifadə etmək olar, amma bütün bunlar funksionallıqda Unix terminala çata bilmir. Bu kiçik bölmədə fayllarla işləmək üçün bizə lazım ola biləcək bəzi əmrlər Windows və Unix sistemləri üçün göstərilib. Bu bilgilər kod mühitində fayllarla daha rahat və əminliklə işləməyə imkan verəcək, odur ki başlayaq.

- **terminalı işə salmaq**: əksər linux distribütivlərində terminalı işə salmaq üçün `ctrl+alt+T` qısayolundan istifadə etmək olar. Əgər yoxdursa, axtarışa "terminal" yazaraq tapıb işə salmaq olar. Windows üçün isə, cmd işə salmaq üçün `Win+R` , ortaya çıxan pəncərə `cmd` yazaraq enter sıxmaq olar.
- **ev qovluğu**: hər bir ƏS-də istifadəçinin əsas qovluğu - ev qovluğu(home directory) var. Bu qovluq istifadəçinin tələbləri üçün nəzərdə tutulur - sənədləri, şəkilləri saxlamaq və sairə üçün. Windows üçün istifadəçinin ev qovluğu - `C:\Users\Aziz` kimi görünür, mənim istifadəçi adım `Aziz` olduğu üçün qovluq da bu cür adlanır:

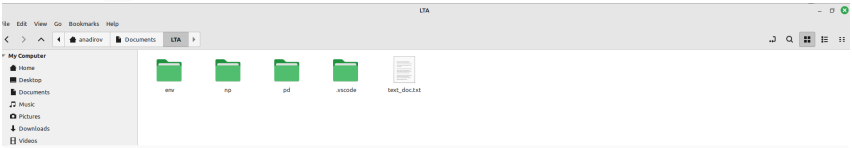
  Linuxda isə bu qovluq `/home/anadirov` olacaq. Burda isə `anadirov` istifadəçi adıdır. Gördüyünüz kimi, Windows ayırıcı simvol kimi `\` , Linux isə `/` istifadə edir.
- **cari qovluğu almaq**: terminalda işləyərkən əmrlər vasitəsilə fayl sistemi üzərində əməliyyatlar apara bilərik - köçürmək, silmək, icazələri dəyişmək və s. Bunun üçün də terminal (və ya cmd, bundan sonra sadəcə terminal deyəcəm) fayl sistemini görə bilməlidir. Terminal işə salınarkın istifadəçinn ev qovluğundan başlayır. Terminal içərisindən hansı qovluq içərisində olduğumuzu öyrənmək üçün, windows üçün `cd` - (current directory - cari qovluq), linuxda isə `pwd` - (print working directory - işçi qovluğu çap et) əmrindən istifadə edilir.



- **cari qovluğu dəyişmək**: bu əmr terminalda bir qovluqdan - başqa qovluğa keçməyə imkan verir. Windows üçün `chdir` - (change directory - qovluğu dəyiş), linux üçün isə `cd` - (**C**hange **D**irectory) istifadə edilir. Əmrin ardıyca keçmək istədiyimiz ünvanın tam və ya nəzəri yolunu yazırıq:

```
anadirov@evawe:~$ cd /home/anadirov/Documents/LTA/
anadirov@evawe:~/Documents/LTA$ pwd
    /home/anadirov/Documents/LTA
```

- **qovluğun məzmununa baxmaq**: qovluğun içərisindəkiləri görmək üçün Windowsda `dirs`, linuxda isə `ls` əmrindən istifadə edilir.




```
anadirov@evawe:~/Documents/LTA$ ls
    env  np  pd  text_doc.txt
```

Gördüyündüz kimi, `.vscode` qovluğu niyəsə çap edilmədi. Məsələ burasındadır ki, nöqtə ilə başlayan fayl və qovluqlar "gizli" hesab edilir və adətən istifadəçilər üçün görünməz olur. Əgər `ls` ilə gizli faylları da həmçinin görmək istəyirsinizsə, `-a` bayrağını da əlavə etməliyik:

```
anadirov@evawe:~/Documents/LTA$ ls -a
    .  ..  env  np  pd  text_doc.txt  .vscode
```

- **faylın məzmununu çap etmək**: mətn faylının məzmununu çap etmək üçün `cat` əmrindən istifadə edilir.

```
anadirov@evawe:~/Documents/LTA$ cat text_doc.txt
    El bilir ki sən mənimsən,
    Yurdum, yuvam, məskənimsən,
    Anam, doğma vətənimsən,
    Ayrılarmı könül candan?
    Azərbaycan! Azərbaycan!
```

- **yeni fayl yaratmaq**: yeni fayl yaratmaq üçün `touch` əmrindən istifadə etmək olar:

```
anadirov@evawe:~/Documents/LTA$ touch azerbaycan.txt
anadirov@evawe:~/Documents/LTA$ cat azerbaycan.txt
anadirov@evawe:~/Documents/LTA$
```

Boş `azerbaycan.txt` faylı yaratdıq. `cat` ilə məzmuna baxmağa çalışdıq, amma fayl boş olduğu üçün heç nə görmədik.

- **faylın içərisinə yazmaq**: faylın içərisinə hər hansı məzmun yazmaq üçün `>` və ya `>>` istifadə olunur. `>` - faylın içərisindəkiləri silərək yeni məzmunu yazır(overwrite modu), `>>` isə köhnə

məzmunun ardıyca yazır(append modu). Faylın içərisinə sətr yazmağa çalışaq, bunun üçün həm də `echo` əmrindən istifadə edəcəyik, bu əmr bir növ linuxun `print`-idir və çap etmək üçün istifadə edilir. Yəni biz sətri bir növ çap edib, `echo` əmrinin qaytaracağı sətri fayla yazacağıq:

```
    anadirov@evawe:~/Documents/LTA$ echo "hello Azerbaijan!" > azerbaycan.txt
    anadirov@evawe:~/Documents/LTA$ cat azerbaycan.txt
        hello Azerbaijan!
    anadirov@evawe:~/Documents/LTA$ echo "hello Baku!" >> azerbaycan.txt
    anadirov@evawe:~/Documents/LTA$ cat azerbaycan.txt
        hello Azerbaijan!
        hello Baku!
```

Yazılacaq məzmun əvəzinə sətr yox, başqa bir əmrin nəticəsini də istifadə etmək olar. Məsələn, `cat text_doc.txt` əmri bizə bir kuplet şeirimizi qaytarırdı, biz həmin mətni `azerbaycan.txt` faylına yaza bilərik:

```
anadirov@evawe:~/Documents/LTA$ cat text_doc.txt
    El bilir ki sən mənimsən,
    Yurdum, yuvam, məskənimsən,
    Anam, doğma vətənimsən,
    Ayrılarmı könül candan?
    Azərbaycan! Azərbaycan!
anadirov@evawe:~/Documents/LTA$ cat text_doc.txt > azerbaycan.txt
anadirov@evawe:~/Documents/LTA$ cat azerbaycan.txt
    El bilir ki sən mənimsən,
    Yurdum, yuvam, məskənimsən,
    Anam, doğma vətənimsən,
    Ayrılarmı könül candan?
    Azərbaycan! Azərbaycan!
```

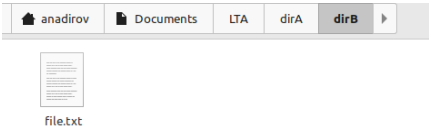
- **silmək**: faylı silmək üçün `rm`, boş qovluğu silmək üçün isə `rmdir` əmrindən istifadə edilir:

```
anadirov@evawe:~/Documents/LTA$ rm azerbaycan.txt
anadirov@evawe:~/Documents/LTA$ ls
    env  np  pd  text_doc.txt
```

İndi isə boş qovluq yaradıb silək:

```
anadirov@evawe:~/Documents/LTA$ mkdir some_dir
anadirov@evawe:~/Documents/LTA$ ls
    env  np  pd  some_dir  text_doc.txt
anadirov@evawe:~/Documents/LTA$ rmdir some_dir/
anadirov@evawe:~/Documents/LTA$ ls
    env  np  pd  text_doc.txt
```

İndi isə, təsəvvür edin ki `LTA` içərisində `dirA/dirB/file.txt` var. Yəni, belə bir şey:



`rmdir` ilə `dirA` qovluğunu silməyə çalışsaq xəta çıxacaq, çünki qovluq boş deyil:

```
anadirov@evawe:~/Documents/LTA$ rmdir dirA/
    rmdir: failed to remove 'dirA/': Directory not empty
```

Belə olan halda `rm` əmrini `-r` və `-f` bayraqları ilə istifadə etmək olar və ya hər ikisini birləşdirib `-rf` kimi yazmaqla:

```
anadirov@evawe:~/Documents/LTA$ rm -rf dirA/
anadirov@evawe:~/Documents/LTA$ ls
    env  np  pd  text_doc.txt
```

- **faylın nüsxəsinin köçürülməsi(copy)**: fayl və ya qovluqların nüsxələrini köçürmək üçün `cp` əmrindən istifadə edilir. Ümumi sintaksisi: `cp <bunu> <bura>` . Yəni ilk arqument köçürmək istədiyimiz "şey", ikinci isə hara köçürmək istədiyimizdir. Deyək ki, iki ədəd qovluq var: `LTA/A` və `LTA/B` , `A` içərisində `file.txt` faylı var, bu faylın nüsxəsini `B` qovluğuna köçürmək istəyirik. Belə olan təqdirdə əmr belə olacaq(təsəvvür edin ki `LTA` içərisindəyik):

```
cp A/file.txt B/
```

Faylı başqa adla da saxlamaq olar:

```
cp A/file.txt B/file_2.txt
```

Qovluğun nüsxəsini köçürmək üçün isə `-r` bayrağı əlavə edirik. `A` qovluğunun nüsxəsini `B` qovluğuna köçürmək üçün:

```
cp -r A B/
```

- **faylın köçürülməsi**(move): faylı köçürmək üçün `mv` əmrindən istifadə edilir və istifadə qaydası `cp` ilə eynidir.

Linux fayl sisteminin müzakirəsi kitabın çərçivələrindən kənara çıxır və kitabın ümumi məzmununu yükləmiş olur. Odur ki, əmrlər barədə hələ ki bu qədər yetər.

> **Maraqlı**
>
> Terminalda istənilən əmr haqqında məlumat almaq olar. Bunun üçün əmrin qarşısına `--help` ötürmək lazımdır:

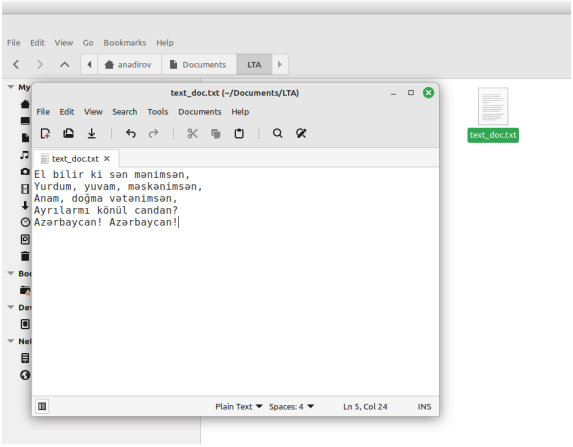
```
cp --help
   Usage: cp [OPTION]... [-T] SOURCE DEST
or:  cp [OPTION]... SOURCE... DIRECTORY
or:  cp [OPTION]... -t DIRECTORY SOURCE...
Copy SOURCE to DEST, or multiple SOURCE(s) to DIRECTORY.
```

7.1. Pythonda Fayllar

Maraqlı

Bu bölmədə biz fayllara yazmaq və onları oxumaq haqqında danışacağıq. Fayllarla, onlsrın yolları ilə daha effektiv işləmək üçün pythonda `os`, `pathlib` və `shutil` kimi modullar var. Kitabın 3-cü bölməsində standart kitabxana modulları haqında danışarkən, bu modullara da toxunacağıq. Ən bəsid fayllar - mətn fayllarıdır. Mətn fayllarını adi notepad ilə açıb redaktə etmək olar, pythonda həmçinin onlarla işləməkçün hər hansı əlavə "hərəkətlərə" ehtiyac yoxdur. Odur ki, bölmə ərzində məhz mətn faylları ilə işləyəcəyik.

Pythonda faylları açmaq üçün `open` funksiyasından istifadə edilir. Funksiyanın ilk parametri - açmaq istədiyimiz faylın yoludur. Funksiya geriyə `TextIOWrapper` tipli obyekt qaytarır, bu obyekt fiziki fayl ilə onun pythonda açılmış təsviri arasında bir bağlantıdır - **axındır**(ing stream). Biz fayla dəyişiklikləri bu axın vasitəsilə aparırıq, etdiyimiz dəyişikliklər axına, ordan da fiziki fayla tətbiq olunur. `open` funksiyasının ikinci parametri - faylı oxuma modudur. Faylın **oxunma modu** - *faylı nə kimi*(baytlar ya mətn) və hansı səlahiyyətlər(ancaq oxu, ancaq yazı və ya qarışıq) ilə oxunmalı olduğunu təyin edir. Ümumiyyətlə, qeyri-mətn faylları adətən bayt olaraq açırıq. Fayllarla işləyərkən diqqət etməli olduğumuz başqa bir məsələ - sonda axını müvəfəqiyyətlə bağlamağımızdır. Məsələ burasındadır ki, fiziki fayl ilə proqram arasında körpü rolunu oynayan axını sonda özümüz bağlamalıyıq, əks təqdirdə faylı zədələyə bilərik və ya etdiyimiz dəyişikliklər heç də hamısı yazılmamış ola bilər. Əslində, bir alternativ yolumuz da var ki bu yolla fayl özü avtomatik olaraq bağlanır, amma bu barədə az sonra. Gəlin, ilk öncə mövcud olan adi mətn faylını açıb, içərisindəki mətni çap etməyə çalışaq. Mətn faylının məzmunu aşağıdakı şəkildəki kimidir:



Fayl `/home/anadirov/Documents/LTA` qovluğunda yerləşir və `text_doc.txt` adlanır. Gəlin bu faylı oxu modunda açıb içərisindəki mətni oxuyaq:

```
file_path = "/home/anadirov/Documents/LTA/text_doc.txt"

file_stream = open(file_path, mode='r')
content = file_stream.read()
print(content)
file_stream.close()
```

```
El bilir ki sən mənimsən,
Yurdum, yuvam, məskənimsən,
Anam, doğma vətənimsən,
Ayrılarmı könül candan?
Azərbaycan! Azərbaycan!
```

Nə baş verdi: ilk öncə faylın mütləq yolunu `file_path` dəyişəninə mənimsədirik. Daha sonra faylın yolunu `open` funksiyasına ötürürük, `mode='r'` parametri ilə faylı oxu modunda açırıq. Bu o deməkdir ki biz bu faylı ancaq oxuya bilərik, yaza bilmərik. `open` bizə fayla bağlı axın qaytarır, hansını ki `file_stream` dəyişəninə mənimsədirik. Bu axın obyektinin `read` metodu - faylın məzmununu qaytarır və biz bu məzmunu `content` dəyişəninə mənimsədirik, daha sonra çap edirik. Sonda isə, axın obyektinin `close` metodu ilə axını bağlayırıq. Davam etməzdən əvvəl gəlin fayl açılma modlarını sadalayaq:

mod	təyinatı
`r`	oxumaq
`w`	yazmaq
`rb`	bayt kimi oxumaq
`wb`	bayt yazmaq
`r+`	oxumaq-yazmaq

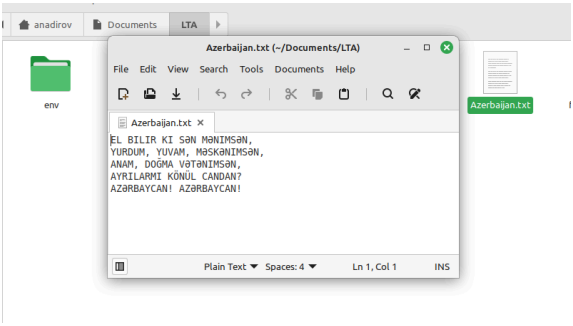
Oxu modunda açdığımız fayl mütləq mövcud olmalıdır, əks təqdirdə `FileNotFoundError` xətası alacağıq. Yazı modunda açılan fayl üçünsə belə deyil, mövcud olmayan faylı göstərmək olar, belə olan təqdirdə fayl yaradılacaq və içərisinə yazacağımız məzmun yazılacaq. Gəlin, yenidən şeirimizi açıb, `Azerbaijan.txt` faylına yazaq. Üstəlik ortaya bir az "action" qatmaq üçün şeirin bütün hərflərini böyük yazaq:

```
file_path = "/home/anadirov/Documents/LTA/text_doc.txt"

file_stream = open(file_path, mode='r')
new_file_stream = open("/home/anadirov/Documents/LTA/Azerbaijan.txt", mode='w')

content = file_stream.read()
new_file_stream.write(content)
```

```
file_stream.close()
new_file_stream.close()
```



Gördüyünüz kimi, ilk öncə lazımi faylları açırırq, daha sonra oxu-yazı əməliyyatlarını aparırıq və sonda faylları bağlayırıq. Ancaq, burda bir "əmma" var - işdir, əgər oxu-yazı mərhələsində nəsə xəta baş verərsə, onda nə baş vercəcək? Xəta baş verən zaman proqram icrası dayandığı üçün, faylların bağlandığı mərhələ icra olunmayacaq və bu o deməkdir ki, həmin anadək fayllara edilən dəyişikliklər itə bilər və yaxud ümumiyyətlə, fayl zədələnə bilər. Ona görə də, xəta qaldıra biləcək riskli hissəni `try` blokuna daxil edib, faylların bağlanmasını `finally` blokuna yazmaq olar:

```
try:
    file_path = "/home/anadirov/Documents/LTA/text_doc.txt"

    file_stream = open(file_path, mode='r')
    new_file_stream = open("/home/anadirov/Documents/LTA/Azerbaijan.txt",
mode='w')

    content = file_stream.read()
    new_file_stream.write(content)
finally:
    file_stream.close()
    new_file_stream.close()
```

Beləliklə, artıq istənilən halda fayllarımız sonda bağlanmış olur.

`with` kontekst meneceri ilə.

Hər dəfə fayllar ilə işləyərkən try-finally qurmaq o qədər də rahat deyil və düşünə bilərsiniz ki bunun hansısa bir alternatvi olmalıdır və deyim ki var. Bunun üçün kontekst menecer(context manager) və ya məzmun meneceri istifadə oluna bilər. Faylı bu yolla açdıqda əməliyyatın sonunda fayl avtomatik bağlanır və try-finally yazmağa ehtiyac qalmır. Yuxarıda istifadə olunmuş `Azerbaijan.txt` faylını belə oxuya bilərik:

```
with open("/home/anadirov/Documents/LTA/Azerbaijan.txt") as sheir:
    print(sheir.read())
```

Özü-özlüyündə kontekst menecer sadəcə abyekt üçün `with` üzərindən əlavə funksionallıq qatır, bir növ dekoratorlar kimi, bu haqqda OOP bəhsində danışacağıq. İndi isə bunu bilmək kifayətdir: faylı

`with` ilə bu cür açmaq olar `with open(<fayl_yolu>) as var_name: ...`, `var_name` faylın menecer içərisindəki adı olacaq. Bu bizə onu verir ki, `open` funksiyasının geriyə qaytardığı obyekt üçün kontekst menecerin sonunda avtomatik olaraq `.close()` çağrılır. Qeyd etmək lazımdır ki, bu halda menecerin kodu `with` blokunun içərisində yazılmalıdır. Məsələn, öncə try-finally ilə yazdığımız kodu belə yazmaq olardı:

```
file_path = "/home/anadirov/Documents/LTA/text_doc.txt"
with open(file_path, mode='r') as file_stream:
    with open("/home/anadirov/Documents/LTA/Azerbaijan.txt", mode='w') as
new_file_stream:
        content = file_stream.read()
        new_file_stream.write(content)
```

Tapşırıqlar

1. Boş bir `students.csv` faylı yaradın.
2. Fayla `"name,email,password"` sətrini yazın.
3. Sizə artıq tanış olan aşağıdakı `users_dict` lüğəti verilib. Hər müraciyətdə `users_dict` içərisində verilmiş lüğətlərə bənzəyən bir lüğət alıb, onu `students.csv` faylına yazacaq `write_student` funksiyası yazın.

```
    users_dict = [
        {"name": "Rəşad", "email": "reshad@school.edu", "pass":
"12345678pass"},
        {"name": "əziz", "email": "eziz@school.edu", "pass": "pass"},
        {"name": "Ülvi", "email": "ulvi@school.edu", "pass": "123pass"},
        {"name": "Hikmət", "email": "hikmet@school.edu"},
    ]
```

Link: https://github.com/AzizNadirov/python-road-book/blob/master/code/tasks/files_7.ipynb

> **Diqqət**
>
> Bu bölmədən başlayaraq, kitabın ikinci böyük mövzusuna - obyekt yönümlü proqramlaşdırma öyrənməyə başlayacağıq. Bu bölmədə, kifayət qədər yeni və maraqlı mövzular olacaq. Ümumiyyətlə, bu bölmədən başlayaraq, nümunələrimizə, yazacağımız koda yavaş-yavaş daha yaradıcı yanaşacağıq. *Zehin açıqlığı ilə!*

OYP ilə ilkin tanışlıq

8.0. Obyekt Yönümlü dedikdə ??

İlk başda, sual yaranır - "obyekt yönümlü" proqramlaşdırma nə deməkdir? İndiyə qədər, bizim üçün obyekt - pythonda üzərində işlədiyimiz vahidlər - obyektlər idi. Məsələn verilənlər, funksiyalar, iteratorlar və sairə, biz bütün bunları müəyyən tipə məxsus obyekt kimi tanıyırıq. Ancaq, obyekt yönümlü proqramlaşdırma dilinin obyekt anlayışı bununla məhdudlaşmır. Obyekt Yönümlü Proqramlaşdırma(OYP) sayəsində - vahid proqram məntiqini bir obyekt şəklində yaza bilərik. Bu obyektin öz xüsusiyyətləri, öz metodları ola bilər. Məsələn, təqribən belə bir tapşırıq var: "elə bir proqram yazın ki, əl ilə insanın ad, soy ad, doğum ili daxil edilsin və lazım olanda(tələb olunanda) yaşı hesablansın". Bir qədər dumanlı məsələ qoyuluşudur, bəli. İlk ağıla gələn həllərdən biri belə ola bilər:

```
def take_person(name: str, surname: str, birth_year: int)->dict:
    """ take users properties """
    return {"name": name,
            "surname": surname,
            "birth_year": birth_year}

def get_age(person: dict)->int:
    """ calculate age of person """
    current_year = 2024
    return current_year - person.get("birth_year")

veli = take_person(name="Vəli",
                   surname="Vəliyev",
                   birth_year=1990)

eli = take_person(name="əli",
                   surname="Vəliyev",
                   birth_year=1993)

print(f"Vəlinin yaşı: {get_age(veli)}; əlinin yaşı: {get_age(eli)}")
```

```
Vəlinin yaşı: 34; əlinin yaşı: 31
```

Yuxarıda hər bir insanı - bir lüğət kimi qeyd etdik, yaşı hesablamaq üçün isə ayrıca `get_age` funksiyası yazdıq. Bəs bu yanaşmanın nə kimi çatışmamazlıqları var?

1. Bütöv bir məntiq - 2 ədəd bir-birindən xəbəri olmayan funksiya şəklində yazılıb. Bu funksiyalar məntiqi cəhətdən biri-birilərini tamamlasalar da, kod yazarkən bu məntiqi gərək ağlımızda saxlayaq, yadda saxlayaq.
2. Şəxslər üzərində aparıla biləcək əməliyyatları genişləndirməyə cəhd etsək(ad və soy addan istifadə etməklə email yaratmaq məsələn) yeni-yeni əlaqəsiz funksiyalar yazmış olacağıq, beləliklə, kodu idarə etmək çətinləşir. Bəs OYP istifadə edərək nə edə bilərdik? Hər bir şəxsi bir lüğət kimi yox, bir "Person" tipli obyekt kimi təsvir edə bilərdik, bu obyektin `name`, `surname`, `birth_year` adlı xüsusiyyətləri və `get_age` adlı metodu olardı. Daha sonra, hər bir yeni şəxsi `p = Person(name=..., surname=..., birth_year=...)` kimi yazmaqla yaradıb, yaşı `p.get_age()` kimi yazmaqla ala bilərdik. Beləliklə, biz şəxs adlı məvhumu vahid bir obyekt kimi təsvir etdik.

Obyektin xüsusiyyətləri və metodları. Müəyyən məvhumu obyekt kimi təsvir etdikdə, onunla bağlı olan iki növ məqamı göstərmək olar:

1. obyektlərin müəyyən *xüsusiyyətləri* olur - bunlar bir növ obyektə aid olan dəyişənlərdirlər və adətən hər hansı verilən strukturu olur(demək olar ki). Obyektin xüsusiyyəti - adından da göründüyü kimi, obyektin hər hansı xassəsini - xüsusiyyətini ifadə edir. Məsələn, öncəki şəxs misalında şəxsin `name`, `surname` və `birth_year` adlı xüsusiyyətləri vardı. Bunlar bir növ obyekt haqqında əlavə informasiya saxlayır.
2. obyekt üzərində müxtəlif əməliyyatlar icra edən funksiyalar - *metodlar* olur: metod - obyektə bağlı olan adi funksiyadır. Əgər xüsusiyyətlər informasiya saxlamaq üçün istifadə olunurdusa, metodlar obyekt üzərində konkret əməliyyatlar, hərəkətlər icra etmək üçündür. Öncəki şəxs obyektinin `.get_age()` metodu vardı, əgər obyektimiz `p` dəyişəninə mənimsədilibdirsə, o zaman `p.get_age()` bizə yaşı qaytarardı. Daha bir nümunə isə, indiyə qədər tanış olduğumuz xeyli sayda standart verilənlərin metodlarıdır. `'salam'.upper()` əmri ilə `salam` sətrinin `upper` metodunu çağırırıq, hansı ki öz növbəsində bizə həmin sətrin yuxarı registrli formasını qaytarır. Eyni şey siyahıların `append` metodu və digər metodlara da aiddir, bu metodlar - aid olduqları obyekt üzərində hansısa əməliyyat icra edir.

Obyektin qəlibi - siniflər. Müəyyən növ obyekt yaratmaq üçün hər hansı qəlib - forma olmalıdır. Deyək ki, əgər mən `name` və `surname` xüsusiyyətlərinə malik obyekt yaratmaq istəyirəmsə, bunu necəsə təsvir etməiyəm. Obyektlərin yaradılması üçün *siniflərdən* (class) istifadə edilir. *Sinif* və ya *klass* - yaradılacaq obyektin hansı xüsusiyyətlərə və hansı metodlara malik olacağını təsvir edən bir növ sxemdir, qəlibdir. Bu qəlib bir növ funksiyaya bənzəyir, biz həmin funksiyanı lazımi parametrlərlə çağırırıq və o bizə obyekt qaytarır. Yəni, biz sinifi hər dəfə çağırdıqda o bizə yeni bir obyekt yaradır və qaytarır, sinfi bir növ - obyekt yaratma dəzgahı kimi təsvir etmək olar. Sinif yaratmaq üçün `class` açar sözündən istifadə edilir. Gəlin birbaşa ən sadə sinifdən başlayaq:

```
class Person:
    name = "boş"
    surname = "boş"
```

```
p1 = Person()
print("p1.name: ", p1.name)
print("p1.surname: ", p1.surname)
```

Yuxarıda `class Person` deyərək `Person` adlı sinif yaratdıq və içərisində iki ədəd dəyişən yaratdıq - `name` və `surname`, hər ikisinin dəyəri `boş` sətri kimi təyin etdik. Bu ilk üç sətr ilə `name` və `surname` xüsusiyyətləri olan `Person` sinfi yaratdıq. Daha sonra biz `Person` sinfini çağırdıqda o bizə bir `Person` sinfinin obyektini qaytarır. Bu obyekt `Person` sinfinin məhsuludur, başqa sözlə - `Person` sinfinin *nüsxəsidir*(instance). `Person` içərisində təyin etdiyimiz `name` və `surname` xüsusiyyətləri sinfə məxsusdur və dəyərləri artıq sinfin içərisidəcə təyin olunub. Buna görə də, biz `Person` sinfinin nüsxəsinin bu xüsusiyyətlərinə müraciət etdikdə(`<nüsxə>.<xüsusiyyət_adı>` kimi yazmaqla) sinifdə təyin etdiyimiz `boş` sətrini almış oluruq. Ancaq biz obyektin xüsusiyyətinə dəyər mənimsətməklə, onun dəyərini dəyişə bilərik, qeyd edim ki, bu mövcud olmayan bir xüsusiyyət də ola bilər, əgər elə bir xüsusiyyət var idisə - dəyəri yenisi ilə əvəzlənəcək, yox idisə də yaradılıb mənimsətdiyimiz dəyərə sahib olacaq:

```
class Person:
    name = "boş"
    surname = "boş"

p1 = Person()
print("p1.name: ", p1.name)
print("p1.surname: ", p1.surname)

print("--------------------------")
p1.name = "əli"
p1.surname = "Vəliyev"
p1.email = "ali.valiyev@fakemail.com"

print("p1.name: ", p1.name)
print("p1.surname: ", p1.surname)
print("p1.email: ", p1.email)
```

```
p1.name:  boş
p1.surname:  boş
--------------------------
p1.name:  əli
p1.surname:  Vəliyev
p1.email:  ali.valiyev@fakemail.com
```

Bir sinifdən istifadə etməklə qeyri-məhdud dəfə nüsxə yaratmaq olar:

```
p2 = Person()
print(f"p2.name: {p2.name}; p2.surname: {p2.surname}")
```

```
p2.name: boş; p2.surname: boş
```

`p2` üçün yenə hər iki xüsusiyyət öz susmaya görə dəyərlərini alır.

8.1. Konstruktorlu siniflər

Öncəki `Person` sinfinin iki xüsusiyyəti `name` və `surname` vardı. Bu xüsusiyyətlər *sinif xüsusiyyətləri*(class variables, class properties) idi, çünki sinif içərisində təyin edilmişdilər. Sinif xüsusiyyətləri isə, bütün sinif nüsxələri üçün eyni olur. Buna görə də yüzlərlə `Person` nüsxələri yaratsaq belə, onların hamısı üçün xüsusiyyətlər eyni olacaq. Bəs əgər fərqli xüsusiyyətli nüsxələr yaratmaq istəsək necə etməli? Hər dəfə hər bir xüsusiyyət üçün `.name` və `.surname` xüsusiyyətlərini əl iləmənimsətmək olardı, amma bu o qədər də yaxşı çıxış yolu deyil. Bizə lazım olan sinif dəyişənləri yox *nüsxə dəyişənləridir*(instance variables) . Yəni, biz nüsxə yaradarkən, sinfə xüsusiyyətləri bir növ parametr kimi ötürək və bu parametrlər bizə lazım olan xüsusiyyətləri təyin etsin. İstəyimizə konstruktor vasitəsilə çata bilərik. *Konstruktor* - nüsxə yaradılarkən sinfin avtomatik olaraq işə düşən metodudur. Yəni, sinif daxilində bir metod təsəvvür edin, bu metod hər dəfə sinfimiz çağrılarkən işə düşəcək. Əlavə olaraq, sinfi çağırarkən ötürəcəyimiz parametrlər avtomatik olaraq bu konstruktor metoda ötürüləcək. Əksər proqramlaşdırma dillərində konstruktor metodun adı - sinfin adı ilə eyni olur, Pyhtonda isə konstruktor metod - `__init__` adlanır. Bütün nüsxə metodları kimi(bu haqqda ətraflı irəlilərdə danışacağıq) konstruktorun da ilk tələb olunan arqumenti - `self` olacaq. `self` adı nüsxəni ifadə edir. Bu məqamda hər şey bir az(ya da xeyli) dolaşıq və anlaşılmaz görünə bilər. Gəlin nümunələrlə hər şeyi yerbəyer edək. Person sinfimizi bir az da təkminləşdirək - konstruktor əlavə edək, konstruktor standart `self` arqumentindən savayı, `name` və `surname` adlı parametrlər də alsın və onlardan istifadə edərək nüsxəmizin `ad` və `soyad` xüsusiyyətlərini təyin etsin:

```
class Person:
    def __init__(self, name: str, surname: str):
        print("in __init__")
        self.ad = name
        self.soyad = surname
        print("__init__ done.")

p1 = Person(name="əli", surname="Vəliyev")
print(f"Ad: {p1.ad}; Soyad: {p1.soyad}")
print(p1.name)      # biz 'name' adlı xüsusiyyət yaratmamışıq
```

```
in __init__
__init__ done.
Ad: əli; Soyad: Vəliyev
AttributeError: 'Person' object has no attribute 'name'
```

Gəlin indi kodu addım-addım incələyək. Sinfin `__init__` konstruktorunu təyin etmişik. Bu konstruktor `self`, `name` və `surname` parametrlərini alır. Biz `p1 = Person(name="əli", surname="Vəliyev")` kimi yazdıqda `name="əli", surname="Vəliyev"` arqumentləri avtomatik olaraq konstruktora ötürülür. Bəs `self` ? Avtomatik olaraq `self = p1` kimi ötürülür. Yəni yaradılan

nüsxə(`p1` kimi işarə etdiyimiz) konstruktorun ilk arqumenti olur və avtomatik ötürülür. Deməli: `self` yeni yaradılan nüsxədir, `name` və `surname` parametrlərini də almışıq. Etməli olduğumuz tək şey - `self` nüsxəsinin bizə lazım olan xüsusiyyətlərini yaratmaqdır ki, onu da `self.ad` və `self.soyad` kimi yazmaqla edirik. Yəni, nüsxənin iki `ad` və `soyad` adlı xüsusiyyətini yaradırıq və `name` , `surname` dəyərlərini onlara mənimsədirik. Konstruktor icra olunandan sonra yaradılan nüsxə `Person` sinfi tərəfindən qaytarılır və `p1` dəyişəninə mənimsədilir. Sonda `p1.name` deyərək `p1` nüsxəsinin `name` adlı xüsusiyyətini almaq istəyirik və xəta alırıq: `AttributeError: 'Person' object has no attribute 'name'` . Məsələ burasındadır ki, bizim nüsxəmizin `name` adlı xüsusiyyəti, attributu yoxdur, yəni nə sinif içərisində `self.name = ...` demişik, nə də nüsxə sinif tərəfindən qaytarılandan sonra `p1.name = ...` demişik. Odur ki, nüsxəmizdə `name` adlı bir şey yoxdu. Yuxarıdakı sinifdə biz parametrləri `name` və `surname` kimi alıb, `ad` və `soyad` kimi saxlayırdıq, ancaq elə `name` və `surname` kimi də saxlamaq olardı, bunun üçün konstruktoru belə yazmaq olardı:

```
...
    def __init__(self, name: str, surname: str):
            print("in __init__")
            self.name = name
            self.surname = surname
            print("__init__ done.")

p1 = Person(name="əli", surname="Vəliyev")
print(f"Ad: {p1.name}; Soyad: {p1.surname}")
```

Belə "eyniadlı" yazılış geniş istifadə olunur, sadəcə, yeni başlayanları çaşdırdığı üçün xüsusiyyətləri `ad` və `soyad` kimi saxlamışdıq. Gəlin `p1 = Person(name="əli", surname="Vəliyev")` icra edilərkən baş verənlərə bir daha ardıcıllığa baxaq:

1. `Person` çağrılır və sinif içərisində `Person` sinfinin yeni "boş" nüsxəsi yaradılır.
2. Sinfin konstruktoru işə düşür və öncə yaradılmış "boş" nüsxə(`self` adı altında) və sinfi çağırarkən ötürülmüş parametrlər(`name="əli"` , `surname="Vəliyev"`) bu konstruktora ötürülür. Konstruktor içərisində istədiyim xüsusiyyətləri yaradırıq.
3. Konstruktor işini bitirən kimi, (1) addımında çağrılmış `Person` sinfi geriyə - ilk öncə "boş" yaradılmış və daha sonra konstruktor içərisində müəyyən xüsusiyyətlərlə tamamlanmış nüsxəni qaytarır.
4. `Person` sinfinin qaytardığı tamamlanmış nüsxə `p1` dəyişəninə mənimsədilir.

8.2. Action! Metodlar

Ötən başlıqda, bütün sinif məntiqini iki hissəyə informasiya(xüsusiyyətlər) və hərəkətlərə(metodlara) bölmüşdük. Sinif və nüsxə metodları ilə müəyyən qədər tanışlığımız yarandı, indi isə keçək metodlara! Özü-özlüyündə, metod sadəcə nüsxəyə bağlı olan bir funksiyadır. Əslində irəliləyən bölmələrdə görəcəyik ki heç də bütün metodlar nüsxəyə bağlı olmur, bəziləri sinfə, bəziləri isə heç birinə bağlı olmur. Odur ki, hələki məhz *nüsxə metodları*(instance methods) haqqında danışacağıq. *Nüsxə metodları* sinif daxilində təyin olunmuş və ilk aldıqları parametr nüsxənin özü olan funksiyalardır. Bu metodlar ilk aldıqları parametr - `self` , yəni nüsxənin özü olur və avtomatik olaraq ötürülür. Nümunə üçün, `Person` sinfimizə `get_age` metodu əlavə edək:

```
class Person:
    def __init__(self, name: str, surname: str, birth_year: str):
        self.name = name
        self.surname = surname
        self.birth_year = birth_year

    def get_age(self)->int:
        current_year: int = 2024
        return current_year - self.birth_year

p1 = Person(name="əli", surname="Vəliyev", birth_year=1995)
print(f"Ad: {p1.name}; Soyad: {p1.surname}")
p1_age = p1.get_age()
print(f"Doğum ili: {p1.birth_year} Yaşı: {p1_age}")
```

```
Ad: əli; Soyad: Vəliyev
Doğum ili: 1995 Yaşı: 29
```

`get_age` nüsxə metodu olduğu üçün, təyin edilərkən yazılan ilk arqument `self` olur. Deyək ki, metodu `p1.get_age()` deyib çağırırıqsa, avtomatik olaraq `self=p1` kimi ötürülür. Metod daxilində `self` adından istifadə edərək, nüsxənin xüsusiyyətlərini ala və yaxud başqa metodları çağıra bilərik. Məhz buna görə, metod daxilində `self.birth_year` yazdıqda `p1.birth_year` almış oluruq. Gəlin, daha bir metod reallaşdıraq: `generate_email()` metodu yaradaq, bu metod `domain` adlı sətr tipli parametr alacaq, şəxsin ad, soy adından və ötürülən domendən istifadə etməklə email ünvan qaytaracaq. Əlavə olaraq, email ünvan içərisində ə, ğ, ç, ı, ü kimi hərfləri başqa hərflərlə əvəz edirik:

```
class Person:
    def __init__(self, name: str, surname: str, birth_year: str):
        self.name = name
        self.surname = surname
        self.birth_year = birth_year

    def get_age(self)->int:
        current_year: int = 2024
        return current_year - self.birth_year

    def generate_email(self, domain: str)->str:
        replacements = {'ə': 'e', 'ğ': 'gh', 'ç': 'ch', 'ı': 'i', 'ü': 'u'}
        email = f"{self.name}.{self.surname}@{domain}".lower()
        for letter in replacements.keys():
            email = email.replace(letter, replacements[letter])
        return email

p1 = Person(name="əli", surname="Vəliyev", birth_year=1995)
print(f"Ad: {p1.name}; Soyad: {p1.surname}")
p1_age = p1.get_age()
print(f"Doğum ili: {p1.birth_year} Yaşı: {p1_age}")
```

```
email = p1.generate_email("fakemail.com")
print(f"Email: {email}")
```

```
Ad: əli; Soyad: Vəliyev
Doğum ili: 1995 Yaşı: 29
Email: eli.veliyev@fakemail.com
```

Bir daha, `email = p1.generate_email("fakemail.com")` kimi yazdıqda, bir növ `p1.generate_email(self=p1, domain="fakemail.com")` kimi yazmış oluruq. Bununla da demək olar ki məqsədimizə çatdıq. İlk başda lüğətlərlə etdiklərimizi indi xüsusi `Person` sinfi ilə edirik. Sinfimizə üstəlik bir `id` adlı xüsusiyyət də əlavə edək və bir neçə nüsxə yaradaq:

```
class Person:
    def __init__(self, id: int, name: str, surname: str, birth_year: str):
        self.id = id
        self.name = name
        self.surname = surname
        self.birth_year = birth_year

    def get_age(self)->int:
        current_year: int = 2024
        return current_year - self.birth_year

    def generate_email(self, domain: str)->str:
        replacements = {'ə': 'e', 'ğ': 'gh', 'ç': 'ch', 'ı': 'i', 'ü': 'u'}
        email = f"{self.name}.{self.surname}@{domain}".lower()
        for letter in replacements.keys():
            email = email.replace(letter, replacements[letter])
        return email

p1 = Person(0, "Ali", "Valiyev", 1990)
p2 = Person(1, "Hikmet", "Nesibov", 2000)
p3 = Person(2, "Arzu", "Salmanli", 1999)
```

Sinif xüsusiyyətləri və Nüsxə xüsusiyyətləri

Siniflərlə ilk tanışlığımızda sinif xüsusiyyətlərindən aktiv istifadə edirdik. Sinif xüsusiyyətləri - adından da göründüyü kimi, ümumi sinfə aid olan, nüsxədən-nüsxəyə dəyişməyən məlumatları saxlamaq üçün istifadə edilir. Baxmaq üçün sadə bir `Triangle` - üçbucaq sinfi yazaq:

```
class Triangle:
    sides: int = 3
    def __init__(self, d1: float, d2: float, d3: float):
        self.d1 = d1
        self.d2 = d2
        self.d3 = d3

    def get_perimeter(self)->float:
```

```
        return self.d1 + self.d2 + self.d3

t1 = Triangle(5, 5, 7)
t2 = Triangle(10, 5, 11.1)

print("t1 Figure sides: ", t1.sides)
print("t2 Figure sides: ", t2.sides)

print("t1 Perimeter: ", t1.get_perimeter())
print("t2 Perimeter: ", t2.get_perimeter())
```

```
t1 Figure sides:  3
t2 Figure sides:  3
t1 Perimeter:  17
t2 Perimeter:  26.1
```

Sinifdəki `sides` xüsusiyyəti sinif xüsusiyyətidir, konstruktordan əvvəl təyin olunub. `d1`, `d2` və `d3` isə nüsxə xüsusiyyətləridirlər - konstruktor içərisində təyin olunub və ən əsası `self.xx = ...` kimi yaradılıb, yəni nüsxəyə "pərçimlənib". Odur ki, sinif xüsusiyyəti olan `sides` bütün nüsxələr üçün təyin olunmuş qiyməti daşıyacaq, tərəflər(`d1, d2, d3`) isə konstruktorda ötürdüyümüz qiymətləri alacaq.

Nüsxə, Sinif və Statik Metodlar

İndiyə qədər yazdığımız metodlar nüsxə metodları idi. Nüsxə metodları nüsxəyə bağlı olur və ilk parametr kimi həmin nüsxəni qəbul edir, bu nüsxəni `self` kimi işarə edirdik. Əlavə olaraq - *sinif metodları* (class methods) və *statik metodlar*(static methods) da mövcuddur.

Sinif metodları

Sinif metodları konkret nüsxəyə deyil, sinfin özünə bağlı olur və ilk parametr kimi avtomatik olaraq sinfi qəbul edir. Nüsxə metodlarında nüsxəni `self` kimi işarə etdiyimiz kimi, sinif metodlarında sinfi bildirən ilk parametri `cls` kimi işarə edirik. Üstəlik, sinif metodları nüsxəyə bağlı olmadıqları üçün, onları ümumi sinfin adından çağırmaq olar: `A.cls_method()` - burda `A` sinfin adı, `cls_method` isə sinfin sinif metodur. Sinif metodları sinif üçün ümumi olan əməliyyatları icra edir. Metodu sinif metodu kimi elan etməkçün `classmethod` dekoratorundan istifadə olunur. Gəlin bəsit `Car` sinfi təyin edək:

```
class Car:
    wheels = 4
    doors = 4
    def __init__(self, brand: str, model: str)->None:
        self.brand = brand
        self.model = model

    @classmethod
    def class_info(cls)->str:
        return f"Cars with {cls.wheels} wheels and {cls.doors} doors."
```

```
m5 = Car(brand='BMW', model='M5')
print("m5.brand, m5.model, m5.wheels, m5.doors:")
print('\t', m5.brand, m5.model, m5.wheels, m5.doors)
print("class info:")
print('\t', Car.class_info())
```

```
m5.brand, m5.model, m5.wheels, m5.doors:
     BMW M5 4 4
class info:
     Cars with 4 wheels and 4 doors.
```

Sinfin `wheels` və `doors` adlı sinif xüsusiyyətləri, `class_info` adlı sinif metodu var, üstəlik sinfin konstruktoru `brand` və `model` adlı parametrlər alıb nüsxə xüsusiyyətlərini təyin edir. `class_info` metodu `@classmethod` dekoratoru ilə təyin edildiyi üçün, sinif metodu kimi təyin olunur və ilk arqumenti - metodun aid olduğu sinif, yəni `Car` sinif obyekti olacaq. Bu arqumenti adətən `cls` kimi işarə edirlər. Sinif üzərindən sinif xüsusiyyətlərini ala bildiyimiz üçün, `cls.wheels` və `cls.doors` deyərək sinfin adından sinif xüsusiyyətlərini alırıq. Öncə də dediyim kimi, sinif metodlarını - sinif adından istifadə etməklə çağırmaq olar: `Car.class_info()`. Ancaq, əlavə olaraq nüsxədən istifadə də sinif metodlarından istifadə etmək olar. Belə olan zaman python nüsxənin sinfini alıb, həmin sinfə görə sinif metodunu çağırır: `print(m5.class_info())`. Əslində bu davranış kifayət qədər məntiqlidir, çünki nüsxədən istifadə edərək onun sinfini almaq o qədər də çətin deyil, əslində biz də istənilən obyektin sinfini ala bilərik, bunun üçün obyektin `.__class__` attributundan istifadə etmək olar.

```
print(m5.__class__)     # <class '__main__.Car'>
```

Bu isə o deməkdir ki `m5` nüsxəsindən istifadə edərək, yuxarıdakı üsulla `Car` sinfini alıb, ona nəzərən `class_info` sinif metodunu çağıra bilərik:

```
print(m5.__class__.class_info())   # Cars with 4 wheels and 4 doors.
```

Gəlin, yekun olaraq daha bir sinif yaradaq: `Email` sinfi iki üsulla email ünvanı yaratmağa imkan verir:

1. Konstruktoun aldığı `name` və `surname` parametrləri vasitəsilə
2. `from_string` adlı sinif metodu vasitəsilə: metod bir sətr alır və onu parçalayıb `name`, `surname` kimi ilk iki altsətri götürüb onlardan email yaradır.

```
class Email:
    domain = "@fakemail.com"
    separator = "."
    def __init__(self, name: str, surname: str)->None:
        self.name = name
        self.surname = surname
```

```
        self.email = self.build_email(name=self.name, surname=self.surname)

    @classmethod
    def build_email(cls, name: str, surname: str)->str:
        replacements = {'ə': 'e', 'ğ': 'gh', 'ç': 'ch', 'ı': 'i', 'ü': 'u'}
        email = f"{name}{cls.separator}{surname}{cls.domain}".lower()
        for letter in replacements.keys():
            email = email.replace(letter, replacements[letter])
        return email

    @classmethod
    def from_string(cls, string: str, separator=' ')->str:
        name, surname = string.split(separator)[:2]
        email = cls.build_email(name=name, surname=surname)
        return email

e1 = Email("Əli", "Vəliyev")
print(e1.email)
print(Email.from_string("Hikmət Nəsibov"))
```

```
eli.veliyev@fakemail.com
hikmet.nesibov@fakemail.com
```

`Email` sinfinin özəyini `build_email` sinif metodu təşkil edir. Bu metod `name` və `surname` sətrlərini alaraq email hazırlayır. Metodu həm konstruktordan, həm də `from_string` sinif metodundan çağırırıq. `from_string` metodu sinfin **nüsxə yaratmadan** işləməsini təmin edir. Sinif metodlarının da əsas təyinatı məhz budur.

Statik metodlar

Statik metodlar ümumiyyətlə, nə sinfə, nə də ki nüsxəyə bağlı olur, bu metodlar sadə funksiyalardırlar. Təyin olunmaları - `staticmethod` dekoratoru vasitəsilə həyata keçirilir. Bu metodlardan - sinif və nüsxədən heç bir asılılığı olmayan, müstəqil proqram məntiqinin yazılması üçün istifadə olunur. Statik metod əvəzinə adi nüsxə metodu yazıb, `self` arqumenti alıb, bu arqumentdən istifadə etməmək də olar, amma nüsxədən istifadə etməyəcəyiksə, niyə də alaq... Üstəlik, statik metodları da sinif metodları kimi sinif və nüsxə adından istifadə edərək çağırmaq mümkündür:

```
    class Triangle:
        sides = 3
        def __init__(self, d1: float, d2: float, d3: float):
            self.d1 = d1
            self.d2 = d2
            self.d3 = d3

    @staticmethod
    def calc_perimeter(d1: float, d2: float, d3: float):
        return sum([d1, d2, d3])
```

```
    def get_perimeter(self):
        return self.d1 + self.d2 + self.d3

    t1 = Triangle(5, 5, 7)
    print(t1.calc_perimeter(5, 5, 7))   # nüsxə üzərindən #17
    print(Triangle.calc_perimeter(5, 5, 7)) # sinif üzərindən #17
```

Öncə də dediyim kimi, əksər vaxtı statik və ya sinif metodlarından istifadə edə bildiyimiz halda istifadə etməyə də bilərik, ümumiyyətlə, adətən bir şeyi yazmağın, işləməyə vadar etməyin çoxlu sayda yolu olur, hər dəfə nəyisə daha yaxşı yazmaq olur. Belə olan halda bizim üçün kod yazmaq - yalnız məsələni həll etmək yox, həmçinin "gözəl həll etmək" kimi əlavə şərtə çevrilir. Əlbəttə ki, burdakı "gözəl" sözü yalnız kodun gözəlliyini bildirmir, həmçinin kodun optimallıq(qısa və sürətli) və oxumluluq(koda baxdıqda hansı hissənin nə iş görməsi və necə əlaqələndirilməsi aydın olmalıdır) cəhətdən gözəl olmalı olduğunu bildirir. Ona görə də, bu bölmənin əvvəlində demişdim ki biz artıqyavaş-yavaş yalnız "nə yazdığımıza" deyil, həmçinin "necə yazdığımıza" diqqət edəcəyik. Bəs, kitabdakı kod nümunələri hamısı gözəldirmi - xeyir. Hardasa daha aydın etmək üçün yersiz uzun yazmışam, hardas isə ağlıma gələn ən yaxşı variantı yazmışam, öncə də dediyim kimi - həmişə daha yaxşı yazmaq olar.

property - hesablanan xüsusiyyətlər

Biz nüsxənin xüsusiyyətinin alınmasına(*getting*) və dəyişdirilməsinə(*setting*) nəzarət edə bilərik. Bunun üçün ən sadə yol - `property` dekorator funksiyasıdır. Bu dekorator ilə metod təyin etdikdə - metod bir növ xüsusiyyətə çevrilir və biz hər dəfə bu xüsusiyyəti əldə etməyə çalışdıqda, əslində pərdəarxası metodu çağırmış oluruq. Bununla xüsusiyyətin alınması prosesini tənzimləyə bilərik. Az öncə yazdığımız `Triangle` sinfini yenidən yazaq, amma bu dəfə `get_perimeter` məntiqini bir xüsusiyət kimi qeyd edək:

```
class Triangle:
   sides = 3
   def __init__(self, d1: float, d2: float, d3: float):
       self.d1 = d1
       self.d2 = d2
       self.d3 = d3
       self.__p = self.d1 + self.d2 + self.d3

   @property
   def perimeter(self):
       print("Getting Perimeter...")
       return self.__p

t = Triangle(1,2,3)
print(f"t.perimeter: {t.perimeter}")
```

```
Getting Perimeter...
t.perimeter: 6
```

Nə baş verdi: `perimeter` əslində bir metod olsa da, `@property` dekoratoruna bürüdüyümüzə görə metodun adı xüsusiyyətə çevrilir, bu xüsusiyyəti almağa çalışdıqda isə, metod işə düşür. Bunun sayəsində, özündə perimetri saxlayan `__p` dəyişənini istifadəçidən kənar saxlayıb, `perimetr` adlı metod-xüsusiyyəti veririk. Bu metodun içərisində istədiyimiz məntiqi yaza bilərik - istifadəçini nədənsə xəbərdar edə, xüsusiyyəti hansısa məntiqə görə dəyişib verə və sairə edə bilərik. Bunlar sadəcə xüsusiyyətin "getting" - almağa çalışarkən etdiklərimiz idi. Əlavə olaraq, xüsusiyyətin dəyişdirilməsini - "setting" prosesini də oxşar yolla nəzarətdə saxlaya bilərik. Bunun üçün, `propery` dekoratoruna bürünmüş getter metodumuzun `.setter` dekorator-funksiya dekoratoru ilə yenidən təyin edirik, bu metod əlavə olaraq daha bir `value` parametri də alacaq. Bu parametr - xüsusiyyətin dəyərini dəyişəcəyimiz yeni dəyər olacaq. Yeni setter metodunu sinfimizə əlavə edək:

```
class Triangle:
    sides = 3
    def __init__(self, d1: float, d2: float, d3: float):
        self.d1 = d1
        self.d2 = d2
        self.d3 = d3
        self.__p = self.d1 + self.d2 + self.d3

    @property
    def perimeter(self):
        print("Getting Perimeter...")
        return self.__p

    @perimeter.setter
    def perimeter(self, value):
        if self.__p != value:
            print("You are setting incorrect value...")
        else:
            self.__p = value

t = Triangle(1,2,3)
print(f"t.perimeter: {t.perimeter}")
t.perimeter = -1
print(f"t.perimeter: {t.perimeter}")
t.perimeter = 6.0
print(f"t.perimeter: {t.perimeter}")
```

```
Getting Perimeter...
t.perimeter: 6
You are setting incorrect value...
Getting Perimeter...
t.perimeter: 6
Getting Perimeter...
t.perimeter: 6.0
```

Hər dəfə istifadəçi `property` dekoratoruna bürünmüş `perimeter` xüsusiyyətini dəyişməyə cəhd etdikdə `@perimeter.setter` dekoratoru ilə təyin olunmuş metod işə düşəcək. Bu metod daxilində

yazdığımız məntiqə görə, əgər istifadəçinin təyin etmək istədiyi dəyər üçbucağın perimetrinə bərabər deyilsə - xəta ismarıcı göstərib perimetri dəyişmirik. Əks təqdirdə isə perimetri ötürülmüş dəyərə dəyişirik. Setterlər validasiya prosesi üçün əla vasitədirlər. Əslində, `@property` Pythonda deskriptorlar - obyektin alınmasını, dəyişdirilməsi və silinməsini təmin edən metodlar üçün qısayoldur, bütün bunlara baş vurmadan, tək bir dekorator ilə xüsusiyyətin həm alınmasına, həm də dəyişdirilməsinə nəzarət etməyə imkan verir.

Son nümunə. Öyrəndiklərimizi daha yaxşı bərkitmək üçün, `Person` sinfimizə əlavə "fayla yazmaq" və "fayldan oxumaq" funksionallıqlarını qataq. Təsəvvür edək ki, sinfin nüsxələri insanlardır və biz bu insanları hər hansı mətn faylına yazmaq və oradan geriyə - oxuyub `Person` sinfinin nüsxəsi halına gətirmək istəyirik. Bunun üçün iki ədəd sinif metodu: `upload` - fayla yükləmək və `load` - fayldan yükləmək üçün yazaq. Hər iki metodun məntiq cəhətdən nüsxələrə aidiyyatı olmadığı üçün, onları sinif metodları şəklində yazaq. Üstəlik, fayla ancaq mətn yaza bildiyimiz üçün, aralıqda `Person` nüsxəsini bir sətr mətnə və əksinə - bir sətr mətni `Person` nüsxəsinə çevirəcək iki ədəd metod lazım olacaq.

```
from typing import List

   class Person:
       def __init__(self, id, name, surname, birth_year):
           self.id = id
           self.name = name
           self.surname = surname
           self.birth_year = birth_year

       def get_age(self):
           current_year = 2024
           return current_year - self.birth_year

       def generate_email(self, domain):
           replacements = {'ə': 'e', 'ğ': 'gh', 'ç': 'ch', 'ı': 'i', 'ü': 'u'}
           email = f"{self.name}.{self.surname}@{domain}".lower()
           for letter in replacements.keys():
               email = email.replace(letter, replacements[letter])
           return email

       @staticmethod
       def person_to_str(person: 'Person', sep: str)->str:
           return f"{person.id}{sep}{person.name}{sep}\
               {person.surname}{sep}{person.birth_year}\n"

       @classmethod
       def str_to_person(cls, person_str: str, sep: str)->'Person':
           splitted_person = person_str.split(sep)
           id_, name, surname, birth_year = splitted_person
           person_instance = cls(id_, name, surname, birth_year)
           return person_instance

       @classmethod
       def upload(cls,
                  people: List['Person'],
                  to: str,
```

```
                  overwrite: bool=False,
                  sep=',')->None:
            """ upload people as text into file """
            print(f"Uploading people into '{to}', overwrite: '{overwrite}' ")
            mode = 'w' if overwrite else 'a'
            with open(to, mode) as file:
                for person in people:
                    person_str = cls.person_to_str(person, sep=sep)
                    file.write(person_str)
            print("Done.")

        @classmethod
        def load(cls, file_path: str, sep=',')->List['Person']:
            """ load list of Person instances from file """
            people = []
            with open(file_path, 'r') as file:
                file_lines = file.readlines()
                for person_str in file_lines:
                    person_instance = cls.str_to_person(person_str=person_str,
                                                        sep=sep)
                    people.append(person_instance)
            return people

    # 1. Nüsxələri yaradaq
    p1 = Person(0, "Ali", "Valiyev", 1990)
    p2 = Person(1, "Hikmet", "Nesibov", 2000)
    p3 = Person(2, "Arzu", "Salmanli", 1999)

    # 2. people.txt faylına yazaq
    filename = 'people.txt'
    person_list = [p1, p2, p3]
    Person.upload(people=person_list, to=filename, overwrite=True)

    # 3. Fayldan yükləyək
    loaded_people = Person.load(file_path=filename)
    print("Loaded People: ", loaded_people)
```

```
Uploading people into 'people.txt', overwrite: 'True'
Done.
Loaded People:
    [<__main__.Person object at ...>,
    <__main__.Person object at ...>,
    <__main__.Person object at ...>]
```

İlk öncə "Person nüsxələrindən ibarət siyahı" anlamına gələn tip göstəricisini import edirik. Bunun üçün `List` tip göstəricisindən istifadə edirik. Dəyişənin tipini `List['Person']` və ya `List[Person]` kimi işarə edərək, dəyişənin `Person` tipli nüsxələrdən ibarət bir siyahı olacağını deyirik. Sinfimizdə iki əsas yenilik var - `upload` və `load`. `upload` sinif metodu standart `cls` parametrindən əlavə: `people` - mətnə çevrilib fayla yazılmalı olan `Person` nüsxələri siyahısı; `to` - içərisinə yazacağımız faylın yolu; `overwrite` - faylın içərisindəkilərin üzərindən yazılsınmı(faylın

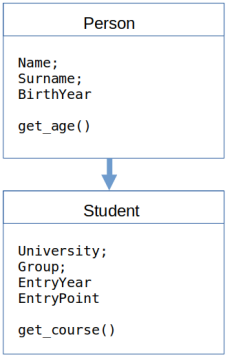
açılma modunu bu parametrlə tənzimləyirik, üzərindən yazmaq lazım olsa `mode='w'`, əks halda `mode='a'` olacaq); `sep` - nüsxələri mətn formasına çevirərkən xüsusiyyətlərin arasında ayırıcı kimi kimi istifadə olunacaq simvol. Nüsxənin mətnə çevrilməsini ayrıca `person_to_str` statik metodunda baş verir. Metod sadəcə f sətr içərisinə nüsxənin bütün xüsusiyyətlərini yazıb araya `sep` ayırıcısını daxil edir. Daha sonra alınan mətn fayla yazılır və beləcə siyahıdakı bütün nüsxələr üçün. `load` metodunda bunun əksi baş verir: `file_path` - içərisində şəxslər olan mətn faylının yolu; `sep` - isə ayırıcıdır. Mətni sətrbəstr oxuyaraq nüsxəyə çeviririk. Mətn sətrini nüsxəyə çevirmək üçün ayrıca `str_to_person` sinif metodunu yazmışıq. Metod `person_str` - mətn sətrini və `sep` ayırıcını qəbul edir. Sətri `sep` ayırıcısından istifadə edib parçalayır və uyğun xüsusiyyətləri bildirən dəyişənlərə mənimsədir. Daha sonra o dəyişənləri `person_instance = cls(id_, name, surname, birth_year)` deyərək `Person` sinfinə ötürür və geriyə artıq `Person` nüsxəsi alır. Bu əməliyyat növbə ilə mətndəki bütün sətrlərə tətbiq olunur və sonda nüsxələr siyahısı qaytarılır.

> **Tip**
>
> Real təcrübədə verilənləri bu cür emal edib fayla yazmaq o qədər də yaxşı fikir deyil. Bunun üçün gələcəkdə tanış olacağımız data klasslar və ya kənar tərəf kitabxana olan pydantic istifadə etmək olardı. Daha bir variant kimi, sadə mətn faylı əvəzinə hər hansı bir SQL tipli verilənlər bazasından istifadə etmək olardı. Sevindirici xəbər odur ki, biz sizinlə bunların hamısı haqqında danışacağıq: pydantic və ya dataclasslarla verilənlərin sxemalarını qurmaq, SQL sorğu dilinin əsasları, SQLAlchemy ilə sadə sqlite3 bazasına qoşulmaq və sairə. Sonda bizim yuxarıda etdiyimiz, əjdaha bir nümunəyə çevriləcək.

8.3. Siniflərdə Vərəsəlik

Yadınızdadırsa, funksiyalarla tanış olarkən funksiyaların kodun "təkrar istifadəsi" üçün necə böyük imkanlar yaratdığını, necə faydalı olduğunu yazırdım. Baxın, siniflər - "təkrar istifadə" cəhətdən daha güclüdürlər! Bunun əsas səbəbkarlarından biri - siniflərdə *vərəsəliyin*(inheritance) olmasıdır. Vərəsəlik sayəsində bir sinif digər sinfin metod və xüsusiyyətlərinə sahib olur, onları bir növ vərəsəsi kimi "miras alır".

Bu nədir və necə işləyir: adətən bir əsas - özündə ümumi xüsusiyyət və metodları saxlayan sinif olur. Deyək ki, `Person` sinfimiz var, bu sinif bütün şəxslər-insanlar üçün ümumi xüsusiyyətlərə sahibdir - ad, soyad və doğum ili. Bundan sonra bir `Student` - tələbə sinfi də yazmaq istəsək, məntiqi cəhərdən `Person` içərisindəki metod və xüsusiyyətləri `Student` üçün də yenidən yazmalıyıq, çünki tələbə də insandır və insanlara məxsus xüsusiyyətlərə sahib olmalıdır. Bu məqamda köməyimizə vərəsəlik anlayışı gəlir. Əgər, `Student` sinfi `Person` sinfinin vərəsəsidirsə, başqa sözlə - `Person` sinfindən törəyibsə(əslində o qədər də uyğun ifadə deyil), o zaman `Student` sinfi `Person` sinfinin malik olduğu xüsusiyyət və metodları miras alacaq. Bizim artıq `Person` sinfimiz var, gəlin bir `Student` sinfi də yaradaq, bu sinfin `university` -universitetin adı, `group` - tələbənin qrupu, `entry_year` -qəbul ili və `entry_point` -qəbul balı kimi xüsusiyyətləri və `get_course` adlı metodu olacaq, hansı ki tələnin neçənci kursda oxuduğunu qaytaracaq(sadəlik naminə sadəcə cari ildən `entry_date` çıxacağıq). `name`, `surname`, `birth_year` kimi xüsusiyyətləri isə elə `Person` sinfində var. Yeri gəlmişkən, əgər `Student` sinfi `Person` sinfinin vərəsəsdirsə, o zaman `Person` - *supersinif*(super class), *valideyn sinif*(parent class), `Student` sinfi isə *övlad sinif*(child class) adlanır.




```
class Student(Person):
    def __init__(self,
                 id: int,
                 name: str,
                 surname: str,
                 birth_year: int,
                 university: str,
                 group: str,
                 entry_year: int,
                 entry_point: int
                 ):
        super().__init__(id, name, surname, birth_year)
        self.university = university
        self.group = group
        self.entry_year = entry_year
        self.entry_point = entry_point

    def get_course(self)->int:
        current_year = 2024
        return current_year - self.entry_year

veli = Student(id=0, name="Veli", surname="Veliyev", birth_year=2005,
university="BDU",
               group="1234A", entry_year=2022, entry_point=450)

print("veli.name: ", veli.name)
print("veli.university: ", veli.university)
print("veli.get_age(): ", veli.get_age())
print("veli.get_cource: ", veli.get_course())
print("veli.generate_email: ", veli.generate_email(domain="fakemail.com"))
```

```
veli.name:  Veli
veli.university:  BDU
veli.get_age():  19
```

```
veli.get_cource:  2
veli.generate_email:  veli.veliyev@fakemail.com
```

İlk öncə ondan başlayaq ki `B` sinfinin - `A` sinfinin vərəsəsi olduğunu göstırmək üçün, `B` sinfini belə təyin edirik: `class B(A)` . Yəni, valideyn sinif, övlad sinfinin başlığında göstırilir. `class Student(Person)` kimi yazaraq `Student` sinfimizin `Person` sinfinin vərəsəsi, övlad sinfi olduğuna işarə edirik. `Student` sinfinin konstruktoru tələbənin bütün xüsusiyyətlərini qəbul edir - tələbənin həm bir şəxs kimi almalı olduğu parametrlər, həm də bir tələbə kimi. Daha sonra bu məlumatların `Person` sinfinə aid olanlarını seçib valideyn sinfi olan `Person` -a ötürməliyik ki, tələbəmiz bir şəxs olsun - `Person` sinfinin xüsusiyyətlərinə sahib olsun. Bunun üçün `super` qurma funksiyasından istifadə edirik. `super` funksiyası - sinif daxilində çağırıldıqda valideyn sinfi ilə övlad sinfinin nüsxəsi arasında bir bağ, əlaqə qaytarır. Odur ki biz `super().__init__()` çağırdıqda, `Person` sinfinin konstruktorunu çağırmış oluruq. Bir `Person` sinfinin tələb etdiyi bütün parametrləri bu konstruktora ötürdükdən sonra tələbənin öz nüsxə xüsusiyyətlərini yaradırıq: `self.university = university` və s. Daha sonra sadəcə `Student` metodlarını təyin edirik, vəssalam. `super().__init__(id, name, surname, birth_year)` əvəzinə `Person.__init__(self, id, name, surname, birth_year)` də yazmaq olardı, yəni `super()` əvəzinə - bir başa valideyn sinfinin adını yazmaq olardı. Ancaq, bu zaman sinfin adını dəyişmək qərarına gəlsək, artıq bir neçə yerdə dəyişməli olacağıq, üstəlik valideyn sinfin konstruktoruna əl ilə övlad nüsxəsi olan `self` də ötürməli oluruq. `super()` isə avtomatik olaraq valideyn sinfin "proksi-sinfini" qaytarır, odur ki daha sinfin adını yazmağa ehtiyac qalmır.

Attributların örtülməsi - overwriting

Öncəki başlıqda gördük ki əgər `B` sinfi `A` sinfinin vərəsəsidirsə, o zaman `A` sinfinin metod və xüsusiyyətlərini miras alır. Yəni, əgər `A` sinfində `m` metodu varsa, `B` sinfində həmçinin olacaq. Bəs əgər `B` sinfinin özünün də elə bir metodu olsa - onda hansı metodtapılacaq, sinfin öz metodu, ya valideyn sinfinin metodu? Suala cavab tapmaq bu cür iki ədəd sinif ataq:

```
class A:
       a = "A.a"
       b = "A.b"
       def method(self):
           print("method, from class: ", self.__class__.__name__)

   class B(A):
       def __init__(self, a):
           super().__init__()
           self.a = a

       def method(self):
           print("method, from class: ", self.__class__.__name__)

   b1 = B("B")
   print(f"b1.b: {b1.b}")
   print(f"b1.a: {b1.a}")
```

```
b1.b: A.b
b1.a: B
```

Yuxarıda iki ədəd sinif: `A` və `B` təyin edirik. `A` sinfinin iki ədəd sinif xüsusiyyəti: `a="A.a"` və `b="A.b"` və nüsxə metodu - `method` var. `A` sinfinin vərəsəsi olan `B`-nin həmçinin `a` xüsusiyyəti və `method` metodu var. `B` sinfinin konstruktorunda nüsxənin özünün `a` adlı xüsusiyyətini yaradırıq, yəni `B` sinfinin nüsxəsinin özünün də `a` adlı xüsusiyyəti olmalıdı. `b1 = B("B")` deyərək `B` sinfinin nüsxəsini yaradırıq və ilk öncə `b1.b` çap edirik. `b` adlı xüsusiyyəti almaq istəyirik, baxaq siniflərə: `b` adlı xüsusiyyət ancaq `A` sinfində təyin edilib, `B` sinfinin özündə yoxdur - odur ki, `b` xüsusiyyətini valideyn sinifdən - `A`-dan miras alacaq, yəni `b1.b` bizə `b`-nin `A`-da təyin olunmuş dəyərini - `"A.b"` verəcək. Belə nəticəyə gəlmək olar ki, əgər müraciət etdiyimiz attribut nüsxənin öz sinfində yoxdursa, onda valideyn sinfində axtarılır. İndi isə, `b1.a` attributuna baxaq: `a` xüsusiyyəti həm `A` sinfində, həm də `B` sinfinin özündə var. Odur ki, valideyn sinfindəki dəyəri deyil, nüsxənin öz sinfinin dəyərini - `"B"` almış oluruq. Bu davranış məntiqlidir, çünki vərəsəlik bir növ övlad sinfi tamamlamaq - çatışmayan attributlarını valideynlərdən almaq üçün istifadə edilir və əgər attribut sinfin özündə varsa, daha valideynlərə nə üçün müracət edək... Eyni şey metodlara da aiddir - hər iki sinifdə `method` adlı metod var: metod `self.__class__.__name__` deyərək nüsxənin sinfinin adını çap edir(`__class__` obyektin sinfini, `__name__` isə sinfin adını qaytarır). Yəni, metod - aid olduğu sinfin adını çap etməlidi. `b1.method()` deyərək metodu çağırsaq: `method, from class: B` alacağıq. Yəni, `B`-nin özündə təyin olunmuş metod çağrılır. Belə çıxır ki valideyn sinfində var olan attributu öz sinfimizdə təyin edərək, valideyn sinfindəki attributu "itiririk", onu "əvəzləmiş" oluruq. Orijinal olaraq, ingilis dilində "overwriting" - yəni, silinərək yenidən yazılma, üzərindən yazılma və sairə kimi tərcümə etmək olar, mən isə bunu "örtülmə" kimi adlandıracam. Beləliklə, valideyn sinfindəki eyni addlı attributu övlad sinfində yaratdıqda - valideyn sinfinin həmin attributunu üzərini yenisi ilə "örtmüş" oluruq.

Adların axtarış zənciri: yuxarıda aydınlandıq ki python nüsxənin attributunu axtaranda əvvəlcə nüsxənin öz sinfində, daha sonra isə valideyn sinfində axtarır. Bəs əgər, bir-birindən "törəyən" siniflər zəncirimiz olsa? Belə bir atma koda baxaq:

```
class A:
   a = 'a'

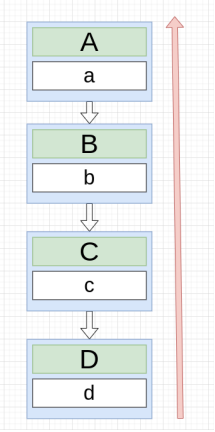
class B(A):
   b = 'b'

class C(B):
   c = 'c'

class D(C):
   d = 'd'

d = D()
print(f"d.d: {d.d}")  # d.d: d
print(f"d.c: {d.c}")  # d.c: c
print(f"d.b: {d.b}")  # d.b: b
print(f"d.a: {d.a}")  # d.a: a
```

Vərəsəlik zəncirini təqribən bu cür təsvir etmək olar:



Siniflər yuxarıdan-aşağıya doğru bir-birini tamamlayır. `A` sinfi `B`-ni, `B` -> `C`, `C` -> `D`. Əgər `d.c` desək, `c` attributu `C` sinfindən alınacaq, əgər `d.a` desək, python şəkildəki qırmızı ox istiqamətində bütün valideyn siniflərində `a` attributunu axtarmağa başlayıb,nəhayət `A` sinfində tapacaqdı, yəni `d` nüsxəsi öz `a` attributunu `A` sinfindən alacağ. Əgər, `C` sinfində də `a` attributu olsaydı, o zaman `C` sinfinin `a` attributunu tapmış olardıq, çünki vərəsəlik ağacında aşağıdan yuxarı qalxdıqda `C` sinfi, `A` sinfindən daha yaxındır. Bir növ attribut lazım olduqda özümüzdən başlayıb, ən yaxın qohumlardan - uzaq qohumlara qədər gedirik.

Nəqliyyat-maşın məsələsi: demək olar ki bütün proqramlaşdırma kitablarında OYP mövzusuna gəldikdə nümunə kimi "Vehicle-Car" (Nəqliyyat-Maşın) kimi göstərilir, biz də bu kitabda bu tendensiyaya sadiq qalacağıq. Məsələnin əsas məğzi - proqramlaşdırma dilində siniflərin vərəsəlik anlayışını və attributların necə örtülməsini göstərməkdir. Bir ədəd əsas sinfimiz - `Vehicle` var, sinif içərisində istifadə edəcəyimiz nəqliyyat vasitələri üçün ortaq məntiqi özündə saxlayacaq. Daha sonra, bu sinfin vərəsələri olan `Car` - maşın, `Boat` - qayıq, `Plane` - təyyarə sinifləri yaradırıq.

```
class Vehicle:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

    def display_info(self):
        return f"{self.year} {self.make} {self.model}"

    def start(self):
        return "Vehicle is starting..."

    def stop(self):
        return "Vehicle is stopping..."

class Car(Vehicle):
    def __init__(self, make, model, year, num_doors):
        super().__init__(make, model, year)
        self.num_doors = num_doors
```

```
    def display_info(self):
        return f"{super().display_info()}, {self.num_doors} doors"

    def drive(self):
        print(self.start())
        print("Car is driving...")
        print(self.stop())

class Boat(Vehicle):
    def __init__(self, make, model, year, length):
        super().__init__(make, model, year)
        self.length = length

    def display_info(self):
        return f"{super().display_info()}, {self.length} feet long"

    def float(self):
        print(self.start())
        print("Boat is floating...")
        print(self.stop())

class Plane(Vehicle):
    def __init__(self, make, model, year, max_altitude):
        super().__init__(make, model, year)
        self.max_altitude = max_altitude

    def display_info(self):
        return f"{super().display_info()}, {self.max_altitude} max altitude"

    def fly(self):
        print(self.start())
        print("Plane is flying...")
        print(self.stop())
```

Deməli, yuxarıdakı kodda nə baş verir: valideyn sinfi olan `Vehicle` bizdən üç ədəd parametr alır: `make`, `model` və `year`. Əlavə olaraq `display_info`, `start` və `stop` adlı üç metodu var. Bu sinfin vərəsələri valideyn `Vehicle` sinfinin parametrlərini `super().__init__(make, model, year)` sətrində ötürərək bir "nəqliyyat vasitəsi" olurlar. Deməli, övlad siniflərdə də `Vehicle` - nəqliyyata xas olan xüsusiyyət və metodlar əlçatan olacaq. Hər bir övlad sinfi öz içərisində `display_info` metodunu örtür və bu o deməkdir ki, hər bir sinif üçün öz `display_info` metodu işə düşəcək. Əlavə olaraq, hər bir övlad sinfin öz "işlədici" metodu var: `Car` üçün bu - `drive` (sürmək), `Boat` üçün - `float` (üzmək), `Plane` üçün isə - `fly` (uçmaq) olacaq. Bu metodlar ilk öncə `start` metodunu çağırır, bu metod övlad siniflərin özlərində olmadığı üçün, valideyn sinfində tapılır və çağrılır. Daha sonra, sinfin öz işlədici metodu çağrılır və sonda valideyn sinfinin `stop` metodu çağrılır:

```
x5 = Car("BMW", "X5", 2022, num_doors = 4)
boing = Plane("Boing", "747", 2020, max_altitude = 100)
searay = Boat("Searay", "SunDancer", 2021, length = 20)
print('---------------------------------------')
print(x5.drive())
print('---------------------------------------')
```

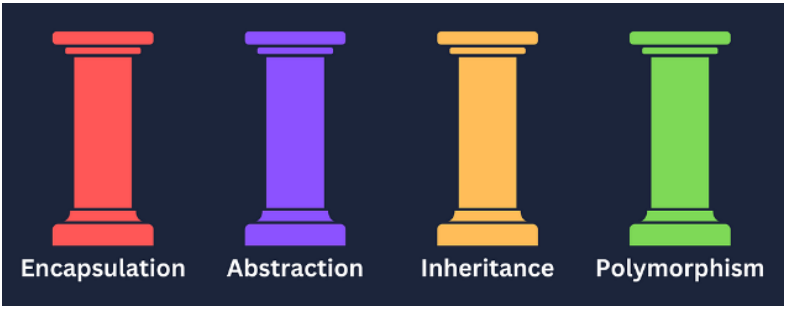
```
print(boing.fly())
print('--------------------------------------')
print(searay.float())
print('--------------------------------------')
```

```
--------------------------------------
Vehicle is starting...
Car is driving...
Vehicle is stopping...
None
--------------------------------------
Vehicle is starting...
Plane is flying...
Vehicle is stopping...
None
--------------------------------------
Vehicle is starting...
Boat is floating...
Vehicle is stopping...
None
--------------------------------------
```

Obyekt yönümlü proqramlaşdırmanı öyrənməyə başlayan kimi, bu işdə daha çox yaradıcı olacağımızı demişdim və bu bölmədə məhz bu haqqda danışacağıq - OYP istifadə edərək necə "yaxşı" kod yaza bilərik. OYP bizə bir sıra imkanlar verir, məntiqi bir sinif içərisində - bir obyekt kimi saxlamaq, vərəsəlikdən istifadə edərək bir sinfi digəri üzərində genişləndirmək və sairə. Paradiqma - sözü proqramlaşdırmada ümumi qəbul olunmuş, proqram kodu yazılmasına dair fikir və yanaşmalar məcmusudur. Yəni, on illiklərdir ki proqramçılar kod yazır və bu proqramçı "kruqlarında" müəyyən standart, qaydalar və yanaşmalar formalaşıb. Bu yanaşma, bu paradiqmaya əsasən baxacağıq: OYP istifadə edərək nələr edə bilərik və necə edə bilərik.

8.4. Ümumi müddəalar

Obyekt yönümlü proqramlaşdırmada dörd əsas konsepsiya var:



- ***Vərəsəlik*** - bir sinif digər sinfin xüsusiyyət və metodlarına yiyələnir, bu barədə artıq müəyyən qədər danışmışıq.
- ***Polimorfizm*** - metod və ya operator - tətbiq olunduğu obyektin tipindən asılı olaraq, özünü fərqli cürə apara bilər. Buna bariz nümunə kimi "+" əməlini göstərmək olar - bu operator ədədlər üçün toplama, sətrlər üçün isə konkatenasiya ifadə edirdi.
- ***İnkapsulyasiya*** - sinfin yalnız daxili məntiqi üçün nəzərdə tutulan xüsusiyyət və əməllər sinfin xaricindən görünməməlidir, istifadəçi "çöldən" içərinin mühüm - içərinin məntiqi üçün nəzərdə tutulan xüsusiyyət və metodları görməməlidir.
- ***Abstraksiya*** - nisbətən böyük siniflər elə qurulmalıdır ki, istifadəçi bu siniflərlə işləyəndə içəridə baş verən bütün əməliyyatları bilmək və udarə etmək məcburiyyətində olmasın. Biz hələki müddəaların adlarını sadaladıq, növbəti başlıqdan başlayaraq, hər biri haqqında ayrıca danışacağıq. Bu müddəalar əzbərlənilməcək və ya hərfbəhərf riayət ediləcək bir şey deyil, zamanla - kod yazaraq, işləyərək təkminləşdirilən bir şeylərdir. Mənim burada əsas missiyam - sizlərə bu müddəalar arxasında dayanan məntiqi çatdırmaq olacaq. Proqramlaşdırma dilləri haqqında yazılan kitabların əksəriyyətində, OYP haqqında danışmağa başlayan kimi, elə bu dörd müddəadan başlayırlar. Ancaq, bu kitabda biz "fəlsəfəyə" dalmadan sinif-nüsxə-metod-xüsusiyyət və hətta vərəsəliyi belə müyyən dərəcədə öyrəndik və fəlsəfəyə başlamağa tam haqqımız çatır.

8.5. Vərəsəlik - II hissə

Çox vərəsəlik və MRO

Pythonda bir sinfin, birdən çox valideyn sinfi ola bilər. Yəni, `A` adlı sinif `A0`, `A00` adlı iki sinifin vərəsi ola bilər və bu zaman `A` sinfi belə elan ediləcək:

```
class A(A0, A00):
    ...
```

Yəni, bu dəfə bütün valideyn sinifləri sinfin başlığında sadalayırıq. Bəs bu necə işləyir? İlk öncə belə bir siniflər zəncirinə baxaq:

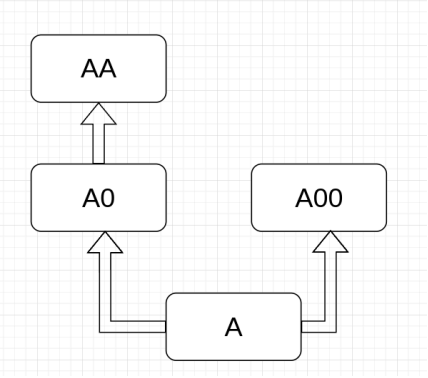
```
class AA:
    a = "A0.a"
    aa = "A0.aa"
    x = "aa"

class A0(AA):
    a = "A0.a"
    a0 = "A0.a0"

class A00:
    a = "A0.a"
    a00 = "A0.a00"
    x = "aa"

class A(A0, A00):
    a = "A.a"
```

Bu zənciri belə vizuallaşdırmaq olar:



Bizim üçün vərəsəlik nə edirdi - biz valideyn siniflərin attributlarından istifadə edirdik. Əgər attribut öz sinfimizdə varsa onu götürürdük, yox idisə yuxarı - valideyn sinifdə axtarırdıq(daha doğrusu python axtarırdı). Yuxarıdakı nümunədə isə, göründüyü kimi, budaqlanma var, yəni artıq yalnız aşağıdan-yuxarı deyil, soldan-sağa doğru da axtara bilərik. Bəs hansı ardıcıllıqda? Ümumiyyətlə, bu ardıcıllığı təyin edən qayda necə adlanır? Pythonda "hansı metodun, adın çağrılmalı olduğunu" tapan mexanizm *Method Resolution Order*, və ya qısaca MRO adlanır və belə çalışır: sinfin soldan sağa bütün valideyn sinifləri bir-bir götürülür və götürülən sinif boyunca yuxarıya doğru attribut axtarılır. Yuxarıdakı nümunədə iki budaq var: `A0` və `A00`. MRO alqoritmi ilk öncə `A0` yolu ilə

yuxarıya doğru gedəcək, axtarılan attribut tapılmasa - `A00` budağına keçəcək. Əgər budaqların özlərində də şaxələnmə olsa idi, eyni qayda budaqların özlərinə də şamil olunardı, ancaq gəlin nümunələri qəlizləşdirməyək :) .

Gəlin ən aşağıda dayanan `A` sinfinin bir nüsxəsini yaradaq:

```
i = A()
```

Indi isə `i` nüsxəsi üzərindən, `a` , `aa` , `a0` və `a00` attributlarını çap etməyə çalışaq:

```
print(f"i.a: {i.a}")        # A.a
print(f"i.a0: {i.a0}")      # A0.a0
print(f"i.aa: {i.aa}")      # A0.aa
print(f"i.a00: {i.a00}")    # A0.a00
```

- `sətr 1` : `a` attributu nüsxənin öz sinfində var olduğu üçün elə `A.a` alırıq
- `sətr 2` : `a0` attributu axtarılır, sol budaqdan axtarmağa başlayırıq. `A0` sinfində axtarılan attribut tapılır.
- `sətr 3` : `aa` attributunu axtararkən, sol budaqla yuxarı qalxır - `A0` sinfində belə attribut tapılmır, bir mərtəbə də qalxır və çatır `AA` sinfinə. `AA` sinfində axtarılan `aa` attributu tapılır.
- `sətr 4` : `a00` attributu axtarılır, sol budaqla əvvəlcə `A0` , daha sonra isə `AA` sinfində axtarılır amma tapılmır, bununla da sol budaqda axtarış bitir və keçir sağ budağa. Sağ budaqda `A00` sinfində attribut tapılır və alırıq `A00.a00` . İndi isə `x` attributuna baxaq. `x` sxemimizin iki sinfində var: sol budağın zirvəsi sayılan `AA` və sağ budağın ilk və son sinfi olan `A00` içərisində. Əgər `i.x` almağa çalışsaq:

```
print(f"i.x: {i.x}")    # aa
```

Yəni, sol budağın zirvəsində tapırıq. Çünki, mahiyyət etibarilə, sol budağın zirvələri bizim üçün sağ budağın istənilən elementləri daha yaxındır - əvvəlcə sollar, yalnız ondan sonra sağlar.

Əslinə qalsa, praktikada bu cür "əndrabadi" sxemlər nadir hallarda rast gəlinir və daha çox proqramlaşdırma alətləri - kitabxanalar, freymvörklər və sairə yazarkən istifadə edilir. Odur ki, bu söhbətə burda nöqtə-vergül qoyub, davam edək.

8.6. Polimorfizm

Polimorfizm öncə tanış olduğumuz vərəsəlikdən fərqli olaraq, konkret metod və ya kodla başa gələn bir şey deyil, bu daha çox bir növ abstrakt anlayışdır, yanaşmadır. Texniki olaraq, burada görəcəyimiz bütün mexanizmlər vərəsəlik əsasında çalışır, sadəcə ortaya çıxan effekti polimorfizm adlandıracağıq. Beləliklə, nədir Polimorfizm? Polimorfizm - eyni adlı metodun və ya operatorun, tətbiq edildiyi obyektin tipindən asılı olaraq davranışını dəyişməsidir. Vərəsəliyi yada salaq: deyək ki A -> B -> C istiqamətində vərəsə siniflər var(B A-nın, C isə B-nin vərəsəsidir) və hər bir sinfin öz `m()` metodu var. Belə olan təqdirdə bildiyiniz kimi A-nın nüsxəsi üçün A-nın `m` metodu və B sinfi üçün B, C sinfi üçün isə C-nin öz sinfinin metodu çağrılacaq. Öncə tanış olduğumuz MRO prinsipiş. Bəs bu kənardan necə görünür: bir `m()` metodu var və müxtəlif metodlarda müxtəlif cür işləyir. Gələcəkdə görəcəyik ki operatorlar da(+, -, % və s.) da əslində pərdəarxası çağrılan metodlardır,

hansıları ki biz öz siniflərimizdə reallaşdırıb və vərəsəliklə ötürə bilərik, odur ki operatorlar da adi metodlar kimi MRO qaydasına tabedir.

Vektor və Skalyar misalı - 1

Yuxarıda yazılanları daha yaxşı həzm etmək üçün təklif edirəm ki iki ədəd sinif yazaq:

1. `Skalyar` sinfi - özündən adi bir ədəd ehtiva edəcək, int və ya float tipində. Sinif içərisində toplama və çıxma üçün uyğun olaraq `add` və `sub` metodları reallaşdıracağıq ki skalyarları bir-biri ilə toplayıb-çıxa bilək.
2. `Vector` sinfi - özündən cəbri vektoru ehtiva edəcək, bizim üçün sadəcə bir ölçülü massiv olacaq, məsələn ədədlərdən ibarət adi siyahı. Burada da sub və add metodları olacaq. Deyək ki, vektor ancaq eynitipli ədədlərdən ibarət olmalıdır. Əlavə olaraq, vektor barədə bəzi məlumatları(uzunluq, tip) saxlayacaq xüsusiyyətlər də yaradacağıq. Sonda elə etməliyik ki, `vektor.add(skalyar)` , `vektor.add(vektor)` , `skalyar.add(vektor)` və sairə yazaraq vektorları öz aralarında və ya skalyar ədədlə toplayıb-çıxa bilək.

Hazırlıq: deyək ki `v1 = [1,2,3]` , `v2 = [4,5,6]` vektorları və `k=5` skalyarı var. Vektorları toplamaq üçün onların uzunluqları(içərilərindəki element sayı) eyni olmalıdır. Toplama zamanı isə sadəcə mövqelərinə görə uyğun elementləri cəmləyirik: `v1 + v2 = [(1+4), (2+5), (3+6)] = [5, 7, 9]` . Vektorun ilə skalyarı toplamaq isə, vektorun hər bir elementinin üzərinə skalyarı gəlmək deməkdir: `v1 + k = [(1+5), (2+5), (3+5)] = [6, 7, 8]` . PS, xətti cəbrə filan dalmayacağıq yuxarıda yazılanlar bizə tamamilə bəs edir.

Skalyar

Əvvəlcə Skalyar sinfinin reallaşdırılmasından başlayaq. Əslinə baxanda skalyar sinfinin nüsxəsi sadəcə bir ədəd olmalıdı. `add` və `sub` metodlarından savayı, validasiya(doğruluğu yoxlama) üçün bir `validate` metodu da yazacağıq, çünki yoxlamaq lazımdır, skalyar içərisinə hər hansı sətr filan yazılmayıb ki. Skalyar və Vektor sinifləri vahid bir `MathStructure` adlı sinifdən törəyəcək, bu sinifdə `Scalar` və `Vector` siniflərinin ortaq metodları olan `add` və `sub` qismən reallaşdırılacaq - sadəcə metodların başlıqlarını yazacağıq. Əlavə olaraq - `shape` xüsusiyyəti riyazi strukturun(Skalyar və ya Vektor) formasını qaytaracaq. Skalyar üçün `shape=0` olur, vektor üçün isə `(len(v), )` olur, yəni `(1,3,3)` vektorunun forması(`shape`) olacaq: `(3, )`

```
from numbers import Number

class MathStructure:
    shape = None
    def add(self, other):
        pass

    def sub(self, other):
        pass

    def validate(self):
        pass


class Scalar(MathStructure):
    shape = 0
```

```
    def __init__(self, value: Number) -> None:
        self.value = value
        self.validate()

    def validate(self):
        """ raise error if value is not a number """
        if not isinstance(self.value, Number):
            raise TypeError(f"Scalar value must be numeric, not
{type(self.value)}")

    def add(self, other):
        if isinstance(other, Scalar):
            return Scalar(self.value + other.value)
        elif isinstance(other, Number):
            return Scalar(self.value + other)
        else:
            return None

    def sub(self, other):
            if isinstance(other, Scalar):
                return Scalar(self.value - other.value)
            elif isinstance(other, Number):
                return Scalar(self.value - other)
            else:
                return None

s1 = Scalar(1)
s2 = Scalar(2.0)
s3 = s1.add(s2)
s4 = s3.sub(10)
print(s3.value, s4.value)   # 3.0 -7.0
print(s3.shape, s4.shape)   # 0 0
```

Pythonda bütün ədədlər `numbers.Number` sinfinin törəmələridirlər, odur ki, ədəd tipini `Union[int, float]` kimi göstərmək əvəzinə, sadəcə `Number` yazıram. `Scalar` sinfimizin mərkəzində ədəd - `value` xüsusiyyəti dayanır, konstruktora ötürülən bu dəyəri `validate` metodu vasitəsilə elə konstruktor içərisindəcə yoxlayırıq, odur ki skalyar içərisində yanlış dəyər ola bilməz. Toplama- `add` və ya çıxma- `sub` zamanı əməliyyatın ikinci tərəfi - `other` olur, və əgər bu dəyər başqa bir `Scalar` nüsxəsi olarsa - skalyarların dəyərlərini toplayıb/çıxıb nəticəsindən yeni bir skalyar yaradıb geriyə qaytarırıq, yox - `other` `Skaler` yox, hər hansı başqa bir ədəd olarsa - o zaman skalyarımızı bu ədədlə toplayıb/çıxıb yenə nəticədən bir skalyar yaradıb geriyə qaytarırıq. Çünki, `Scalar` ilə hər hansı əməliyyat apararkən, nəticədə də həmçininn `Scalar` almaq istəyirik.

> ⚠ **Diqqət!**
>
> Əgər `Scalar` nüsxələrini `+` operatoru ilə toplamağa cəhd etsəz o zaman xəta alacaqsınız.
>
> ```
> >>> Scalar(3) + Scalar(3)
>    `TypeError: unsupported operand type(s) for +: 'Scalar' and 'Scalar'
> ```

Məsələ burasındadır ki, python hələki `Scalar` sinfinin nüsxələri üçün `+` operatorunun >necə işləməli olduğunu bilmir. `+`, `-`, `%` və sairə kimi standart əməliyyatları öz siniflərimizdə tanıtmaq, reallaşdırmaq üçün *dunder* və ya *magic* metodlarla tanış olmalıyıq. Bu metodlar `__` ilə başlayır və bitir. Onlarla biz yaxın gələcəkdə tanış olub, `Scalar` sinfimizi daha real və rahat edəcəyik.

Vektor

Vektor sinfinin mərkəzində - eynitipli ədədlərdən ibarət massiv olacaq, massiv dəyişməyəcəyiyi üçün - pərdəarxasında yazılardan istifadə edəcəyik. Skalyarlarla olduğu kimi, buda da `add` və `sub` metodları olacaq.

```
class Vector(MathStructure):
    shape = (None, )
    def __init__(self, values: list) -> None:
        self.values = values
        self.validate()
        self.shape = (len(self.values),)
        self.values = self.as_tuple()

    def as_tuple(self)->tuple:
        return tuple(self.values)

    def validate(self):
        """ raise error if values is not list """
        # 1. validate `values`
        if not isinstance(self.values, (list, tuple)):
            raise TypeError(f"Vector values must be list or tuple, not
{type(self.values)}")
        # 2. validate items in `values`
        if len(self.values) > 0:
            first_type = type(self.values[0])
            for item in self.values:
                if not isinstance(item, first_type):
                    raise TypeError(f"Vector values must be all same type.")

    def add(self, other):
       if isinstance(other, Vector):
           if self.shape == other.shape:
               return Vector([x + y for x, y in zip(self.values,
other.values)])
           else:
               return None
       elif isinstance(other, Number):
           return Vector([x + other for x in self.values])

       elif isinstance(other, Scalar):
           return Vector([x + other.value for x in self.values])
       else:
           return None

   def sub(self, other):
```

```
        if isinstance(other, Vector):
            if self.shape == other.shape:
                return Vector([x - y for x, y in zip(self.values,
other.values)])
            else:
                return None

        elif isinstance(other, Scalar):
            return Vector([x - other.value for x in self.values])

        elif isinstance(other, Number):
            return Vector([x - other for x in self.values])
        else:
            return None

v1 = Vector([1, 2, 3])
v2 = Vector([4, 5, 6])
v3 = v1.add(v2)
v4 = v1.sub(1)

print(f"v1.add(v2): {v3.values}")
print(f"v1.sub(1):  {v4.values}")
```

```
v1.add(v2): (5, 7, 9)
v1.sub(1):  (0, 1, 2)
v1.shape: (3,)
v2.shape: (3,)
```

Həm skalyarlar, həm də vektorlar üçün `add` metodu var və bu metod fərqli tipli obyektlər üçün fərqli tərzdə çalışır - skalyarlar üçün skalyar, vektorlar üçün vektor qaytarır. Bu polimorfizm nümunəsidir. Dunder metodlarla tanış olandan sonra bu siniflərə bir daha əl gəzdirib, daha başadüşülən vəziyyətə gətirəcəyik.

8.7. İnkapsulyasiya və Abstraksiya

İnkapsulyasiya

Bəzən sinfin daxili tələbatı üçün metodlar yazmalı oluruq. Məsələn, öncəki `Vector` sinfində `validate` metodu reallaşdırmışdıq, bu metod konstruktor içərisində çağrılaraq ötürülən verilənlərin doğruluğunu yoxlayırdı, yəni məhz daxili yoxlama üçün nəzərdə tutulurdu. Əgər `v = Vector((1,2,3))` deyib bir `v` obyekti yaratsaq, tam rahatlıqla `v.validate()` deyib nüsxənin `validate` metodunu çağıra bilərik. Düzdür, bu zaman heç bir pis şey baş verməyəcək, amma bu cür metodların sinfin çölündən çağrılması o qədər də düzgün deyil. Bəs birdən metod nəsə vacib, yersiz istifadə zamanı proqramı çökdürəcək bir şey edirsə, o zaman nə edək? Metodun çöldən istifadəsinə necə məhdudiyyət tətbiq edək? Bunun üçün bəzi proqramlaşdırma dillərində metodun "görünmə fəzasını" tənzimləyən *public*, *private*, *protected* kimi açar sözlər var, amma pythonda bunlar **yoxdur**, onun yerinə metodları adlandırarkən *xüsusi qaydalara* riayət etmək lazımdır. Bu qaydalar:

1. **Bir ədəd '_' ilə:** metodun adı bir ədəd `_` simvolu ilə başlayırsa, bu o deməkdir ki metod daxili istifadə üçün nəzərdə tutlub və sinfin çölündə istifadə edilməməsi "məsləhət görülür". Niyə məhz "məsləhət görülür" - çünki istifadəçi bunlara rəğmən yenə də bu metodlardan çöldə(sinfin xaricində) istifadə edə bilər. Yəni, bu adlandırma məhdudiyyət xarakteri daşımır, sadəcə buna eyham vurur.
2. **İki ədəd '_' ilə:** metodun adı iki ədəd `_` simvolu ilə başladıqda isə metod çöldən əlçatmaz olur. Əgər bir ədəd `_` ilə bu sadəcə tövsiyyə idisə, iki ədəd `_` ilə bu faktiki məhdudiyyətdir. Qısası, əgər ad bir ədəd `_` ilə başlayırsa - bu adı çöldən çağırmaq olar, amma çağırmasaq yaxşıdır. Əgər iki ədəd `_` ilə başlayırsa - o zaman həmin adı çöldən çağıra bilmirik. Baxaq:

```
class Student:
    def __init__(self, name: str, surname: str):
        self.name = name
        self.surname = surname
        self.email = self.__build_email()

    def __build_email(self) -> str:
        return f"{self.name}.{self.surname}@domain.com"

    ali = Student("ALi", "Aliyev")
    ali.__build_email() # error
    # AttributeError: 'Student' object has no attribute '_build_email'
```

Yuxarıdakı nümunədə `Student` sinfinin gizli `__build_email` metodu var. Metod çağrılarkən `name` və `surname` xüsusiyyətlərindən istifadə edərək nüsxənin `email` xüsusiyyətini yaradır. Prosess konstruktorun içərisində başlandığı üçün, nüsxənin email ünvanı avtomatik olaraq yaradılır. Yəni, kənardan daha `email.build_email()` kimi çağırmağın mənası yoxdur, email yaradılması prosesi gizlidir - sinfin öz "daxili" işidir. Gizli olduğu üçün də `ali.__build_email()` bizə `AttributeError` xətası alırıq. Bu xəta obyektin mövcud olmayan attributunu almağa çalışarkən baş verir. `print(dir(ali))` deyərək `ali` obyektinin bütün mövcud attributlarını çap edə bilərik. Bunu etsək, çoxsaylı attributların arasında(sizə tanış olmayan xeyli əndrabadi adlar ola bilər, tələşlanmağa ehtiyac yoxdur) belə bir attribut da görərik: `_Student__build_email`

```
print(dir(ali))
# ['_Student__build_email', ..., 'email', 'name', 'surname']
```

Əgər bu `_Student__build_email` adını obyektin önündə yazaraq çağırmağa çalışsaq, görərik ki bu əslində elə bizim gizli `__build_email` metodudur! Yəni sinfin xarici üçün gizli metodların adı dəyişir və belə bir ada sahib olur: `_<sinfin_adı>__<metodun_adı>` . Bu o deməkdir ki gizli attributları bu dəyişdirilmiş adla almaq mümkündür, amma bu o demək deyil ki belə etmək lazımdır. Əgər attrbutun çöldən çağrılması məntiqi sındıracaqsa - attributu gizli etmək olar, yox, attribut təhlükəsizdirsə və xaricdən istifadə üçün nəzərdə tutulmayıbsa `_` ilə obyektin daxili istifadə üçün olduğunu göstərə bilərik.

Abstraksiya

Mövzuya belə dəxlisiz bir sualla başlayacam: sizcə, maşın sürücülərinin neçə faizi mühərrikdəki silindrlərin iş prinsipi ilə tanışdı? Məncə, o qədər də çox deyil, ən azından mən onların arasında deyiləm. Ancaq, bunu bilməməyim mənim maşın sürməyimə mane olmur! Bu o deməkdir ki, bəzən

nədənsə normal istifadə etmək üçün, həmin şeyin bütün xırdalıqlarını bilmək lazım deyil. Mürəkkəb proqram məntiqi də bu cürdür - arxitekturanı elə qurmağa çalışırlar ki, proqramın hər bir kod hissəsinin öz müstəqil, vahid məntiqi olsun. Məsələn, `Vector` sinfi var, hansı ki sadə riyazi vektoru təcəssüm etdirir. Bu sinfin `add()` metoduna başqa bir vektor ötürdükdə, metod geriyə iki vektorun cəmini qaytarır. Koda ilk dəfə baxan bir istifadəçi üçün, iki vektoru cəmləmək üçün `add` metodunu tam başa düşmək lazım deyil, sadəcə metodu çağırır, vəssalam. Çünki, vektorların toplanması məntiqi "gözdəniraq" çəkildə reallaşdırılıb, qurdalayıb `add` vektorunun hardasa pərdəarxası istifadə etdiyi digər metodları üzə çıxarmaq olar, amma adi istifadəçi üçün bunun heç bir önəmi yoxdur, adi istifadəni bütün bunlarla yüklənmir. Yəni, `add` metodu özündə daha böyük, qəliz bir məntiqin "sadələşdirilmiş", "məvhumi" və ya terminoloji dildə desək - abstrakt formasını daşıyır. İlk baxışdan, bu bir növ inkapsulyasiyaya bənzəyir, ancaq o deyil. İnkapsulyasiya - daha çox təhlükəsizlik, attributlara girişin idarə edilməsi üçün istifadə edilir. Abstraksiya isə, daha çox proqram məntiqinin istifadəçi üçün daha rahat qurulmasını təmin etməyə çalışır. Yenə də, maşın sürmək üçün mühərriyi tam tanımağa ehtiyac yoxdur. Sadə sürücü üçün maşını sürmək üçün salondakı düymələr, sükan və sürət qutusu keçiricisi bəs edir. Maşının "kapot"-nun altındakılar ustalara qalsın. Düzdür, maşın yarıyolda qırıldıqda adi istifadəçi başını o "kapot"-un altına salmalı olur, amma proqram məsələsində hər şey bir qədər fərqlidir.

Vector v2

Gəlin vektor sinfinə bir balaca əl gəzdirək. Əvvəla, sinfə `length` adlı xüsusiyyət əlavə edək, adından da göründüyü kimi, vektorun uzunluğunu ifadə edəcək. Daha sonra `get_stats` adlı metod əlavə edək. Metod dörd sadə statistik ölçünü hesablayacaq: ədədi orta(mean), minimal element(min), maksimal element(max) və aralıq(range) - maksimal çıxılsın minimal element. Kitabı kodla doldurmamaqçün, öncəki `Vector` sinfinin ancaq dəyişikliyə uğramış hissəsini qeyd edəcəm.

```
class Vector(MathStructure):
    shape = (None, )
    def __init__(self, values: list) -> None:
        self.values = values
        self.validate()
        self.length = self._get_len()
        self.shape = (self.length,)
        self.values = self.as_tuple()

    def _get_len(self) -> int:
        return len(self.values)
    ....................................
    def _get_mean(self) -> float:
        return round(sum(self.values) / self.length, 3)

    def _get_min(self)->Number:
        return min(self.values)

    def _get_max(self) -> Number:
        return max(self.values)

    def _get_range(self) -> Number:
        return max(self.values) - min(self.values)

    def get_stat(self) -> dict:
```

```
        """ calculate basic statistics """
        mean = self._get_mean()
        min_ = self._get_min()
        max_ = self._get_max()
        range_ = self._get_range()
        return {'mean': mean, 'min': min_,
                'max': max_, 'range': range_}

v1 = Vector([1, 2, 3])
v2 = Vector([4, 5, 6])
v3 = v1.add(v2)
v4 = v1.sub(1)

print(f"v1.add(v2): {v3.values}")
print(f"v1.sub(1):  {v4.values}")
print(f"v2.get_stat(): {v2.get_stat()}")
```

```
v1.add(v2): (5, 7, 9)
v1.sub(1):  (0, 1, 2)
v2.get_stat(): {'mean': 5.0, 'min': 4, 'max': 6, 'range': 2}
```

Mövzu nöqteyi cəhətdən yeni heç bir şey yoxdur, sadəcə bir `get_stat()` metodu əlavə edildi, metod hər bir göstəricini hesablamaq üçün müvafiq `_get_...()` adlı köməkçi metodu çağırır. Əslində, bu köməkçi metodları məntiq etibarilə adi şəkildə `_` olmadan da yazmaq olardı, çünki xaricdən vektorun ədədi ortasını almaq məntiqli və faydalı bir iş ola bilər. Amma, təsəvvür edək ki, biz bunu istəmirik. Bizim üçün bu metodlar sadəcə `get_stat` içərisində istifadə etmək üçün lazımdır. Bunun üçün onları tam olaraq gizlətməsək də, `_` ilə daxili metod kimi işarə edirik. Bu məsələnin inkapsulyasiya tərəfi idi. Digər tərəfdən, istifadəçi bütün statistik məlumatları `get_stat` metodu içərisindən alır. Metodun hansı əlavə metodlardan istifadə etməsi istifadəçi üçün heç bir fərq kəsb etmir. Sadəcə, hər bir kiçik hissə öz işini icra edir. Bunu da bir növ abstraksiya jesab etmək olar. Əslində, bu nümunələr barmaq üzərində - sadələşdirilmiş, sadəcə ümumi mənzərə haqqında təsəvvür yaratmağa xidmət edir.

Bəs sonra??

Beləliklə, bu bölmədə sizinlə OYP dünyasının üzərində dayandığı dörd fil - vərəsəlik, polimorfizm, inkapsulyasiya və abstraksiya haqqında danışdıq. Öncədən də dediyim kimi, bu yanaşmalar bir növ yaradıcı xarakter daşıyır və harda nəyin və necə tətbiq olunmalı olduğu barədə heç bir dəqiq qaydası yoxdur. Bunlar sadəcə yaxşı kod, yaxşı arxitektura adlı Olimp zirvəsinə aparan cığırlardır və bu cığırları tapmaq üçün kifayət qədər gəzmək lazımdır. Analogiyanı kənara atsaq, bu paradiqmaları əjdaha səviyyəsində mənimsəmək üçün, əjdaha ömrü qədər(çox-çox) kod yazmaq və oxumaq lazımdır. Odur ki, təcrübə və əzmlə özü formalaşacaq.

8.8. Xüsusi metodlar

Bizim yarada biləcəyimiz sinif nüsxələri ilə, Pythonın özünün integer sinif nüsxəsi arasında hansı fərqlər var? int tipli obyektləri - tam ədədləri toplaya, çıxa, üzərində müxtəlif əməliyyatlar aparmaq olar. Bizim yaratdığımız nüsxələr isə sadəcə sinif daxilində təyin etdiyimiz metod və xüsusiyyətlərlə məhdudlaşır. Yadınızdadırsa, öncəki başlıqda Vector sinfinin nüsxələri üçün toplama əməlini reallaşdırmaq üçün `add()` metodu yazmışdıq. Ancaq, vektorları `vektor_1 + vektor_2` kimi yazmaqla toplaya bilmirik. Niyə? Çünki, Python bilmir - `+` işarısi bizim `Vector` sinfi üçün necə işləməlidir, nə iş görməlidir... Buna görə, biz özümüz təyin etməliyik - bizim sinfin nüsxələri üçün `+`, `-`, `*` və sairə operatorlar necə işləməlidir. Bəs bunu necə təyin edək, necə edək ki, `Vector` nüsxəsindən sonra `+` operatoru yazılarsa, python təqribən `add` metodundakı məntiqlə vektorları toplasın?

Xüsusi Metodlar: Pythonda demək olar ki hər bir operatora bağlı olan xüsusi bir metod adı var. Məsələn `+` operatoru üçün bu metodun adı `__add__()` olacaq. Hər hansı `A` sinfinin nüsxəsi olan `a` üçün `a + x` yazdıqda, `A` sinfində `__add__()` adlı metod axtarılır. Belə bir metod çağrılır və metodun geriyə qaytaracağı nəticə toplamanın nəticəsi olacaq. Hər opertaorun öz bu cür metodu və qəbul etdiyi parametr sayı var. Məsələn, haqqında danışdırğımız `__add__` metodu üçün metodun başlığı belə olur:

```
class A:
    def __init__(self):
        ...
    def __add__(self, other):
       ...
```

Burada, `other` aparametri `+` operatorunun sağında dayanan obyektdir və avtomatik olaraq ötürülür. Yəni biz, `a + x` deyib çağırdıqda bir növ `a.__add__(other=x)` kimi yazmış oluruq. Əgər obyektin sinfinin müvafiq metodu təyin olunmayıbsa, `TypeError` xətası alırıq. Məsələn, `Vector` sinif nüsxələri olan `v1` və `v2` üçün `v1 + v2` yazsaq, müvafiq `__add__` metodunu reallaşdırmadığımız üçün belə bir xəta alarıq: `TypeError: unsupported operand type(s) for +: 'Vector' and 'Vector'`

Bu metodlar *xüsusi metoadlar*(special methods) və ya *sehrli metodlar*(magic methods) adlanır, əlavə olaraq, `__` ilə əhatə olunduğuna görə *dunder methods* da adlanır. Xüsusi metod kimi adlandırlması məntiqlidir, çünki bu metodlar Python üçün xüsusi məna daşıyır. Gəlin ilk öncə bu metodların bəzilərini sadalayaq:

operator | funksiya	xüsusi metod
`+`	`__add__`
`-`	`__sub__`
`*`	`__mul__`
`/`	`__truediv__`
`%`	`__mod__`
`//`	`__floordiv__`
`**`	`__pow__`
`@`	`__matmul__`

operator | funksiya	xüsusi metod
`>`	`__gt__`
`<`	`__lt__`
`>=`	`__ge__`
`<=`	`__le__`
`==`	`__eq__`
`!=`	`__ne__`
`str(x)`	`__str__`
`int(x)`	`__int__`
`float(x)`	`__float__`
`repr(x)`	`__repr__`
`x(a,b)`	`__call__(a, b)`
`with` girişdə	`__enter__`
`with` sonda	`x.__exit__(exc_type, exc, traceback)`

Xüsusi metodlar daha çoxdur və onların hamısını burda sadalamaq sadəcə kitabda yer israfçılığı deməkdir. Metodları tətbiq etdikcə, tanış olacağıq. Gəlin, ciddi bir şeyə baş vurmazdan əvvəl, sadə nümunələr üzərində xüsusi metodların bəzilərinə baxaq.

```
from numbers import Number
from typing import Union, Self

class Num:
   def __init__(self, value: Number):
       self.__value = value

   def __add__(self, other: Union[Self, Number]):
       if isinstance(other, Number):
           sum_ = other + self.__value
       elif isinstance(other, Num):
           sum_ = self.__value + other.__value
       else:
           raise ValueError(f"Unsupported + for {type(self)} and
{type(other)}")
       return Num(value=sum_)

   def __str__(self):
       return f"Number: {self.__value}"

num_1 = Num(value=1)
num_2 = Num(value=2)
num_3 = num_1 + num_2   # 3
num_4 = num_3 + 10      # 3 + 10
print(f"num_3: {num_3}")
print(f"num_4: {num_4}")
```

```
num_3: Number: 3
num_4: Number: 13
```

Yuxarıda sadə bir `Num` - ədəd sinfi reallaşdırılıb. Sinfin mərkəzində ədədin dəyəri olan `__value` xüsusiyyəti dayanır. `Num` tipli obyektlərini cəmləmək üçün `__add__` xüsusi metodundan istifadə edirik. Belə ki, biz `num_1 + num_2` yazdıqda, python bir növ `num_1.__add__(num_2)` çağırır. Biz isə bu metod daxilində öz məntiqimizi qurmuşuq: `+` operatorunun sağ tərəfindəki obyekt metoda `other` adlı arqument kimi daxil olur. Yoxlayırıq, əgər bu obyekt ədəddirsə - `Number` tiplidirsə(int, float), o zaman bu ədədi soldakı nüsxənin(`self`) `__value` xüsusiyyəti ilə cəmləyirik. Yox, obyekt eyni sinfin - `Num` sinfinin nüsxəsidirsə, o zaman onların `__value` xüsusiyyətlərini toplayıb yeni bir `Num` sinfinin yeni nüsxəsi şəklində geriyə qaytarırıq. Eyni məntiqlə çıxma, vurma və bölmə əməlləri üçün `__sub__` , `__mul__` , `__truediv__` metodlarını da reallaşdırmaq olar. Sinfin `__str__` metodunun sayəsində, `Num` nüsxəsini sətrə çevirməyə çalışdıqda(`str(num_3)` kimi məsələn) bu metodun geriyə qaytardığı sətri alırıq. Nəzərə alsaq ki, `print` içərisində nüsxəni format etdikdə elə sətr formasını almış oluruq, görürük ki `__str__` metodunun nəticəsini almış oluruq. Qısası, `__str__` bizə obyektin string-sətr təsvirini verir. Bu cür, xüsusi metodları reallaşdırdıqca, sinfimizin nüsxələri daha çox pythonın özünün qurma obyektlərinə bənzəyir. Daha bir məsələ `Self` - tip göstəricisidir. Adından da göründüyü kimi - sinfin özünün tipini göstərmək üçün istifadə olunur. `other: Union[Self, Number] - other` parametrinin sinfin özü və ya ədəd tipli olacağına işarə verir.

Vector V3

Gəlin xüsusi metodların sayəsində `Vector` sinfini daha da genişləndirək.

Əlavə edəcəklərimiz:

- Arifmetik əməliyyatlar: toplama, çıxma, vurma və bölmə. Həm vektorlar, həm də ədədlər üçün.
- Müqayisə operatorları. Bu zaman bir növ məntiqi tipli vektor almalıyıq.
- Sətr təsviri. Tam `Vector` sinfi böyük olacağı üçün, kodu hissə-hissə yazacağıq. Sonda, kodun tam formasını linkdən görə biləcəksiniz.

Arifmetik əməliyyatlar:

Hər dörd əməliyyat üçün məntiq oxşar olacaq. Əgər (vektor + vektor) cəmini tapmaq lazımdırsa, əmin oluruq ki bu vektorların uzunluqları bərabərdir(bunu `shape` xüsusiyyəti vasitəsilə də yoxlaya bilərik - əgər bərabərdirsə-cəmləyirik), daha sonra isə, uyğun elementlərini cəmləyib, bu ədədlərdən yeni yeni `Vector` sinfi yaradıb geriyə qaytarırıq. Eynilə artıq reallaşdırdığımız `add()` metodundakı kimi. Eyni şeyi digər əməliyyatlar üçün də edirik. Toplama üçün:

```
...
class Vector(MathStructure):
    ...
    def __add__(self, other):
        if isinstance(other, Vector):
            if self.shape == other.shape:
                return Vector([x + y for x, y in zip(self.values,
other.values)])
            else:
```

```
                raise ValueError(" Vectors must have same shape")
        elif isinstance(other, Number):
            return Vector([i + other for i in self.values])

v1 = Vector([1, 2, 3])
v2 = Vector([4, 5, 6])
v_sum = v1 + v2
v_sum_2 = v1 + 10
print(f"v_sum: {v_sum}")
print(f"v_sum_2: {v_sum_2}")
print(f"v_sum.values: {v_sum.values}")
```

```
v_sum: <__main__.Vector object at 0x798060ccc810>
v_sum_2: <__main__.Vector object at 0x798060ccfc90>
v_sum.values: (5, 7, 9)
```

Gördüyünüz kimi, `print(v_sum)` bizə anlaşılmaz qalır, çünki vektorun sətr təsvirini təyin etməmişik. Gəlin bu məsələni qaydasına qoymaq üçün `__str__` və `__repr__` xüsusi metodlarını reallaşdıraq. `__repr__` obyektin shell-də olan təsviridir. Bəzən `__str__` ilə eyni olur, odur ki biz də `__repr__` içərisindən `__str__` metodunun nəticəsini qaytaracağıq.

```
...
class Vector(MathStructure):
    ...
    def __str__(self):
        return f"Vector: {self.values}"

    def __repr__(self):
        return self.__str__()

v1 = Vector([1, 2, 3])
v2 = Vector([4, 5, 6])
v_sum = v1 + v2
v_sum_2 = v1 + 10
print(f"v_sum: {v_sum}")
print(f"v_sum_2: {v_sum_2}")
print(f"v_sum.values: {v_sum.values}")
```

```
v_sum: Vector: (5, 7, 9)
v_sum_2: Vector: (11, 12, 13)
v_sum.values: (5, 7, 9)
```

Bu başqa məsələ! Çıxma, vurma və bölmə üçün oxşar məntiq olacaq, odur ki daha burda təkrarən yazmayacam. Əlavə olaraq, bölmə zamanı bölənlər arasında 0 olub-olmamasını yoxlayıb xəbərdarlıq verə bilərik, bu yerdə hər şey təxəyyülə bağlıdır.

Müqayisə operatorları:

Vektorlar üçün müqayisə əməliyyatı necə işləyə bilər? Gəlin təsəvvür edək:

`(1, 3, 5, 7) > 4`

Vektoru ədədlə müqayisə etmək - vektordakı bütün elementləri bir-bir həmin ədədlə müqayisə edib məntiqi cavabı (`True` / `False`) müvafiq sıraya yazmaq deməkdir(diqqət, xətti cəbr nöqteyi nəzərdən etdiklərimizin demək olar ki mənası yoxdur. Ancaq, səthi olaraq `pandas` kitabxanası haqqında danışanda görəcəksiniz ki bu məntiqi ordan əkişdirmişəm). Bir növ `(1,3,5,7) > 4 -> (1>4, 3>4, 5>4, 7>4) -> (False, False, True, True)` olacaq. İki vektoru müqayisə etmək isə, müvafiq elementləri müqayisə edib məntiqi cavabı müvafiq sırada yazmaq deməkdir: `(1,3,5) > (-1,4,1) -> (1>-1, 3>4, 5>1) -> (True, False, True)`. Göründüyü kimi, iki vektoru müqayisə etmək üçün, onların uzunluğu bərabər olmalıdır. Hansı operator üçün hansı xüsusi metod istifadə edilməlidir - öncəki cədvəldən görə bilərik. Bizi maraqlandıran `>`, `<`, `>=`, `<=`, `==` və `!=` operatorları olacaq. Kitabda isə, ancaq `>` və `==` üçün göstərəcəm, yenə də kitabda vərəqlərə qənaət etmək üçün, yenə də, tam kod üçün link olacaq.

Vector V4:

```
...
class Vector(MathStructure):
    ...
    # > üçün
    def __gt__(self, other):
        res_list = []
        if isinstance(other, Number):
            for item in self.values:
                res_list.append(item > other)
            return Vector(values=res_list)
        elif isinstance(other, Vector):
            if other.shape != self.shape:
                raise ValueError("Vectors must have same shape")
            else:
                for i in range(len(self.values)):
                    res_list.append(self.values[i] > other.values[i])
                return Vector(values=res_list)
        else:
            raise ValueError(f"Unsupported for {type(self)} and {type(other)}")
    # == üçün
    def __eq__(self, other):
        res_list = []
        if isinstance(other, Number):
            for item in self.values:
                res_list.append(item == other)
            return Vector(values=res_list)
        elif isinstance(other, Vector):
            if other.shape != self.shape:
                raise ValueError("Vectors must have same shape")
            else:
                for i in range(len(self.values)):
                    res_list.append(self.values[i] == other.values[i])
                return Vector(values=res_list)
```

```
        else:
            raise ValueError(f"Unsupported for {type(self)} and {type(other)}")
    ...

v1 = Vector([1, 2, 3])
v2 = Vector([4, 5, 6])
v1_2 = Vector([1, 2, 3])
print(f"v1 > v2: {v1 > v2}")
print(f"v2 > 5:  {v2 > 5}")
print(f"v2 == 5: {v2 == 5}")
print(f"v1 == v1_2: {v1 == v1_2}")
```

```
v1 > v2: Vector: (False, False, False)
v2 > 5:  Vector: (False, False, True)
v2 == 5: Vector: (False, True, False)
v1 == v1_2: Vector: (True, True, True)
```

Göründüyü kimi, istər arifmetik əməliyyatlar üçün, istərsə də müqayisə əməliyyatları üçün məntiq oxşardır, sadəcə vektorun bütün elementlərinə bir-bir müəyyən əməliyyatı tətbiq edirik.

8.9 Metasiniflər (Metaclasses)

> ⚠ **Diqqət!**
>
> Metasiniflər Python-un ən mürəkkəb və çox vaxt yanlış başa düşülən konsepsiyalarından biridir. Əksər Python proqramçıları heç vaxt öz metasiniflərini yazmaq ehtiyacı duymayacaqlar. Bu bölmə daha çox "Python pərdə arxasında necə işləyir?" sualına cavab tapmaq və dilin dərinliklərinə baş vurmaq üçündür. Praktiki olaraq, bu mövzu ancaq digər proqramçılar üçün kitabxanalar yaradan kəslər üçün lazım ola bilər, digər - normal hallarad isə, demək olar ki istifadə edilmir. Odur ki, bu bölməni məhz burda və indi tam olaraq qavramaq məcburiyyətində deyilsiniz.

Siniflər də Obyektdir İndiyə qədər sinifləri obyekt yaratmaq üçün qəliblər kimi tanıyırdıq. Amma Python-da maraqlı bir məqam var: **siniflərin özləri də obyektdir!** Bəs bu obyektləri nə yaradır? Cavab: **metasiniflər**.

Sadə dillə desək:

- Siz `p1 = Person()` yazanda `Person` sinfi `p1` obyektini yaradır.
- Bəs `Person` sinfini nə yaradır? *Metasinif* adlı bir sinif!

Python-da bütün siniflərin susmaya görə metasinifi `type` adlı qurma metasinifdir. Bəli, `type()` funksiyası təkcə obyektin tipini qaytarmır, həm də siniflərin yaradılmasında iştirak edən əsas metasinifdir.

`type` ilə Dinamik Sinif Yaratmaq `type` metasinifini birbaşa çağıraraq dinamik şəkildə siniflər yarada bilərik. `type` üç əsas arqument qəbul edir:

1. `name` : Yaradılacaq sinfin adı (sətr).
2. `bases` : Sinfin vərəsəsi alacağı valideyn siniflərin yazısı (tuple). Boş yazı `()` heç bir sinfin vərəsəsi olmamaq deməkdir (əslində bu zaman pərdəarxası şəkildə `object` sinfinin vərəsəsi olur).
3. `attrs` : Sinfin atributlarını (xüsusiyyətlər və metodlar) saxlayan lüğət (dictionary).

Gəlin, sadə bir `Heyvan` sinfini `type` ilə yaradaq:

```
# Adi yolla sinfi belə yaradırdıq:
# class Person:
#     def __init__(self, name):
#         self.name = name

#     def say_hi(self):
#         return f"Hi, my name is {self.name}."

# Eyni sinfi type ilə yaradaq
def person_init(self, name): # __init__ üçün funksiya
    self.name = name

def person_say_hi(self): # "say_hi" metodu üçün funksiya
    return f"Hi, my name is {self.name}."

# Sinfin atributları lüğəti
person_attrs = {
    "__init__": person_init,
    "say_hi": person_say_hi,
    "email": None # Bir sinif xwsusiyyəti da əlavə edək
}

# type ilə Heyvan sinfini yaradırıq
PersonDynamic = type("PersonDynamic", (), person_attrs)
# "PersonDynamic" - sinfin adı
# () - valideyn sinif yoxdur
# person_attrs - sinfin atributları lüğəti

# İndi HeyvanDinamik sinfinin nüsxəsini yaradaq
ali = PersonDynamic("əli")
print(f'ali.name: {ali.name}')
print(f'ali.email: {ali.email}')
print(f"type(ali): {type(ali)}")
 # Sinfin öz tipi (metasinifi)
print(f"type(PersonDynamic): {type(PersonDynamic)}")
```

```
ali.name: əli
ali.email: None
type(ali): <class '__main__.PersonDynamic'>
type(PersonDynamic): <class 'type'>
```

Onda belə qənaəyə gəlmək olar ki, `class` açar sözü arxa planda əslində `type` metasinifini istifadə edərək sinif obyektlərini yaradır.

Metasiniflərin Yaradılması Əgər `type` bütün sinifləri yaradırsa, bəs biz bu yaratma prosesinə necə müdaxilə edə bilərik? Buna - `type` metasinfindən törəyib, onun `__new__` kimi xüsusi metodlarını yenidən təyin etməklə nail olmaq olar. Yadınızdadırsa, bu xüsusi metod sinfin nüsxəsi yaradılmazdan öncə işə düşürdü. `type` -ın alt sinfində bu metodu `def __new__(mcs, name, bases, attrs)` kimi təyin edirik, burada: `mcs` -metasinif obyektinin özü(metod özlüyündə sinif metodu olduğu üçün, bunu adi siniflərdəki sinif metodlarının `cls` arqumentinə bir tutmaq olar), name - metasinif vasitəsilə yaradılan sinfin adı, bases - yaradılacaq sinfin valideyn sinifləri(vərəsəlik ardıcıllığı ilə), `attrs` - sinfin attributları(metod və xüsusiyyətləri). Bütün bu arqumentlər üzərində bizə lazım olan əməliyyatları aparıb, sonda sinfi yaratmaq üçün valideyn sinfin(`type`) `__new__` metoduna ötürürük. Bu məntiq bir növ dekoratorları xatırlada bilər, sadəcə miqyas bir qədər fərqlidir.

Bir sinfin hansı metasinifdən istifadə etməli olduğunu göstərmək üçün `metaclass` arqumentindən istifadə edirik:

```
class MyMetaClass(type):
   def __new__(mcs, name, bases, attrs):
       print(f"MetaClass __new__ çağrıldı: {name} sinfi yaradılır.")
       print(f"  Valideynlər: {bases}")
       print(f"  Atributlar (ilkin): {attrs}")

       # Bütün atribut adlarını böyük hərfə çevirək (sadə bir nümunə)
       new_attrs = {}
       for attr_name, attr_value in attrs.items():
           if not attr_name.startswith("__"): # Dunder metodlara toxunmayaq
               new_attrs[attr_name.upper()] = attr_value
           else:
               new_attrs[attr_name] = attr_value

       print(f"  Yeni atributlar: {new_attrs}")
       # valideyn sinfə dəyişdirilmiş attributları göndəririk.
       return super().__new__(mcs, name, bases, new_attrs)
```

Demişdik ki, siniflər susmaya görə metasinif kimi `type` -dan istifadə edir. Əgər yaradılan sinfin, bizim öz yazdığımız metasinifdən istifadə etməsini istəyiriksə, o zaman yaradılan sinfin başlığında adlı arqument şəklində `metaclass` arqumentinə metasinif sinif obyektimizi ötürürük:

```
class SomeClass(metaclass=MyMetaClass):
   # belə bir sinif xüsusiyyəti olsun:
   some_prop = 10
   # və belə bir metod:
   def some_method(self):
       return "i am some method from SomeClass"

print("SomeClass nüsxəsi yaradılır...")
some_class_object = SomeClass()
print("some_class_object.: ", some_class_object.SOME_PROP)
print("some_class_object.SOME_METHOD(): ", some_class_object.SOME_METHOD())
```

```
# AttributeError: 'SomeClass' object has no attribute 'some_prop'
print("some_class_object.some_prop: ", {some_class_object.some_prop})
```

```
MetaClass __new__ çağrıldı: SomeClass sinfi yaradılır.
     Valideynlər: ()
     Atributlar (ilkin): {'__module__': '__main__', '__qualname__':
'SomeClass', 'some_prop': 10, 'some_method': <function SomeClass.some_method at
0x796ae05a2340>}
     Yeni atributlar: {'__module__': '__main__', '__qualname__': 'SomeClass',
'SOME_PROP': 10, 'SOME_METHOD': <function SomeClass.some_method at
0x796ae05a2340>}
SomeClass nüsxəsi yaradılır...
some_class_object.:  10
some_class_object.SOME_METHOD():  i am some method from SomeClass
```

Yuxarıdakı nümunədə nə baş verir: `MyMetaClass` adlı bir metasinif yaratdıq. Metasinif attributları valideyn sinfinə ötürməzdən əvvəl, onların adlarını yuxarı registrə çevirir(`__` ilə başlayan attributlardan başqa).

> **Tip**
>
> Yəqin ki hiss etdiyiniz - metasiniflərlə edilən demək olar ki hər şeyi, sadə vərəsəlik və ya sinif dekoratorları etmək olar. Ona görə də metaklasslar daha çox - tərtibatçılar üçün nəzərdə tutulmuş iri freymvörklərin pərdəarxasında istifadə olunur.

Tapşırıqlar

1. Konstruktoru iki parametr - `width` və `height` alacaq `Rectangle` sinfini yaradın, bu sinif özündən bir düzbucaqlı ehtiva edəcək.
2. `width` və `height` əsasında düzbucaqlının sahəsini hesablayacaq `get_area` metodunu əlavə edin.
3. `get_area` metodunu `@property` dekoratoru ilə `area` xüsusiyyəti ilə əvəzləyin.
4. `area.setter` ilə istifadəçinin düzbucaqlının sahəsinə yanlış ədəd mənimsətməsinin qarşısını alın.
5. Əlinizdə müxtəlif valyutaların avro ilə dəyərlərindən ibarət olan lüğət var(tam json faylını github reposundan tapa bilərsiniz):

```
{
    "ADA": {
        "code": "ADA",
        "value": 1.4926597641
    },
    "AED": {
        "code": "AED",
        "value": 4.1635106239
    },
    "AZN": {
            "code": "AZN",
            "value": 1.9275247084
        },
        ...
}
```

`value` sahəsindəki dəyər - 1 avronun həmin valyutadakı dəyəridir. Məsələn, `"AZN"` üçün bu dəyər `1.9275247084` kimi qeyd olunub, bu o deməkdir ki, 1 avro = 1.9275247084 AZN. Bu lüğətdən istifadə edərək, elə bir `Currency` sinfi yazın ki:

- Sinfin konstruktoru iki ədəd parametr qəbul etsin: `code` - valyutanın kodu(lüğətdəki kod adı) və `coef` - valyutanın əmsalı olacaq. Məsələn, `Currency(code='AZN', coeff=2)` bizə - "2 manat" verəcək.
- Sinfin `__str__` metodundan istifadə edərək, nüsxələrin sətr görünüşünü yazın. Məqsəd odur ki, biz nüsxələri `print` ilə çap etdikdə anlaşıqlı bir şey görək. Məsələn:

```
>>> azn2 = Currency(code='AZN', coeff=2)
>>> print(azn2)
    Currency: AZN [2]
```

- Sinfin xüsusi metodlarından istifadə etməklə, müxtəlif valyutaların toplanmasını, çıxılmasını, ədədə vurulmasını və bölünməsini təmin edin:

```
azn2 = Currency(code='AZN', coeff=2)
rb_100 = Currency(code="RUB", coeff=100)
print(azn2 + rb_100)  # Currency: AZN [4.1197]
print(azn2 / 2)  # Currency: AZN [1.0]
```

- `>>` operatoru üçün çağrılan `__rshift__` xüsusi metodunun köməyi ilə, elə edin ki biz bir valyutanı digərinə çevirə bilək. Məsələn:

```
azn2 =  Currency(code='AZN', coeff=2)
print(azn2 >> 'RUB')
# Currency: RUB [94.3489007470925]
```

Link: https://github.com/AzizNadirov/python-road-book/blob/master/code/tasks/python_oop_8.ipynb

9.0. Kitabxanalar və "Plug and Play"

İndiyə qədər əsasən proqramlaşdırma komponentlərini, kərpiclərini öyrənmişik. Dövrlər, funksiyalar, siniflər və sairə... Bütün bunlar proqramlaşdırma dilinin özəyini və ya nüvəsini(core) təşkil edir. Dilin əsas təyinatı - proqram hazırlamaqdır. Gəlin bir dəqiqəlik düşünək, proqramlar nələr edə bilir - internet bağlantısı üzərindən mətn və ya fayllar göndərir, video, foto və sairə media faylları ilə işləyir. Bütün bunları düşünəndə, bizim Python haqqında bildiklərimiz - çox cılız, yuxarıdakılara heç dəxli olmayan bir şey kimi gəlir. Elədirmi - xeyir, əsla elə deyil. Pythonın içərisində bütün bunları etmək üçün müxtəlif hazır alətlər var və bu alətlər məhz öyrəndiyimiz funksiyalar, siniflər və sairə üzərində yazılıb. Üstəlik, onlardan tab gücü ilə istifadə etmək üçün onların özlərini bəzən elə funksiya və ya siniflər içərisində yazırlar. Beləliklə, öyrəndiyimiz özək komponentlər irəlidə qarşılaşacağımız bütün alətlərin tərkibini, məntiqini təşkil edəcək.

İndi isə, qayıdaq kitabxanaya. Bəlli məsələdir ki, Python dilinin bacarıqları sıfırdan yazdığımız sadə skript şəntiqi ilə yekunlaşmır. Biz gözəl - düyməli, pəncərəli proqramlar yarada bilərik, bu proqramlar fayllarla işləyə, internet və ya başqa şəbəkələr ilə işləyə bilər, ümumiyyətlə, demək olar ki heç bir məhdudiyyət yoxdur. Bütün bunları etmək üçün Pythonda hazır alətlər toplusu - *standart kitabxana*(standard library) var. Bu kitabxanada təyinatına görə qruplaşdırılmış minlərlə alət var. Məsələn, saat və tarix ilə işləmək üçün `datetime`, riyazi funksiyalar üçün `math`, qrafik interfeysli proqramlar yaratmaq üçün `tkinter`, əməliyyat sistemi ilə bağlı əməliyyatlar üçün `os`, daha çox sistem yönümlü işlər `sys`, fayl yolları üçün `pathlib`, loqlar yazmaq üçün `logging` kitabxanası və sairə var. Bütün kitabxanaların adını çəkməyin mənası yoxdur, çünki birincisi - onlar çoxdurlar, ikincisi - bəziləri o qədər dar təyinatlıdırlar ki, çox nadir hallarda istifadə edilir. Biz isə, pythonik məişətdə ən çox rast gəlinən, istifadə olunan kitabxanalar haqqında danışacağıq.

> **Faydalı**
>
> Dediyim kimi, Pythonda N böyüklükdə kitabxana var və bu kitabxana günü-gündən böyüyür. Kitabxana haqqında professional məlumat üçün ən əsas mənbə elə rəsmi sənədləşmədir: https://docs.python.org/3/library/index.html

İndiyə qədər sizinlə ancaq standart verilənlər strukturları haqqında danışmışdıq, bura sətrlər, yazılar, siyahılar, lüğətlər və çoxluqlar daxil idi. Ancaq, bəzən elə hallar olur ki, məsələni həll etmək üçün bu strukturlar o qədər uğun gəlmir. Belə hallar üçün, pythonda `collections` adlı standart kitabxana modulu var. Bu modul içərisində əsasən "xüsusi təyinatlı" verilənlər strukturları yerləşir. Bu bölmədə haqqında danışacağımız strukturlar - `namedtuple`, `deque`, `OrderedDict`, `Counter`.

9.1.1. `namedtuple` - kim deyirdi ki Pythonda strukturlar yoxdu...

C proqramlaşdırma dili ilə tanışlığı olanlar yəqin ki ordakı strukturlar(structure) adlı verilənlər strukturu ilə tanışdırlar. Bu bir növ özündə attributlar saxlayan sinifə bənzəyir, bir növ sırf xüsusiyyət - data saxlamaq üçün istifadə olunan siniflər. Bir çox istifadəçilər sual verirdi: niyə pythonda buna bənzər bir şey yoxdur? Deyə bilərsiniz - pythonda siniflər, OYP var, niyə struktur əvəzinə adi siniflər şəklində yazmayaq ki? Məsələ orasındadır ki adi siniflər bunun üçün o qədər də uyğun deyil, təsəvvür edin ki noutbukunuzu divardan televizor kimi asmısınız, bəli - filmlərə filan baxmaq olar, amma nə dərəcədə uyğundur... Bu anda köməyimizə haqqında danışacağımız `namedtuple` gəlir. Adından da göründüyü kimi, *named-tuple* - "adlandırılmış yazı" kimi tərcümə olunur. Təsəvvür edin ki, bizə bir `Student` strukturu lazımdır, bu strukturun - qəlibin içərisinə tələbələri qeyd edəcəyik - strukturun hər bir "nüsxəsi" bir tələbə olacaq. Gəlin fikirləşək, bunu necə edə bilərik? Yəqin ki ilk variant kimi lüğətlər siyahısı kimi yazmaq olardı:

```
students = [{'name': 'Ali', 'surname': 'Veliyev'},
            {'name': 'Veli', 'surname': 'Aliyev'}, ...]
```

O qədər də rahat deyil, elə deyilmi? Hansısa tələbəni almaq üçün onun siyahıdakı indeksini bilmək lazımdır, üstəlik hər yeni tələbə - bütün sahələri yenidən və yenidən əllə yazılan lüğətdir. Odur ki, bunu boş verək. Yaxşı, bəs `namedtuple` bizə nə verə bilər? `namedtuple` - bizə *adlandırılmış* və *sahələri əvvəlcədən təyin olunmuş* struktur verir. Məsələmiz üçün, bizə `Student` adlı struktur lazımdır, hansının ki `name` və `surname` adlı sahələri olacaq. Bunun üçün yazırıq:

```
from collections import namedtuple
StudentStructure = namedtuple('Student', ['name', 'surname'])
```

İlk öncə `collections` modulundan `namedtuple` import edirik. `namedtuple` çağıraraq iki arqument ötürürük: `Student` - yaradılacaq strukturun adı, `['name', 'surname']` - sətrlər siyahısı, strukturun malik olacağı sahələr və ya xüsusiyyətlər. Biz burada verilənlər haqqında danışdığımız üçün, buna "sahə" demək daha məntiqlidir. Beləliklə, `namedtuple('Student', ['name', 'surname'])` bizə `Student` adlı, `name`, `surname` sahələrinə malik struktur qaytarır və biz bu strukturu `StudentStructure` dəyişəninə mənimsədirik. `StudentStructure` - bir növ qəlib rolunu oynayacaq, hər dəfə bu dəyişəni çağırıb içərisinə təyin etdiyimiz sahələri ötürdükdə bir tələbə yaratmış olacağıq:

```
ali_veliyev = StudentStructure(name="Ali", surname="Veliyev")
veli_eliyev = StudentStructure("Veli", "Aliyev")
```

Gördüyünüz kimi, sahələri adi arqument kimi struktur qəlibinə ötürürük - mövqeli və ya adlı arqument kimi, heç bir fərqi yoxdur. Nüsxənin sahəsini almaq üçün nöqtə notasiyasından istifadə

edirik - yəni `nüsxə.sahə` kimi:

```
print(ali_veliyev, veli_eliyev)
print(f"ali_veliyev.name: {ali_veliyev.name}; veli_eliyev.surname:
{veli_eliyev.surname}")
```

```
Student(name='Ali', surname='Veliyev') Student(name='Veli', surname='Aliyev')
ali_veliyev.name: Ali; veli_eliyev.surname: Aliyev
```

Çox sadə rahatdır, elə deyilmi? Sadəcə sahələri qeyd etmək üçün ayrıca tam bir sinif yaratmaqdan və ya lüğətlər yazmaqdan daha yaxşıdır. Əlavə olaraq, adlandırılmış yazıları yaradarkən, susmaya görə dəyərləri dəyin edə bilərik - əgər hər hansı sahəni qeyd etməsək - onun susmaya görə dəyəri götürüləcək. Bunun üçün `defaults` parametrinə mövqeyinə görə sahələrin susmaya görə dəyərlərini ötürürük:

```
StudentStructure = namedtuple('Student', ['name', 'surname'],
                              defaults=('unknown_name', 'unknown_surname'))

ali_veliyev = StudentStructure(name="Ali")
veli_eliyev = StudentStructure("Aliyev")

print(ali_veliyev, veli_eliyev)
print(f"ali_veliyev.name: {ali_veliyev.name}; veli_eliyev.surname:
{veli_eliyev.surname}")
```

```
Student(name='Ali', surname='unknown_surname') Student(name='Aliyev',
surname='unknown_surname')
ali_veliyev.name: Ali; veli_eliyev.surname: unknown_surname
```

Əslində, adlandırılmış yazını təyin edərkən sahələrin adlarını başqa üsullarla da təyin etmək olar:

- sətrlər ardıcıllığı kimi - bizim etdiyimiz kimi: `['name', 'surname']`
- bir sətr içərisində, vergül və ya boşluqlarla ayrılmış sözlər kimi:

```
    # boşluq - " " ilə
    StudentStructure = namedtuple('Student', 'name surname')
    # vergül ilə
    Smartphone = namedtuple('Smartphone', 'brand, model')
    ali_veliyev = StudentStructure(name="Ali", surname="Veliyev")
    ip = Smartphone('Apple', 'iPhone 99 Pro Plus Max')
    print(ali_veliyev)
    print(ip)
    ```

```
```

```output
    Student(name='Ali', surname='Veliyev')
    Smartphone(brand='Apple', model='iPhone 99 Pro Plus Max')
```

- generator ilə:

```
    # generator ilə
    n_semestres = 4     # 4 il ~= 8 semestr, qənaət üçün 4 deyək...
    fields = (f"Semester_{i}" for i in range(1, n_semestres+1))
    print('generator:: ', fields)
    print('its values:: ', list(fields))
    # yuxarıdakı "list" ilə generatoru sonlandırmışıq, odur ki yenidən
yaradırıq.
    fields = (f"Semester_{i}" for i in range(1, n_semestres+1))
    StudentGrades = namedtuple('Student', field_names=fields)
    grades_1 = StudentGrades(Semester_1=80, Semester_2=70, Semester_3=90,
Semester_4=85)
    print("instance:: ", grades_1)
```

```
    generator::  <generator object <genexpr> at 0x7dc4d84ff370>
    its values::  ['Semester_1', 'Semester_2', 'Semester_3', 'Semester_4']
    instance::  Student(Semester_1=80, Semester_2=70, Semester_3=90,
Semester_4=85)
```

> **Tip**
>
> Python 3.7 - dən başlayaraq dataklasslar əlavə edilmişdir. Dataklasslar adından da göründüyü kimi "data class"-lardır, struktur kimi istifadə olunan siniflərdir. Hal-hazırda aktiv istifadə olunur və şəxsi fikrimə görə - yuxarıda haqqında danışdığımız `named tuple` rahatlıqla dataklasslarla əvəz etmək olar. Dataklasslar haqqında ətraflı danlşacağıq.

9.1.2. Növbələr və steklər

Sizinlə artıq FIFO və LIFO tipli verilənlər strukturları barədə danışmışıq, o cümlədən stek və növbələri də yad etmişdik. Pythonda stek və növbələri `collections.deque` ilə yaratmaq olar. Gəlin ilk öncə bunun iş prinsipinə, daha sonra isə performans məsələsinə baxaq.

Stek və növbələr barədə qısa xatırlatma.

Növbə - real həyatdakı sıralara, növbələrə bənzəyir. Təsəvvür edin ki, supermarketdə kassanın önündə növbədə dayanmısınız və növbənin bütün iştirakçıları sivil şəkildə öz növbəsini gözləyir. Belə olan təqdirdə, bu struktur FIFO (FirstInFirstOut) - "*birinci girən - birinci çıxır*" tipli olacaq, çünki növbəyə ilk daxil olan tərk edir(sol tərəfdən).

Stek - üst-üstə yığılmış boşqablara bənzətmək olar, ən birinci boşqab - ən altda, ən son daxil olunan boşqab isə ən üstdə yerləşir və altdakı boşqabları çıxartmaq üçün - ilk öncə üstdəki

boşqabları götürmək lazımdır. Əslində, biz stektlərlə təkcə alqoritmlər barədə danışanda yox, həmçinin "çağırış stekləri" barədə danlşanda qarşılaşmışıq. Steklər LIFO (LastInFirstOut) - "*sonuncu girən - birinci çıxır*" tipli verilənlər strukturudur, doğurdan da - ilk olaraq sıradan sonuncu qoyulan boşqab çıxır.

`collections.deque` - ikisi birində

`deque` - bir sıradır, bu sıranın hər iki - həm sağ, həm də sol tərəfinə element əlavə edə və ya silə bilərik. Beləliklə, bunun nəticəsində bir strukturu həm stek, həm də növbə kimi istifadə bilərik. `deque` ilə növbə yaratmaq üçün, `deque` funksiyasını çağırıb, ona hər hansı ardıcıllıq ötürməliyik, heç nə ötürməsək - boş növbə yaratmış oluruq:

```
dq = deque()
# append sağ tərəfdən element əlavə edir
dq.append(1)  # deque([1,])
dq.append(2)  # deque([1, 2,])
dq.append(3)  # deque([1, 2, 3,])
# appendleft - sol tərəfə əlavə edir
dq.appendleft(0)  # # deque([0, 1, 2, 3,])
# pop - sağdan, popleft isə soldakı elementi silib, geriyə qaytarır
removed_left = dq.popleft() # 0 silinir
removed_right = dq.pop() # 3 silinir
print(f"removed_left: {removed_left}; removed_right: {removed_right}")
print(dq)
```

```
removed_left: 0; removed_right: 3
deque([1, 2])
```

`deque` həmçinin `maxlen` adlı parametr alır, bu parametr növbənin maksimal uzunluğunu təyin edir, əgər `maxlen=3` olarsa və növbəyə hər hansı tərəfdən dördüncü elementi əlavə etməyə çalışsaq, o zaman əks tərəfdən bir element "yox olur", növbə bir növ bir vahid sürüşür. Təsəvvür edin ki, mağazada kassa növbəsinin ən əvvəlinə kimsə girir və buna görə də ən sonda dayanan bəxtsiz üzvü növbədən uzaqlaşdırırlar.

```
q = deque([1,2,3], maxlen=3)
print(q)
q.append(4)
print(q)
```

```
deque([1, 2, 3], maxlen=3)
deque([2, 3, 4], maxlen=3)
```

`deque` həmçinin `extend`, `extendleft`, `insert` metodları var, metodların adları sizə hələ siyahılardan tanış gəlməlidir. Özünüz bu metodlarla oynaya bilərsiniz. Unutmayın ki `deque` - dəyişiləbilən obyektdir. Ona görə də `q.append(4)` yazdıqda `q` dəyişəni arxasında dayanan növbə obyektinə `4` əlavə edilir və `append` özü bizə heç nə qaytarmır. Çox güman ki siyahılardan bunu artıq təxmin edirdiniz.

İndi isə, gəlin `deque` əsasında öz `Stack` sinfimizi yazaq. Sinif özünü adi stek kimi aparacaq: elementi yalnız sona əlavə etmək, element almaq istədikdə isə, yalnız ən son elementi ala biləcəyik. Sinfin özəyi - verilənlər strukturu isə növbə - `deque` olacaq:

```
from typing import Sequence, Optional, Any, Union, Tuple
from collections import deque

class Stack:
   """ class simulates a stack based on the que """
   def __init__(self,
                items: Optional[Sequence]=None,
                max_length: Optional[int]=None):
       """
       Args:
           - items: stack initial items
           - max_length - maximum length of the stack
       """
       self.__items = deque(items or (), maxlen=max_length)
       self.__max_len = max_length

   def retrieve(self)->Union[Any, None]:
       """ retrieve item from the stack """
       if len(self.__items) > 0:
           return self.__items.pop()
       print("Stack is empty!")
       return None

   def add(self, item_to_add: Any) -> bool:
       """ add new item to the stack """
       if len(self.__items) == self.__max_len:
           print("Stack is full!")
           return False
       self.__items.append(item_to_add)
       print(f"{item_to_add} added to stack: {self.get_stack()}")
       return True

   def get_stack(self)->Tuple[Any]:
       """ get stack items as a tuple """
       return tuple(self.__items)

   def clear(self) -> None:
       self.__items.clear()
       print("Stack is clean!")

   def __str__(self):
       return str(self.get_stack())
```

```
stack = Stack(items=[1,2], max_length=4)
stack.add(3)
stack.retrieve()
stack.add(3)
stack.add(3)
stack.add(3)
print(f"stack: {stack}")
stack.clear()
print(f"stack after clear: {stack}")
```

```
3 added to stack: (1, 2, 3)
3 added to stack: (1, 2, 3)
3 added to stack: (1, 2, 3, 3)
Stack is full!
stack: (1, 2, 3, 3)
Stack is clean!
stack after clear: ()
```

Sinif dörd ədəd metod təklif edir:

- `add - item_to_add` adlı parametr alır və onu stekə(əslində bilirik ki növbəyə) əlavə etməyə çalışır. Əgər stekin cari uzunluğu, `max_length` parametrində göstərilmiş qiymətə bərabərdirsə, o zaman yeni element əlavə edilmir və geriyə `False` qaytarılır, əks təqdirdə - element əlavə edilir və `True` qaytarılır.
- `retrieve` - stekin son elementini qaytarır, əlbəttə ki stekdən silərək.
- `get_stack` - cari steki bir yazı kimi qaytarır.
- `clear` - steki təmizləyir.

9.1.3. `collections.OrderedDict` - sıralı lüğətlər

Lüğətlərlə yenicə tanış olanda qeyd etmişdim ki, lüğətlərdəki elementlərin(açar-dəyər cütlərinin) sırası - Python 3.7 bəri dəyişmir. Bundan öncə sırası pozulmayacaq lüğətlər yaratmaq üçün `collections.OrderedDict` istifadə edilirdi. `OrderedDict` `dict` sinfinin törəməsidir və demək olar ki eynilə adi lüğətlər kimidir. Bu lüğətləri yaratmaq üçün `OrderedDict` sinfinə cütlər ardıcıllığı ötürmək lazımdır:

```
from collections import OrderedDict

simple_d = {'a': 1, 'b': 2, 'c': 3}
ordered_d = OrderedDict([('a', 1), ('b', 2), ('c', 3)])
print(simple_d)     # {'a': 1, 'b': 2, 'c': 3}
print(ordered_d)    # OrderedDict([('a', 1), ('b', 2), ('c', 3)])
```

Daha bir yaratma üsulu - "boş" nüsxə üzərindən mənimsədərək:

```
od = OrderedDict()
od['a'] = 1
```

```
od['b'] = 2
od['c'] = 3
print(od)   # OrderedDict([('a', 1), ('b', 2), ('c', 3)])
```

Sıralı lüğətlərdə əsas fokus - lüğətdəki elementlərin sıralarına yönəlib, ona görə də elementlərin sırasl ilə bağlı olub, adi lüğətlərdə olmayan bəzi metodlar var:

- `move_to_end(key, last=True)` - `key` açarlı elementi lüğətin sonuna atır, `last=False` olarsa sonuna deyil, əvvəlinə atacaq.
- `popitem(last=True)` - `last=True` olarsa son elementi silərək qaytarır, əks halda isə ilk elementi.

```
ordered_d = OrderedDict([('a', 1), ('b', 2), ('c', 3)])

print(ordered_d)    # OrderedDict([('a', 1), ('b', 2), ('c', 3)])
ordered_d.move_to_end('a') # 'a' sona atılır
print(ordered_d)    # OrderedDict([('b', 2), ('c', 3), ('a', 1)])
ordered_d.move_to_end('c', last=False) # 'c' əvvələ atılır
print(ordered_d)    # OrderedDict([('c', 3), ('b', 2), ('a', 1)])
# elementlərin silinməsi
removed_end = ordered_d.popitem()   # sonuncu elementi silir
print(removed_end)  # ('a', 1)
print(ordered_d)    # OrderedDict([('c', 3), ('b', 2)])

removed_first = ordered_d.popitem(last=False)   # ilk elementi silir
print(removed_first)  # ('c', 3)
print(ordered_d)    # OrderedDict([('b', 2)])
```

Müqaisə edərkən sıra məsələsi: aid lüğətlərdə sıra saxlansa da, müqaisə edərkən heç bir önəm daşımır, belə ki

```
>>> {'a':1, 'b':2} == {'b': 2, 'a': 1}
    True
```

Sıralı lüğətlərdə isə önəmlidir:

```
>>> from collections import OrderedDict
>>> OrderedDict({'a': 1, 'b': 2}) == OrderedDict({'b': 2, 'a': 1})
    False
```

Adi lüğətlərlə də müqaisə etmək olar, bu zaman müqaisə adi lüğətlərlə olduğu kimi olacaq, yəni sıra **önəmsiz** olacaq:

```
>>> {'b': 2, 'a': 1} == OrderedDict({'a':1, 'b':2})
    True
```

Beləliklə, `OrderedDict` sıralı lüğətlərdən o zaman istifadə edirik ki - sıranın gözlənməsi vacib olsun. Bir tərəfdən artıq adi lüğətlərin də sırası dəyişmir, ancaq sıralı lüğətlərdən istifadə etməyimiz

bizə əlavə metodlar və fərqli müqaisə verir. Digər tərəfdən isə, adi lüğət əvəzinə sıralı lüğət istifadə etdikdə, koddan aydın olur ki, burda sıraya diqqət etmək lazımdır.

9.1.4. `collections.Counter` - sayğac

`collections.Counter` - ardıcıllıqda elementləri saymaq üçün istifadə edilir. Ardıcıllığın elementləri xeşlənən olmalıdır. `Counter` də `dict` sinfinin törəməsidir və elementləri daha optimal, pythonic yolla saymağa imkan verir. Gəlin, müxtəlif sayğaclara baxaq:

```
counter_str = Counter("abbcccdddd") # sətr əsasında
counter_list = Counter([1,2,2,3,3,3,4,4,4,4]) # siyahı əsasında
print( counter_str ) # Counter({'d': 4, 'c': 3, 'b': 2, 'a': 1})
print( counter_list ) # Counter({4: 4, 3: 3, 2: 2, 1: 1})
```

Gördüyünüz kimi, counter nüsxəsini çap etdikdə, bir çöv özündə açar kimi - elementi, dəyər kimi isə həmin elementin ardıcıllıqdakı sayını saxlayan lüğət çap edir. Yəni, tez-bazar hər hansı ardıcıllığın elementlərini saymaq lazımdırsa, `Counter` bunu olduqca optimal yolla edəcək. Əlavə olaraq, `.elements()` metodu ilə elementlərin siyahısını ala bilərik, bu zaman `itertools.chain` obyekti alırıq. Qısaca - `chain` - müxtəlif ardıcıllığın elementlərindən yaranmış zəncirdir:

```
>>> from itertools import chain
>>> a,b = [1,2,3], [4,5,6]
>>> chain(a, b)
    <itertools.chain object at 0x708a18bc7bb0>
>>> list(chain(a,b))
    [1, 2, 3, 4, 5, 6]
```

Yuxarıdakı nümunədə `a` və `b` siyahılarından zəncir yaradırıq. Əslində, sadəcə olaraq siyahıları konkat edərək birləşdirmək olardı, amma, biz - generatorlar və iteratorlarla tanış olan uşaqlar olduğumuz üçün, bilirik ki, bu heç də ən yaxşı çıxış yolu deyil. Odur ki, python daha müdrik davranıb, zinciri bir növ ardıcıl generatorlar şəklində reallaşdırıb. İndi isə, qayıdaq `.elements()` metoduna:

```
print(counter_list.elements())  # <itertools.chain object at 0x7f10c161c160>
print(list(counter_list.elements()))  # [1, 2, 2, 3, 3, 3, 4, 4, 4, 4]
```

Əlavə olaraq, `.most_common(n=None)` metodu da var, metod ən çox rast gəlinən top n elementi qaytarır, daha dəqiq desək `(element, sayı)` yazılarından ibarət olan siyahısını qaytarır, susmaya görə, `n=None` olur və bu zaman bütün element-say cütləri sayların azalma ardıcıllığı ilə qaytarılır:

```
print(counter_str)  # Counter({'d': 4, 'c': 3, 'b': 2, 'a': 1})
print(counter_str.most_common(n=2))  # [('d', 4), ('c', 3)]
print(counter_str.most_common())   # [('d', 4), ('c', 3), ('b', 2), ('a', 1)]
```

Counter - sayğacları yalnız nəyisə sayaraq deyil, həmçinin birbaşa - lüğət kimi daxil edərək də yaratmaq olar. Deyək ki, belə bir sayğac yaratmaq istəyirəm: "a olsun 3 dənə, b olsun 1 dənə, c isə olsun 5 dənə". Bunun üçün:

```
>>> from collections import Counter
>>> custom_counter = Counter({'a': 3, 'b': 1, 'c': 5})
>>> custom_counter
    Counter({'c': 5, 'a': 3, 'b': 1})
>>> custom_counter_2 = Counter({'a': 1, 'b': 2, 'c': 3})
>>> custom_counter_2
    Counter({'c': 3, 'b': 2, 'a': 1})
```

Sayları həmçinin toplamaq və çıxmaq olar!

```
>>> Counter({'manat': 2, 'qepik': 50}) + Counter({'manat': 1, 'qepik': 10})
    Counter({'qepik': 60, 'manat': 3})
>>> Counter({'manat': 2, 'qepik': 50}) - Counter({'manat': 1, 'qepik': 10})
    Counter({'qepik': 40, 'manat': 1})
```

Bu hələ harasıdır, hələ sayğacları mənfi saylı da yarada bilərik. Ancaq toplama-çıxma nəticəsində mənfi saylar olmayacaq:

```
>>> Counter({'manat': -2, 'qepik': -50}) + Counter({'manat': 3, 'qepik': 60})
    Counter({'qepik': 10, 'manat': 1})
    #  2.50 borc + 3.60 = 1.50, cibimiz dolu olsun!
```

Tapşırıqlar

1. `namedtuple` istifadə edərək "Book" adlı struktur yaradın. Bu strukturda aşağıdakı sahələr olmalıdır:

   - name - kitabın adı,
   - author - müəllifin adı
   - page_count - səhifə sayı

   Sonra 3 kitab yaradıb, çap edin.

   Gözlənilən nəticə:

   ```
   Book(name='Koroğlu', author='Xalq yaradıcılığı', page_count=250)
   Book(name='Qarabağnamə', author='Mirzə Camal', page_count=180)
   Book(name='Dədə Qorqud', author='Xalq yaradıcılığı', page_count=300)
   ```

2. `Counter` istifadə edərək, `get_most_freq(seq)` funksiyasını yazın. Funksiya hər hansı ardıcıllıq alır və ardıcıllığın ən çox rast gəlinən elementini qaytarır.

   ```
   def get_most_freq(seq):
       pass

   get_most_freq([1,1,1,2,3])    # 1
   ```

3. Bank növbəsini simulyasiya edin. İlk öncə 3 müştəri növbəyə daxil olur: "Əli", "Vəli", "Səməd". Sonra "Leyla" növbəyə əlavə olunur. İlk müştəri xidmət alır (növbədən çıxır). Növbənin cari vəziyyətini göstərin.

   Gözlənilən nəticə:(sadəcə çap edin)

   ```
   İlkin növbə: deque(['Əli', 'Vəli', 'Səməd'])
   Leyla əlavə olundu: deque(['Əli', 'Vəli', 'Səməd', 'Leyla'])
   Əli xidmət aldı və çıxdı
   Cari növbə: deque(['Vəli', 'Səməd', 'Leyla'])
   ```

4. "Contact" adlı namedtuple yaradın, sahələri: name, phone, email. `phone` və `email` susmaya görə dəyərləri boş sətr olsun. Üç ədəd kontakt yaradın:

   - Tam məlumatla, bütün sahələri doldurun.
   - Yalnız ad və telefonla
   - Yalnız adla

   Gözlənilən nəticə:

   ```
   Contact(name='Əli Məmmədov', phone='+9940000000', email='ali@mail.az')
   Contact(name='Leyla Həsənova', phone='+9940000000', email='')
   Contact(name='Rəşad Quliyev', phone='', email='')
   ```

5. Verilmiş mətndə sözlərin tezliyini hesablayın və ən çox təkrarlanan 3 sözü tapın. Böyük-kiçik hərfləri eyniləşdirin və durğu işarələrini silin.

Giriş mətn: "Python programming language is very powerful. Python is easy to learn. Python can do everything. Programming is the future."

Gözlənilən nəticə:

```
 Total word count: 19
Unique word count: 14
Top 3 most repeated words:
    'python' - 3 times
    'is' - 3 times
    'programming' - 2 times
```

5. maksimal uzunluğu 5 olan deque növbəsi yaradın:
    1. İlk olaraq `[1,2,3]` elementlərindən ibarət olsun.
    2. Növbənin sonuna `4` əlavə edin.
    3. Növbənin sonuna `5` əlavə edin.
    4. Növbənin sonuna `6` əlavə edin. Növbənin birinci elementi(`1`) necə oldu ?
6. Sadə kitabxana idarəetmə sistemi yaradın. Bu kitabxana sistemində:

    - Kitabxanada qeydiyyata alınmış kitablar və üzvlər var. Hər iki məvhum üçün adlı yazı yaratmaq olar.
    - Üzvlər kitabxanadan oxumaq üçün kitab icarələyə bilirlər. Deyək ki, bizim sistemdə hər kitabdan ancaq bir ədəd var.
    - Kitab oxunmaya götürüləndə, bu kitab; oxumaqçün müraciət edənlər "gözləmə növbəsinə" daxil olur.
    - Kitab oxucu tərəfindən geri qaytarılan kimi, ilk gözləyən oxucuya verilir. Burada `deque` istifadə etmək olar.
    - Kitaba görə, onun kimdə - hansı üzvdə olduğunu bilmək olur.

    Düşünün, sizcə bu cür sistemi necə dizayn etmək olar? Sinif şəklində reallaşdırmaq olarmı, hansı metod və xüsusiyyətlər olacaq?

Link: https://github.com/AzizNadirov/python-road-book/blob/master/code/tasks/std_lib_collections_9_1.ipynb

---

Siz artıq iteratorlar anlayışı ilə tanışsınız. `itertools` kitabxanası müxtəlif yollarla iterator yaratmağa kömək edən alətlər saxlayır. Az öncə, sizinlə `itertools.chain` ilə tanış olmuşduq, hansı ki adından da göründüyü kimi, ardıcıllıqlardan vahid bir zəncir yaradırdı. Bu alətlərin əksəriyyətini 3-5 sətrlik kodla özümüz də yaza bilərik, sadəcə sual burasındadır ki, bunu etməliyikmi... Moduldakı alətlər bizə generator verdiyi üçün, daha sərfəlidir. Məsələn bilirsiniz ki, qurma `range` funksiyası bizə generator `range` obyekti qaytarır və yalnız bu obyekt üzərindən iterasiya edərkən tələb olunan elementi "yaradır" və qaytarır. Biz adətən iteratoru `list` funksiyasının içərisinə atırıq, ancaq bunu sadəcə vizual olaraq bütün elementləri görmək üçün edirik, yaddaş cəhətdən isə bu o qədər də yaxşı şey deyil. Yadınıza `zip` və `map` funksiyalarını salın, onlar da geriyə iterator qaytarır, Python 2-də isə onlar siyahı qaytarırdı. İterator qaytaran oxşarları isə `itertools.izip` və `itertools.imap` idi, ancaq sonra bu `map` və `zip` də iterator qaytarmağa başladılar, `imap` və `izip` isə `itertools` kitabxanasından silindi.

9.2.1. `itertools.chain` - zəncir

Bu alətlə əslində öncəki bölmədə tanış olmuşuq. `chain` - ixtiyari sayda iterator alır və onlardan ardıcıl bir zəncir düzəldir və geriyə "chain" - zəncir tipli generator qaytarır:

```
letters = ['a', 'b', 'c', 'd', 'e']
numbers = [1, 2, 3, 4, 5]
symbols = itertools.chain(letters, numbers)
print(symbols)  # <itertools.chain at 0x7502c6620a30>
print(list(symbols))  # ['a', 'b', 'c', 'd', 'e', 1, 2, 3, 4, 5]
```

Gəlin öz chain funksiyamızı yazaq, əvvəlcə - sadə siyahı qaytaran variantını yazaq, daha sonra isə, prototipi kimi generator qaytaran variantını yazarıq. Nə yazırıq: funksiya, hansı ki ixtiyari sayda iterasiya olunan(*iterable*) arqument qəbul edir, onları birləşdirir və geriyə qaytarır:

```
def simple_chain(*args)->list:
    """ simple chain function """
    result = []
    for iterable in args:
        result.extend(list(iterable))

    return result

print(simple_chain(letters, numbers))
# ['a', 'b', 'c', 'd', 'e', 1, 2, 3, 4, 5]
```

`simple_chain` funksiyasında ilk öncə ötürülən bütün arqumentləri `*` ilə `args` adlı dəyişənin içərisinə yığırıq. Daha sonra, boş `result` siyahısı yaradıb, dövrü olaraq, ötürülən bütün obyektləri siyahıya çevirib, elementləri `resul` siyahısının sonuna əlavə edirik. Yola vermək üçün pis deyil. Ancaq, biz bilirik ki, generatorlarla olan variant daha optimaldır. Buna görə də biz, gələn bütün iterasiyaolunanları siyahıya çevirib, bir-birinə birdən "calaq etmək" əvəzinə, hər iterasiyada(hər dəfə `next()` çağrıldlqda) növbəsi çatmış iterasiyaolunanın növbəsi çatmış elementini qaytara bilərik. Bunun üçün generatorları, `yield`, `yield from` qısa yazılışını xatırlamaq lazımdır. Başlayaq:

```
from typing import Iterator

def custom_chain(*args)->Iterator:
    """ custom chain function """
    for iterable in args:
        for item in iterable:
            yield item

print(custom_chain(letters, numbers))
 # <generator object custom_chain at 0x7502c6f6ea40>
print(list(custom_chain(letters, numbers)))
# ['a', 'b', 'c', 'd', 'e', 1, 2, 3, 4, 5]
```

`yield from` ilə bir az da qısalda bilərik:

```
from typing import Iterator

def custom_chain(*args)->Iterator:
    """ custom chain function """
    for iterable in args:
        yield from iterable

print(custom_chain(letters, numbers))
 # <generator object custom_chain at 0x7502c6284c70>
print(list(custom_chain(letters, numbers)))
# ['a', 'b', 'c', 'd', 'e', 1, 2, 3, 4, 5]
```

9.2.2. `itertools.accumulate`

Ardıcıllığa "akumulyativ" şəkildə hər hansı funksiyanı tətbiq edir. Akumulyativ - "tədricən artan" kimi başa düşmək olar. Yaxşı, "tədricən" nə deməkdir? Təsəvvür edin ki ardıcıllığımız `l = [1,2,3,4,5]` olsun, hər hansı funksiyamız isə cəm - `sum` olsun. Bu zaman ardıcıllığımız bu cür bölünəcək: [1,2,3,4,5], [1,2,3,4,5], [1,2,3,4,5], [1,2,3,4,5], [1,2,3,4,5] - əvvəlcə funksiyaya ilk element ötürüləcək - `sum(l[0])` , daha sonra alınan cəm və növbəti element ötürülür, beləcə sonacan: təqribən belə bir şey:

```
l = [1, 2, 3, 4, 5]
# İlk elementdən başlayaq
result = l[0]
# cəm növbəti element ilə
result = sum(result, l[1])
# və növbəti
result = sum(result, l[2])
result = sum(result, l[3])
result = sum(result, l[4])
```

`itertools.accumulate` məhz bu cür işləyir, iterasiyaolunan və bir funksiya obyekti alır, hansı ki susmaya görə cəm - `sum` olacaq:

```
from itertools import accumulate

numbers = [1, 2, 3, 4, 5]
print(list(accumulate(numbers)))
# [1, 3, 6, 10, 15]
```

Gəlin, son olaraq yenə də öz variantımızı ataq, ilk variant belə ola bilər:

```
def custom_accumulate(iterable, func:callable=sum)->Iterator:
    """ custom accumulate function """
    for i in range(len(iterable)):
        yield func(iterable[:i+1])

print(custom_accumulate(numbers))
# <generator object custom_accumulate at 0x7502c62bee40>
print(list(custom_accumulate(numbers)))
# [1, 3, 6, 10, 15]
```

Yuxarıda yazdığımız - pis deyil, normal koddur. Amma, bir əmması var: fikir versəz görərsiz ki, biz hər iterasiyada `iterable[:i+1]` (ilk nümunədə qırmızı ilə işarələdiyimiz) üçün funksiyanın nəticəsini təkrarən hesablayırıq. Bunu aradan aradan qaldırmaq üçün hər iterasiyada akumulyasiyanın öncəki nəticəsini də yadda saxlayıb, cari elementlə funksiyaya ötürə bilərik:

```
def custom_accumulate_optimized(iterable, func=lambda x, y: x + y):
   """ custom accumulate function """
   # iteratora çevirək. Hər dəfə next ilə element aldıqda +1 addım atmlş
olacağıq.
   iterator = iter(iterable)
   try:
       # ilk next, ilk elementi alırıq
       accumulated = next(iterator)
       yield accumulated
       # növbə ilə digərləri üçün də hesablayırıq.
       for item in iterator:
           accumulated = func(accumulated, item)
           yield accumulated
   except StopIteration:
       # iterator boşdursa, sadəcə None qaytaraq
       return None

   numbers = [1, 2, 3, 4, 5]
   print(list(custom_accumulate_optimized(numbers)))
   # [1, 3, 6, 10, 15]
```

Tip

`itertools` kitabxanası özündə xeyli bu cür alətlər saxlayır, bu alətlərin hamısı iterator qaytarır və bəziləri o qədər sadə olur ki, onların oxşarlarını yazmaq bir-neçə sətrlik kod edir, amma bunlara rəğmən, bu alətlər kifayət qədər optimal reallaşdırılıb və kiçik məsələlərdə yerli-yerində istifadə olunub kifayət qədər zaman udmağa kömək edə bilər. Bütün alətləri burda

sadalamaq kitabın səhifə sayını və sən oxucunun zamanınını israf etməyə gətirib çıxarda bilərdi. Üstəlik, alətlərin əksəriyyəti ancaq spesifik, nadir hallarda tələb olunduğu üçün, davamlı tanışlıq üçün sadəcə bir-neçə mənbəyə link buraxıram.

- Rəsmi sənədləşdirmə: https://docs.python.org/3/library/itertools.html
- Real Python-da məqalə: https://realpython.com/python-itertools/

Tapşırıqlar

1. Üç qrup tələbə verilib. `itertools.chain` istifadə edərək, vahid bir tələbə siyahısı yaradın:

```
from itertools import chain
group_a = ('Alice', 'Bob', 'Charlie')
group_b = ['David', 'Emma']
group_c = ['Frank', 'Grace']


# Gözlənilən: ['Alice', 'Bob', 'Charlie', 'David', 'Emma', 'Frank', 'Grace']
```

2. Mağazadan son 7 gün ərzində edilən sifarişlərin sayı `daily_orders` adlı siyahıda verilib. Mağaza rəhbərliyi qrafik halında kumulyativ artımı görmək istəyir. Birinci gün - birinci gün üçün sifariş sayı, ikinci gün - ikinci günə qədər(daxil olmaqla, yəni birinci gün və ikinci gün cəm olaraq), üçüncü gün - üçüncü günə qədər və s. `itertools.accumulate` istifadə edərək, gündəlik sifarişlərin kumulyativ cəmini tapın:

```
from itertools import accumulate

daily_sales = [150, 200, 175, 300, 250, 180]

# Gözlənilən nəticə: [150, 350, 525, 825, 1075, 1255]
```

```
Bəs cəm əvəzinə hasil lazım olarsa?
```

3. "Nənənmin Təndiri" mağazalar şəbəkəsi X rayonundan ümumi gəlirin dinamikasını və son olaraq ümumi gəliri görmək istəyir. Deyək ki, sözügedən rayonda üç ədəd mağaza var. `itertools.chain`, `itertools.accumulate` istifadə edərək tələb olunan kumulyativ ardıcıllığı qaytarın.

```
store_a = [150, 200, 175]
store_b = [300, 250, 180]
store_c = [400, 350, 290]

# Kumulyativ gəlir: [150, 350, 525, 825, 1075, 1255, 1655, 2005, 2295]
# Yekun gəlir: 2295
```

4. Öz `chainmap` funksiyanızı yazın. Funksiya bir funksiya obyekti və ixtiyari sayda iterə olunan obyektləri mövqeli arqumentlər kimi alır. Ötürülmüş funksiyanı növbə ilə hər bir iterəolunan obyektin, hər bir elementinə tətbiq edir və geriyə nəticələrdən ibarət siyahı qaytarır. Daha sonra, funksiyanı generator şəklində yazmağa çalışın.

```
from typing import Iterator

def chain_map(func, *iterables) -> Iterator:
    """
    Chain multiple iterables and apply function to each element.
    """
    pass
```

```
numbers1 = [1, 2, 3]
numbers2 = [4, 5]
numbers3 = [6, 7, 8]

result = chain_map(lambda x: x * 2, numbers1, numbers2, numbers3)
print(list(result))  # Gözlənilən: [2, 4, 6, 8, 10, 12, 14, 16]
```

5. AlanAldanmaz mağazası üçün stok idarə etmə sistemi yazmaq tələb olunur. Sistemi bir `InventoryManager` adlı sinif kimi yazın. Bu sistem ilə:

   - Hər bir məhsulu onun - İD nömrəsi, adı, bir ədədinin qiymətini və stokda olan sayı ilə qeyd edirlər.
   - Məhsulun qiyməti sistem üzərindən dəyişdirilə bilər. Qiymət dəyişdikcə, hər bir məhsul üçün qiymət tarixçəsini saxlamaq lazımdır.
   - Ümumi stokun cəmi qiymətini görə bilirik, şərti olaraq, neçə manatlıq malımız var. Unutmayın ki, Əgər stokda `n` ədəd `k` manatlıq X məhsulu varsa, o zaman ümumi qiymət `n*k` olacaq.
   - Zamanla, hər dəfə məhsul əlavə olunduqca stokun kumulyativ qiymətini görə bilməliyik. Məsələn, əgər stokda əvvəlcə 1000 manatlıq noutbuk, daha sonra 500 manatlıq smartfon, və sonda 200 manatlıq qulaqcıqlar əlavə edilibsə, kumulyativ tarixçə belə olacaq: `[1000, 1500, 1700]`

   Məsələnin kodlu şərtinə linkdən keçid edib baxa bilərsiniz.

Link: https://github.com/AzizNadirov/python-road-book/blob/master/code/tasks/std_lib_itertools_9_2.ipynb

9.3 Pythonda zaman və tarix

Tarix və saat. Biz zaman deyərkən tarix və saat nəzərdə tuturuq. Tarix - il/ay/gün olaraq müəyyən bir günü göstərir, məsələn: 28-01-1998. Saat isə, adından da göründüyü kimi müəyyən saatı bildirir - saat: dəqiqə: saniyə millisaniyə(1 san = 1000 ms), məsələn: 21:37:59 900. Ümumiyyətlə, tarix fərqli tərzdə yazıla bilər, məsələn gün-ay-il(ayırıcı kimi "-"), gün/ay/il, il/ay/gün və sairə. Bu cür yazılış fərqlərinə tarixin formatı deyilir, yəni tarixi oxumaqdan söhbət gedirsə, bilmək lazımdır - tarix hansı formatda yazılıb... Əlavə olaraq, tarix və saatı birləşdirib daha dəqiq bir zaman nöqtəsi almaq olar - 28 yanvar 1998-ci il, saat 21:00:50 000.

Saat qurşaqları. Yer kürəsində yaşadığımız üçün, dünyanın bir "küncündə" gecə olanda, digər "küncündə" gündüz olur. Buna görə də, coğrafi bölgələr arasında zaman fərqlərini təyin etmək üçün saat qurşaqları var. Saat qurşaqları arasındakı fərqləri təyin etmək üçün, universal, mərkəzi *UTC* və başqa adı *GMT* (Greenwich Mean Time) saat qurşağından istifadə olunur. Bu saat qurşağı - ən böyük, mərkəzi median olan Qrinviç medianı(İngiltərədə yerləşən Qrinviç rəsədxanasının üzərindən keçdiyi üçün büelə adlanır) üzərində yerləşir. Başqa sözlə, Qrinviçdəki saat göstəricisi bir növ ortaq məxrəc kimi istifadə olunur. Məsələn, Azərbaycan bu saat qurşağından 4 saat irəlidədir(GMT+4), yəni, bizdə saat 14:00 olduqda, Qrinviçə nəzərən bu - [14:00 - 4 saat] - 10:00 olacaq. Texniki dairələrdə çox vaxt GMT yox, UTC adlanır, odur ki ikisindən birini gördükdə bilin ki eyni şeydir. **UNIX timestamp.** Zamanı dəqiq və istinadlı şəkildə ölçmək üçün, "UNIX dövrü"-ndən başlayaraq, indiyə qədər keçmiş saniyələrin sayı götürülür. *UNIX dövrü*(UNIX Epoch) - 01.01.1970 saat 00:00, yəni gecə yarısı 12 tamamda olmuş zaman anıdır. *Unix saatı* isə, həmin andan, ölçmə anına qədər keçmiş saniyələrin sayıdır. Məsələn, aşağıda `datetime.datetime.now()` ilə cari tarixi öyrəşdiyimiz fotmatda alıram. Daha sonra isə `time.time()` ilə UNIX vaxtı alıram, baxdığım an üçün bu təqrbən 1 742 886 940 saniyə idi, yəni 01.01.1970 00:00:00 tarixindən baxdığım ana qədər `1 742 886 940` saniyə zaman keçmişdi. Ortalama il sayını almaq üçün bölmələr edib, 56 il aldım, +- həqiqətə bənzəyir... Zamanın bu cür saniyələr şəklində verilmiş tərzi - *UNIX timestamp*, UNİX zaman ştampı adlanır və adətən ancaq zaman üzərində hesablama etmək və yaddaşda saxlamaq üçün istifadə edilir, bəlli məsələdir ki, biz - insanlar üçün bu cür yazılış heç uyğun deyil, eyni müvəfəqiyyətlə binar və ya onaltılıq say sistemində də saya bilərdik...

```
>>> from datetime import datetime
>>> import time
>>> datetime.now()
    datetime.datetime(2025, 3, 25, 11, 15, 35, 392504)
>>> time.time()
    1742886940.0553772
>>> ts = 1742886940
>>> ts / 60 / 60 / 24 / 30 / 12
    56.03417373971194
```

9.3.1. Pythonda tarix və saat tipi

Pythonda tarix və saatlarla işləmək üçün 3 əsas kitabxana var - `date`, `time` və `datetime`. İlk ikisi ayrı-ayrılıqda tarix və saatlarla işləmək üçün nəzərdə tutulub, `datetime` isə hər ikisini özündə birləşdirir. Biz də, məhz bu modul üzərində fokuslanacağıq.

Sadə `datetime` obyektinin yaradılması. Ən sadə şəkildə `datetime.date(year, month, day)` vasitəsilə tarix yarada bilərik. Gördüyünüz kimi, `date()` üç arqument alır: `year` - il, `month` - ay və

`day` - gün:

```
import datetime
dt = datetime.date(year=2020, month=11, day=20)
print(dt)   # 2020-11-20
```

Oxşar yolla `datetime.datetime()` ilə tarix-zaman tipli obyekt də yarada bilərik:

```
dtt = datetime.datetime(year=2020, month=11, day=20,
                        hour=10, minute=30, second=15, microsecond=0)
print(dtt)  # 2020-11-20 10:30:15
```

Zaman-tarix komponentlərini xüsusiyyət olaraq ala bilərik:

```
print(f"dtt.year = {dtt.year}")
print(f"dtt.second = {dtt.second}")
print(f"dtt.microsecond = {dtt.max}")
```

```
dtt.year = 2020
dtt.second = 15
dtt.microsecond = 0
```

Qəbul edək ki, hələki bu anacan qeyri-adi heç nə ilə rastlaşmadıq.

Tarixin, sətrə və sətrdən. Tez-tez tarix və ya saatı sətr formatından oxumalı oluruq. Məsələn, sətr verilir: `"2020-11-20"`, bu adi tanıdığımız sətrdir, bu sətrin içərisindən tarixi tapıb, tanıyıb oxumaq - onu *pars etmək*(parsing) deməkdir. Ümumiyyətlə, bu termin yalnız tarix-saatlara aid deyil, ümumilikdə hər hansı sətr içərisindən lazım olan struktur(lar)ın tanınıb oxunmasını parsinq adlandırmaq olar. İndi isə qayıdaq əsas məsələyə - tarixi sətr içərisindən necə pars edək? Bunun üçün `datetime` sinfinin `strptime` metodundan istifadə olunur, hansını ki "string parse time" kimi oxumaq olar. Əlavə olaraq, tarixi sətrə çevirən, `strftime` metodu da var, hansını ki "string format time" kimi oxumaq olar. Hər iki halda, tarixin formatını göstərmək lazımdır. Tarixin formatı - tarixin necə, hansı formatda yazıldığını göstərir. Məsələn, bizim sətr `"2020-11-20"` kimidir, yəni `"il-ay-gün"` kimidir. Bunun üçün format `"%Y-%m-%d"` kimi olacaq. Burada `%Y` -il, `%m` -ay, `%d` -gün bildirəcək. Baxaq:

```
import datetime
dt = datetime.date(year=2020, month=11, day=20)
print(datetime.datetime.strptime("2020-11-20", "%Y-%m-%d"))
print(datetime.datetime.strptime("2020-11-20", "%Y-%m-%d").date())
print(dt.strftime("%Y-%m-%d"))
```

```
2020-11-20 00:00:00
2020-11-20
2020-11-20
```

`datetime.strptime()` bizə `datetime` tipli tarix-saat qaytarır, sətrdə ancaq tarix olduğu üçün, saat hissəsi `00:00:00` olur. Daha sonra `.date()` metodu ilə `datetime` obyektini `date` obyektinə - tarix obyektinə çeviririk. Yəni, tarix-saat tipindən tarix tipinə çeviririk, bununla da saat hissəsini atmış oluruq. Formatlar haqqında daha ətraflı şəkildə burdan oxumaq olar: https://docs.python.org/3/library/datetime.html#strftime-and-strptime-behavior

Cari saat və tarix. Pythonda cari tarix və ya saatı almaq üçün `now()` və `today()` metodlarından istifadə olunur. `now()` metodu cari saatı, `today` isə cari tarixi qaytarır. Sizi çaşdıra biləcək məqam - hansı sinfin metodundan istifadə etməyiniz ola bilər, qeyd etmişdik ki `datetime` modulu özündə `date`, `time` və `datetime` siniflərini saxlayır. Həm `date`, həm də `datetime` sinfinin `today` metodu var. `datetime.datetime.today()` tarix-saat, `datetime.date.today()` isə ancaq tarix qaytarır. `now()` isə ancaq `datetime.datetime` sinfində var.

```
from datetime import date, datetime

print("date.today(): ", date.today())
print("datetime.today(): ", datetime.today())
print("datetime.now(): ", datetime.now())
```

```
date.today():  2025-03-27
datetime.today():  2025-03-27 11:15:45.734946
datetime.now():  2025-03-27 11:15:45.735017
```

Əlavə olaraq, tarixlə saatı birləşdirə bilərik, deyək ki müəyyən bir tarix və saat obyektimiz var və bunları bir araya gətirmək lazımdır. Bunun üçün `datetime.datetime` obyektinin `combine()` metodundan istifadə olunur, metod saat tipli arqument alır və geriyə birləşmədən yaranan tarix-saat qaytarır.

```
import datetime

dt = datetime.datetime.strptime("2020-11-20", "%Y-%m-%d")
combined = dt.combine(dt, datetime.time(10, 30, 50))
print("dt: ", dt)
print("combined: ", combined)
```

```
dt:  2020-11-20 00:00:00
combined:  2020-11-20 10:30:50
```

9.3.2. Pythonda zaman miqdarı - `timedelta`

Zaman miqdarı deyərkən, hə hansı zaman aralığının uzunluğunu, davametmə müddətini nəzərdə tuturuq. Məsələn bir həftənin davametmə müddəti - 7 gündür və yaxud bir iş gününün savametmə müddəti 8 saatdır. `datetime.timedelta` sinfi də məhz bu cür zaman müddətlərini göstərmək üçün istifadə olunur. Bəs bu müddətlər necə istifadə oluna bilər? Təsəvvür edin, 10 İyun 2025 tarixində, 1 illik müddətli həqiqi hərbi xidmətə yola düşürsünüz və təxris olacağınız günü hesablamaq istəyirsiniz. Bunun üçün, 10.06.2025 tarixinin üzərinə 1 il gəlirsiniz və alırsınız 10.06.2026. Yəni `tarix + zaman_müddəti = başqa_bir_tarix` alırsınız. Pythonda da məhz bu cür işləyir. `timedelta` sinfi ilə bizə lazım olan zaman-müddət(bundan sonra, sadəcə timedelta[taymdelta]) obyektini yaradırıq və istifadə edirik. Ümumilikdə, sinfin siqnaturası belədir:

```
class timedelta(
       days: float = ...,
       seconds: float = ...,
       microseconds: float = ...,
       milliseconds: float = ...,
       minutes: float = ...,
       hours: float = ...,
       weeks: float = ...
   )
```

Gördüyünüz kimi, timedelta-nın parametr kimi ala bildiyi ən böyük parametr-vahid həftə, ən kiçiyi isə mikrosaniyədir(1 san =1 milion mikrosaniyə). Amma, təcrübədə daha çox gün, saat və dəqiqə lazım olur.

```
from datetime import datetime, timedelta, date

a_week = timedelta(days=7)
today = date.today()
print(f"Today: {today}")
print(f"previous week: {today - a_week}")
```

```
Today: 2025-03-27
previous week: 2025-03-20
```

Nə etdik: `timedelta(days=7)` ilə 7 günlük(1 həftə) timedelta yaratdıq və `a_week` dəyişəninə mənimsətdik. Daha sonra, cari tarixi `today` dəyişəninə mənimsədib, cari tarixdən(`today`) 1 həftəni(`a_week`) çıxdıq və aldıq - 7 gün öncənin tarixini. Yeri gəlmişkən timedelta da bir kəmiyyət olduğu üçün, onu ədədə vurmaq, bölmək olar. Aydın məsələdir ki, 7gün * 2 =14gün və ya 7gün+7gün = 14 gün edəcək. Əlavə olaraq:

```
print(a_week/2)
# 3 days, 12:00:00
```

yəni, 7gün bölünsün 2-ə, bərabərdir - 3 gün 12 saat və ya 3.5 gün. Daha bir "fan-fact":

```
repr(a_week / 2)
# 'datetime.timedelta(days=3, seconds=43200)'
```

timedelta obyektinə `repr` təsviri ilə baxsaq, görərik ki müddəti gün və saniyələr ilə təsvir edir(43200 san=12 gün).

timedelta-nı əslində bir tarixi, başqa tarixdən çıxaraq da almaq olar. Məsələn, `sabahkı_tarix - bu_gün=1_gün` olması məntiqli görünür:

```
from datetime import datetime, timedelta, date

today = date.today()
tomorrow = today + timedelta(days=1)
yesterday = today - timedelta(days=1)
print(f"Today: {today}")    # bu gün
print(f"Tomorrow: {tomorrow}")  # sabah
print(f"Yesterday: {yesterday}") # dünən

print(f"Yesterday - today: {yesterday - today}")
print(f"Tomorrow - yesterday: {tomorrow - yesterday}")
```

```
Today: 2025-03-27
Tomorrow: 2025-03-28
Yesterday: 2025-03-26
Yesterday - today: -1 day, 0:00:00
Tomorrow - yesterday: 2 days, 0:00:00
```

Ola bilsin ki `Yesterday-today = -1 day` olması sizi çaşdırdı. Məsələ ondadır ki, müqayisə nöqteyi-cəhərdən - hər sonrakı tarix, öncəkindən böyükdür. UNİX zamanını xatırlayın, hər ötən saniyə ərzində biz 1970-dən uzaqlaşırıq. Ona görə də gələcək keçmişdən həmişə böyük olacaq. Əyləşib, bu sözlərin fəlsəfi-mənəvi doğruluğu barədə diskussiya aparmaq olar, amma texniki olaraq bu bir faktdır.

9.3.3. Pythonda saat qurşaqları

İndiyə qədər yazdığımız tarix-saat ilə bağlı kodların heç birində saat qurşaqlarına toxunmadıq, istifadə etmədik. Pythonda tarix-saatlar saat qurşağı barədə məlumatı özündə saxlaya da bilər, saxlamaya da. Susmaya görə, tarix-saat yaradarkən saat qurşağı barədə məlumat olmur. Ancaq, biz məlumatı `tz` arqumenti vasitəsilə ötürə bilərik. Özündə saat qurşağı haqqında məlumat saxlayan tarix-saat nüsxələri texniki ədəbiyyatda *aware*, saat qurşağını saxlamayanlar isə *naive* tarix-saat nüsxələri adlandırılır. Sual yaranır - bəs nə ötürək? Əvvala IANA(https://www.iana.org/time-zones) adlı qurum ölkələrin saat qurşaqları barədə məlumatları toplayıb, mütəmadi olaraq yeniləyir. Məlumatları mütəmadi olaraq yeniləməyin əsas səbəbi - bəzi ölkələrdə yay və qış vaxtlarının fərqlənməsidir, yadınızdadırsa, bizdə də bir neçə il əvvəl saatlar 1 saat irəli-geri çəkilirdi. Saat qurşaqlarının da əsas təyinatı - zaman anını tam olaraq dəqiqləşdirməkdir. Deyək ki, tarix-saat verilib. Bəs necə bilək ki verilən zaman anında Bakıda saat

neçə idi, bu nə vaxt idi? Amma, əgər zaman qurşağı "Asia/Tashkent", yəni Özbəkistan olaraq qeyd olunubsa, artıq bilirik ki bu Bakı vaxtından 1 saat irəlidir, çünki Özbəkistan GMT+5, Bakı isə GMT+4 saat qurşağında yerləşir. Saat qurşaqları məsələsi Pythonda yaxın zamanlara qədər bir az qəliz məsələ idi. Onlarla rahat işləmək üçün adətən üçüncü tərəf kitabxanalarından istifadə edirdilər(Pythonın öz rəsmi sənədləşməsində belə) və buna əsas səbəblərdən biri saat qurşaqları ilə işləyəcək alətin IANA ilə sinxron işləməli olması idi. Bunun üçün üçüncü tərəf kitabxanaları `pytz` və `python-dateutil` istifadə olunurdu və hələ də istifadə olunur. Ancaq, Python 3.9-da standart kitabxanaya yeni `zoneinfo` adlı modul əlavə edildi və bu modul IANA bazası əsasında çalışır. Qayıdaq əsas məsələyə, qeyd etdim ki, saat qurşağı, yəni - *time zone*(qısaca tz) tarix-saat yaradarkən sinfin `tz` adlı arqumenti kimi ötürülür. Biz bu arqumenti - `zoneinfo.ZoneInfo` sinfinin nüsvəsi kimi ötürə bilərik. Sinfi çağırarkən, coğrafi zonanın kodunu ötürmək lazımdır, bu adətən belə tərtib olunur: `<qitənin_adı/Paytaxt_şəhərin_adı>` , məsələn, "Asia/Baku". Daha ətraflı bu viki məqalədəki cədvəldən baxmaq olar: https://en.wikipedia.org/wiki/List_of_tz_database_time_zones

```
from zoneinfo import ZoneInfo
from datetime import datetime

dt_baku = datetime.now(tz=ZoneInfo("Asia/Baku"))
print(dt_baku)
dt_utc = datetime.now(tz=ZoneInfo("UTC"))
print(dt_utc)
```

```
2025-03-27 23:13:18.467632+04:00
2025-03-27 19:13:18.471263+00:00
```

Gördüyünüz kimi, Bakı vaxtı üçün tarix-saatın çap formatı `+04:00` ilə, UTC üçünsə `+00:00` ilə bitir. Yadımıza salırıq ki, UTC bir növ koordinat başlanğıcıdır, Bakı vaxtı isə UTC + 4 saatdır.

Saat qurşaqları arasında keçid. Bir saat qurşağında verilmiş tarixi, başqa saat qurşağına daşımaq üçün `astimezone()` metodundan istifadə olunur. Metod arqument kimi yenə `ZoneInfo` nüsxəsi alır:

```
print("in UTC: ", dt_utc)
print("in Baku: ", dt_utc.astimezone(ZoneInfo("Asia/Baku")))
print("difference: ", dt_utc - dt_utc.astimezone(ZoneInfo("Asia/Baku")))
```

```
in UTC:  2025-03-27 19:14:02.264817+00:00
in Baku:  2025-03-27 23:14:02.264817+04:00
difference:  0:00:00
```

Yuxarıda, biz eyni tarix-saatı iki fərqli saat qurşağında təsvir etmişik - UTC və Bakı ilə. Ancaq, fərqli saat qurşaqlarında olmalarına baxmayaraq, əlbəttə ki zaman olaraq eyni vaxtdır. Yəni UTC(Qrinviçdə) və Bakıda bu tarixlər eyni anı ifadə edir, sadəcə aralarındakı 4 saat fəqinə görə

saatlar fəqlidir, vəssalam. Faktiki olaraq, eyni zaman anı olduğu üçün də fərqləri sıfırdır. Ancaq, bu iki tarix-saat nüsxələrinin saat qurşaqlarını silsək və iki naiv tarix-saat kimi fəqlərinə baxsaq 4 saat alacağıq. Yeri gəlmişkən, mövcud zaman-tarix obyektini dəyişmək üçün - daha doğrusu, dəyişdirilmiş sürətini almaq üçün `replace()` metodundan istifadə olunur. Metod eynilə yaradarkən ötürdüyümüz arqumentləri alır, yəni `replace` çağırarkən ötürəcəyimiz xüsusiyyətlər yeni qaytarılan nüsxədə dəyişdiriləcək. Bunda istifadə edərək, tz haqqda məlumatı olan tarix-zaman nüsxəsinin `replace` metodunu çağırıb, `tz=None` ötürərək həmin tarix-saat nüsxəsinin saat qurşağısız olan variantını almış olarıq.

```
baku_without_tz = dt_baku.replace(tzinfo=None)
utc_without_tz = dt_utc.replace(tzinfo=None)
print("in UTC: ", utc_without_tz)
print("in Baku: ", baku_without_tz)
print("difference: ", baku_without_tz - utc_without_tz)
```

```
in UTC:  2025-03-27 19:14:02.264817
in Baku:  2025-03-27 23:14:02.264270
difference:  3:59:59.999453
```

Tip

Tarix və saatlarla bağlı bir xeyli maraqlı məqamlar, təzadlar var. Hamısı barədə burda danışa bilmədiyim üçün, maraqlananlar üçün:

- Maraqlı "PyCon sayğacı" sayğacı layihəsi ilə RealPython məqaləsi: https://realpython.com/python-datetime/#working-with-time-zones
- The Problem with Time & Timezones - Computerphile: https://www.youtube.com/watch?v=-5wpm-gesOY

Tapşırıqlar

1. Doğum tarixini alıb, ad gününə qədər qalan günlərin sayını qaytaracaq `days_until_birthday` funksiyasını yazın.
2. Doğum tarixini alıb, geriyə yaş qaytaracaq `calculate_age` funksiyasını yazın. Yaşı, `tuple[int, int, int]` kimi, yəni üç tam ədəddən ibarət yazı kimi qaytarın - il, ay və gün sayı.

```
    from datetime import date

    def calculate_age(birth_date: date) -> tuple[int, int, int]:
        """
        Calculate age in years, months, and days.

        Args:
            birth_date: Date of birth

        Returns:
            Tuple of (years, months, days)

        Example:
            Birth: 2000-05-15
            Today: 2025-03-27
            Result: (24, 10, 12) - 24 years, 10 months, 12 days
        """
        pass

    # Test
    birth = date(2000, 5, 15)
    years, months, days = calculate_age(birth)
    print(f"Age: {years} years, {months} months, {days} days")
```

3. Sətr formasında verilmiş tarixi alıb, onu `datetime.date` tipli tarix obyekti kimi qaytaracaq `parse_event_date` adlı funksiya yazın. Tarix aşağıdakı formatlarda ola bilər:

- `"2025-03-27"` (ISO format)
- `"27/03/2025"` (European format)
- `"03/27/2025"` (US format)
- `"27.03.2025"` (Dot separated)

```
    from datetime import datetime

    def parse_event_date(date_string: str, format_hint: str = None) ->
datetime:
        """
        Parse event date from various formats.

        Supported formats:
        - "2025-03-27" (ISO format)
        - "27/03/2025" (European format)
        - "03/27/2025" (US format)
```

```
        - "27.03.2025" (Dot separated)

        Args:
            date_string: Date as string
            format_hint: Optional hint like "ISO", "EU", "US", "DOT"

        Returns:
            datetime object
        """
        pass
```

4. Beynəlxalq görüş(zəng) çeviricisi yazın. Görüşün zamanı(`datetime`) və saat qurşağı var. Çevirici görüşün zamanını, saat qurşağını və iştirakçıların saat qurşaqlarından ibarət siyahı alır. Məqsəd - görüşün siyahıdakı hər bir saat qurşağı üçün saatını tapmaqdır. Məsələn, görüş Bakı vaxtı ilə saat 19-00 üçün (tz: Asia/Baku) təyin olunubsa, Asia/Tokyo(UTC+9) saat qurşağı üçün bu saat 10-00 olacaq.

```
def schedule_meeting(
    meeting_time: datetime,
    meeting_timezone: str,
    participant_timezones: list[str]
) -> dict[str, datetime]:
    """
    Convert meeting time to all participant time zones.

    Args:
        meeting_time: Meeting datetime (naive)
        meeting_timezone: Timezone where meeting is scheduled (e.g.,
"Asia/Baku")
        participant_timezones: List of participant timezones

    Returns:
        Dictionary mapping timezone: local meeting time

    Example:
        Meeting: 2025-03-27 14:00 in Baku (Asia/Baku)
        Participants: ["Europe/London", "America/New_York", "Asia/Tokyo"]

        Returns:
        {
            "Asia/Baku": 2025-03-27 14:00:00+04:00,
            "Europe/London": 2025-03-27 10:00:00+00:00,
            "America/New_York": 2025-03-27 06:00:00-04:00,
            "Asia/Tokyo": 2025-03-27 19:00:00+09:00
        }
    """
    pass
```

Link: https://github.com/AzizNadirov/python-road-book/blob/master/code/tasks/std_lib_datetime_9_3.ipynb

9.4 Functools

Pythonın rəsmi sənədləşməsinə əsasən, `functools` modulunda *high-order function* - yuxarı səviyyəli və ya ali dərəcəli funsiyalar yerləşir, bu funksiyalar arqument kimi funksiya alır və geriyə funksiya qaytarır(və yaxud sadəcə funksiya üzərində nəsə icra edir). "funksiya" deməyimizə rəğmən, əslində funksiya əvəzinə istənilən *callable* - çağrılan obyekt ola bilər. OYP xüsusi metodlardan yadınızdadırsa, sinfin `__call__` metodunu yazaraq, bu sinfin nüsxələrini çağrılan edə bilirdik.

9.4.1. Keş alətləri

Keş(cache) - adətən hər hansı hesablama nəticələrinin yaddaşda saxlanması və eyni hesablamanın yenidən aparılmasına cəhd edərkən həmin saxlanılmış nəticənin istifadəsi mesanizmidir. Başqa sözlə, əgər hər hansı əməliyyatın təkrarlanma ehtimalı varsa, keş vasitəsilə əməliyyatın nəticəsini saxlayıb, ehtiyac yarandığı təqdirdə götürüb istifadə edirik. Funksiyalara bunun nə dəxli var - adətən funksiyaya təkrarən eyni arqumentləri ötürdükdə nəticə dəyişmir və bu "yaxşı funksiya" göstəricisidir. Məsələn, verilmiş ardıcıllığın cəmini hesablayan `my_sum` funksiyamız var, funksiyanı `[1,2,3]` ardıcıllığı ilə çağırsaq `my_sum([1,2,3])` bizə verəcək `6` . Əgər təkrarən eyni funksiyanı eyni giriş verilənləri ilə çağırsaq, nəticə dəyişməyəcək, yəni `my_sum([1,2,3])` həmişə bizə `6` verəcək. Bundan istifadə edərək, niyə belə bir cədvəl yaratmayaq - cədvəldə funksiyanın arqumentlərini və həmin arqumentləri ötürdükdən sonra funksiyanın geriyə qaytardığı nətcəni yazaq. Daha sonra, funksiyanı çağırarkən yoxlayaq - əgər bu arqumentlərlə funksiyanı artıq çağırmışıqsa, deməli cədvəldə bu arqumentlər və onların nəticəsi olacaq. Elə isə, funksiyanı bu arqumentlər üçün yenidən hesablamaq əvəzinə cədvəldən istifadə edərək birbaşa nəticəni cədvəldən qaytaraq. Ənənəyə sadiq qalaraq, oxşar mexanizmi özümüz reallaşdırmağa çalışaq. İlk öncə, yadımıza salırıq ki, məqsədimiz funksiyanın giriş və çıxış verilənlərini yadda saxlayıb, hər icra zamanı həmin nəticələrdən istifadə etməkdir. Hədəf obyekt funksiya olduğu üçün, həlli bir dekorator kimi yazmaq tam yerinə düşər. Dekoratorumuzun nəticələri bir lüğət içərisində saxlayacaq, lüğətin açarları - giriş verilənləri(mövqeli və adlı arqumentlər), dəyər isə - bu giriş verilənlərlə funksiyanın geriyə qaytardığı nəticə olacaq. Giriş verilənlərini öncədən tanış olduğumuz packing(`*args` , `**kwargs`) üsulu ilə qəbul edəcəyik. Deyək ki belə bir funksiyamız var:

```
def my_sum_cacher(arr, delay=0):
    time.sleep(delay)
    return sum(arr)
```

Funksiya ötürülmüş massivin cəmini qaytarır və bir növ uzun hesablamanı simulyasiya etməkçün `time.sleep` ilə proqramı gözləmə rejiminə keçiririk. Keçək keş dekoratoruna.

```
def casher(func):
    cache = {}
    def wrapper(*args, **kwargs):
        key = (args, tuple(kwargs.items()))
        if key not in cache:
            cache[key] = func(*args, **kwargs)
        return cache[key]

    return wrapper
```

Dekoratorumuz ötürülən arqumentlərdən bir tuple yaradır və bu onu açar kimi saxlayır. Bilirsiz ki açar kimi ancaq xeşlənəbilən - dəyişiləbilməyən obyektlərdən istifadə etmək olar. Odur ki, tuple əvəzinə `frozenset` də istifadə etmək olardı, anca siyahı yox. İndi isə, funksiyamızı dekoratorumuza bükək:

```
@casher
def my_sum_cacher(arr, delay=0):
    time.sleep(delay)
    return sum(arr)

my_sum_cacher((1,2,3), delay=2) # 2 san
my_sum_cacher((1,2,3), delay=2) # 0 san
```

Funksiya iki ədəd parametr alır - yazı `(1,2,3)` və `delay=2`. Biz ilk dəfə bu arqumentləri ötürdükdə dekorator bu arqumentlərdən açar düzəldib funksiyanın nəticəsini dəyər kimi bu açar altında saxlayır. Funksiya hər dəfə çağrıldıqda yoxlayır - bu açar altında dəyər var, ya yox. Varsa, bu o deməkdir ki artıq bu arqumentlərlə funksiyamız üçün nəticə var və onu qaytarır.

Məhdud keşli variantı: Yuxarıdakı `casher` funksiyasının ən böyük mənfi cəhəti - keş ölçüsünün qeyri-məhdud olmasıdır. Təsəvvür edin ki, bu funksiyanı daim onlayn olan veb patformada istifadə edirik və funksiya hər saniyə çağrılır. Funksiya bütün giriş-nəticə cütlərini yadda saxladığı üçün, əməli yaddaşı israf edə bilərik. Buna görə də, adətən yadda saxlanılmış cütlərin sayına məhdudiyyət tətbiq olunur. Məhdudiyəti keşdə saxlana biləcək cütlərin sayına tətbiq edə bilərik. Məsələn, saxlana biləcək maksimal cütlərin sayı 128 olarsa, o zaman ən son 128 cütü yadda saxlayaq. Niyə məhz ən son - çünki bu daha məntiqli görünür, nəinki bütün keçmişi yadda saxlamaq. Odur ki, cütləri yazdığımız lüğətə cüt əlavə etməzdən əvvəl yoxlamalı olacağıq - lüğətin ölçüsü, təyin etdiyimiz ölçüdən böyükdür, ya yox. Böyük olarsa, ən birinci əlavə edilmiş(ən köhnə) cütü silib və ancaq ondan sonra yeni cütü lüğətə yaza bilərik. Lüğətlərdən bu cür istifadə etmək üçün məhz bu yaxınlarda öyrəndiyimiz `collections.OrderedDict` uyğun gəlir. Gəlin `cacher` funksiyamızı elə yeniləyək ki, funksiya bizdən ölçüyə məhdudiyyəti ifadə edəcək `maxsize` arqumentini alsın.

```
from collections import OrderedDict

def casher_lru(maxsize=128):
    def decorator(func):
        cache = OrderedDict()
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            key = (args, tuple(kwargs.items()))
            if key in cache:
                cache.move_to_end(key)
            else:
                if len(cache) >= maxsize:
                    cache.popitem(last=False)
                cache[key] = func(*args, **kwargs)
            return cache[key]
        return wrapper
    return decorator
```

```
@casher_lru(maxsize=2)
def my_sum_lru(arr, delay=0):
    time.sleep(delay)
    return sum(arr)

print(my_sum_lru((1,2,3), delay=2)) # 2 san
print(my_sum_lru((1,2,1), delay=2)) # 0 san
print(my_sum_lru((1,2,3), delay=2)) # 2 san
```

Əvvəla, dekoratorlar və qapanmalar bəhsindən bilirik ki, parametrli dekorator funksiyası yaratmaq üçün, normal dekorator funksiyasını daha bir lay funksiyaya bükmək lazımdır və bir növ üç lay dərinlikli funksiya almış oluruq. Əsas funksiyamız `maxsize` arqumentini alır, hansı ki məhz məhdudiyyəti təyin edir. Yuxarıda dediyim kimi, cütləri saxlamaq üçün `OrderDict` - sıralı lüğətdən istifadə etmişik. Bunun sayəsində, rahatlıqla lüğətin əvvəlinə və ya sonuna element əlavə etmək, silmək mümkündür. Keşin ölçüsünə baxıb, `maxsize` parametrini aşdığı təqdirdə `cache.popitem(last=False)` ilə ilk elementi silirik. Nümunə olaraq, funksiyamızı 1-ci və 3-cü dəfə eyni parametrlərlə çağırırıq, ancaq `my_sum_lru` funksiyasını dekoratorumuza bükərkən `maxsize=2` kimi təyin etdiyimə görə, 3-cü cütü yazarkən 1-cinin nəticəsi yaddaşdan atılır və 3-cü çağırışımız yenidən icra olunur. Keş funksiyalarını tam mənimsədikdən sonra, baxdığımız `functools` kitabxanasındakı `cache`, `lru_cache` haqqında danışmaq olar. Funksiyalar mahiyyətcə eynilə yazdığımız kimidir. `cache` - sadə keş funksiyasıdır, `lru_cache` isə cüt sayına görə məhdudiyyətli keş funksiyasıdır. Bizim yazdığımızdan fərqli olaraq, `functools.lru_cache` funksiyası əlavə olaraq məntiqi `typed` parametri alır. Əgər, `typed=True` ötürəriksə, qiymətcə eyni, tipləri fərqli olan girişlər fərqli kimi qəbukl ediləcək. Məsələn, əgər funksiyanı `foo(3)` və `foo(3.0)` kimi çağırsaq, hər ikisi ayrı-ayrılıqda keçə yazılacaq, çünki baxmayaraq ki `3 == 3.0` bizə `True` verir - onların tipləri fərqlidir.

```
import functools, time

@functools.lru_cache(maxsize=128)
def my_sum_lru(arr, delay=0):
    time.sleep(delay)
    return sum(arr)

print(my_sum_lru((1,2,3), delay=2)) # 2 san
print(my_sum_lru((1,2,1), delay=2)) # 0 san
print(my_sum_lru((1,2,3), delay=2)) # 2 san
```

⚠ **Diqqət!**

- Keş olunan funksiyanın arqumentlər açar kimi istifadə olunduğuna görə, həşlənən olmalıdırlar. Əks halda

```
print(my_sum_lru([1,2,3], delay=2))
# TypeError: TypeError: unhashable type: 'list'
```

alacaqsınız, çünki siyahılar həşlənən deyil.

- Keşləməni ancaq "ancaq xarici arqumentlərdən asılı olan" funksiyalara tətbiq edin, əks halda gözlənilməz yanlış nəticə alacaqsınız. Məsələn, funksiyanızın daxilində cari zamanı alıb, bu zamanla əlaqəli nələrsə edirsinizsə - bu o deməkdir ki, giriş arqumentlər eyni olsa belə, zaman fərqli olduğu üçün funksiya fərqli işləyəcək, bu cür funksiyanı keş edərsək, keş olunan zaman qaytarılan nəticə yadda qalacaq.

9.4.2. Qismən təyin olunmuş - partial funksiyalar

Təsəvvür edin ki, bir `foo(a, b, c)` funksiyamız var. Necə elə bir surətini almaq istəyirik ki, orada `c=5` olsun, yəni müəyyən arqumentləri "öncədən ötürülmüş" olsun. Məsələn, çox sadə `power` funksiyası verilib:

```
def power(base, exponent):
   return base ** exponent
```

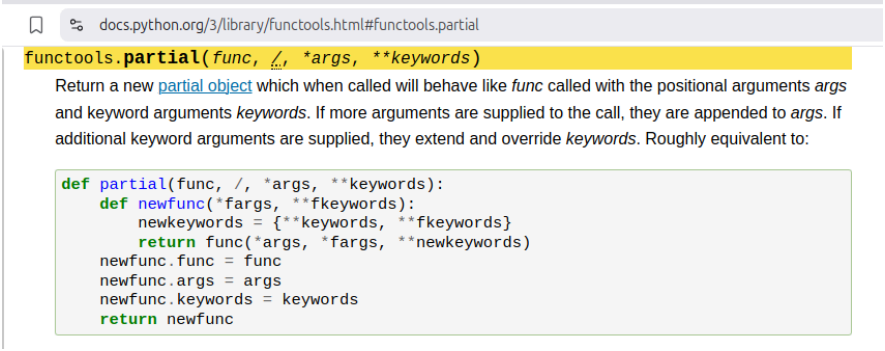
Elə etmək istəyirik ki, bir `square` funksiyası olsun və bu funksiya bir növ `square := power(base, exponent=2)` olsun, yəni ilkin funksiyanın `exponent=2` arqumenti əvvəlcədən təyin edilmiş olsun. Bunu əslində, qapanma qaydasına görə bu cür edə bilərik:

```
def partial(func, *args, **kwargs):
    def wrapper(*more_args, **more_kwargs):
        return func(*args, *more_args, **{**kwargs, **more_kwargs})
    return wrapper
```

`partial` funksiyamız nə edir: funksiya və ixtiyari sayda adlı və ya mövqeli arqumentlər alır. Bu arqumentlər - ötürülən funksiyaya avtomatik olaraq ötürüləcək. `wraper` isə, daha sonra funksiyaya ötürüləcək növbəti parametrləri ikinci növbədə - `partial` funksiyasına ötürülən arqumentlərdən sonra ötürəcək.

```
square = partial(power, exponent=2)
cube = partial(power, exponent=3)
print(square(3))  # 9
print(cube(3))    # 27
```

`square = partial(power, exponent=2)` sətri ilə sanki `power[base=None, exponent=2]` kimi bir funksiya yaradırıq - `power[base=None, exponent=2]` yazılışı ilə ola bilsin ki Amerika kəşf edirəm, ancaq bunu belə başa düşməyinizi istəyirəm - " `power` funksiyasının `base` parametri boşdur, təyin edilməyib, `exponent` isə `2` dəyərini alacaq". Bu fəndi demək olar ki eyni yolla reallaşdıran `functools.partial` funksiyası var. Python rəsmi səhifəsində(https://docs.python.org/3/library/functools.html#functools.partial) buna bənzər ekvivalent kodu da verilib:



9.4.3. Pythonda funksiyaların overload edilməsi - single dispatching

Overloading nədir: kompilyasiya edilən dillərdə eyni adlı funksiyanın(və ya metodun) - müxtəlif sayda, müxtəlif tipli arqumentlər alan çoxlu variantlarını yazmaq olar. Bu zaman, funksiya çağrılarkən ötürülən arqumentlərin sayına və tipinə görə uyğun olan funksiya çağrılır. Bu texnika - *overloading* və ya funksiyanın *aşırıyüklənməsi* adlanır. Niyə "aşırıyüklənmə", çünki biz bir funksiya adına birdən çox reallaşdırılmış məntiq yükləyirik. Java üçün aşırıyüklənməni belə göstərmək olar(ps, kodun sintaksisinə dalmaq lazım deyil, bizim Javalıq işimiz elə burda da bitəcək):

```
// https://www.geeksforgeeks.org/method-overloading-in-java/
// overloading in Java
public class Sum {
    // Overloaded sum()
    // This sum takes two int parameters
    public int sum(int x, int y) { return (x + y); }
    // Overloaded sum()
    // This sum takes three int parameters
    public int sum(int x, int y, int z)
    {
        return (x + y + z);
    }
    // Overloaded sum()
    // This sum takes two double parameters
    public double sum(double x, double y)
    {
        return (x + y);
    }
    // Driver code
    public static void main(String args[])
    {
        Sum s = new Sum();
        System.out.println(s.sum(10, 20));  // burda çap edir
        System.out.println(s.sum(10, 20, 30));
        System.out.println(s.sum(10.5, 20.5));
    }
}
```

```
30
60
31.0
```

Yuxarıdakı Java nümunədə `Sum` sinfinin üç ədəd `sum` metodu reallaşdırılıb və çağrılarkən funksiyanın siqnaturasına(ötürülən arqumentlərin sayına, tiplərinə) görə lazım olan variant çağrılır. Yaxşı, qayıdaq Python-a... Python interpretə olunan dildir və yalnız demək olar ki hər şey məhz icra zamanı baş verir. Ona görə də Pythonda bu cür mexanizm yoxdu. Çünki aşırıyüklənmə kompilyasiya zamanı təyin olunur, lazımi funksiya variantı tapılır və sairə. Bizdə - Pythonda isə, belə söhbət yoxdur.

`singledispatch` . Əvəzində Pythonda `functools.singledispatch` dekorator-funksiyası var. Bu funksiya vasitəsilə biz, bir əsas(generic, əslində ümumi kimi də tərcümə etmək olar) funksiyaya bir neçə funksiya kodu bağlaya bilərik. Adından da göründüyü kimi, bu zaman biz tək bir arqument ilə məhduduq. Yəni bir arqumentin tipinə görə dispatching(lazım olan kod implementasiyasının seçilməsi) baş verəcək. İlk öncə, əsas funksiyamızı `@functools.singledispatch` ilə təyin edirik. Daha sonra, `<funksiya_obyekti>.register` dekorator-metodu ilə hər bir tip üçün ayrıca funksiya bağlayırıq. Bağlanan funksiyanın, arqumentin hansı tipi üçün çağrılacağını təyin etmək üçün, bağlanan funksiyanın başlığındakı arqument tip göstəricisinə(type hinting) sahib olmalıdır. Seçim edilərkən, məhz tip göstəricisinə görə təyin ediləcək - bu funksiya hansı tipli arqument üçün nəzərdə tutulub.

```
import functools

@functools.singledispatch
def my_sum(arg):
   print("Unknown type")
   raise NotImplementedError("Unsupported type")

@my_sum.register
def mysum_list(arg: list):
   print("summing for list")
   return sum(arg)

@my_sum.register
def mysum_dict(arg: dict):
   print("summing for dict values")
   return sum(arg.values())

print(my_sum([1, 2, 3]))
print(my_sum({"a": 1, "b": 2, "c": 3}))
print(my_sum(1))  # raise NotImplementedError
```

```
summing for list
6
summing for dict values
6
```

```
Unknown type
...
 3 @functools.singledispatch
         4 def my_sum(arg):
         5     print("Unknown type")
   ----> 6     raise NotImplementedError("Unsupported type")

   NotImplementedError: Unsupported type
```

Yuxarıdakı nümunədə, `my_sum` funksiyası `@functools.singledispatch` dekoratoru ilə təyin edilib və ümumi funksiyadır. Funksiya bir `arg` adlı arqument alır və çağrıldıqda sadəcə `NotImplementedError` xətası atır. Daha sonra, funksiyanın siyahılar və lüğətlər üçün iki implementasiyasını qeyd edirik: `mysum_list` və `mysum_dict`. Arqument siyahı tipində olarsa - `mysum_list` çağrılacaq, çünki funksiyanın başlığında `def mysum_list(arg: list):` arqumentin tip göstəricisi `list` olaraq göstərilib. `mysum_dict` isə lüğət tipli arqument üçün çağrılacaq, çünki tip göstəricisində `arg: dict` qeyd edilib. Ötürülən arqumentin tipinə uyğun implementasiya tapılmayanda elə ümumi funksiya işə düşür, o isə bizim misalımızda xəta atır.

Yuxarıdakı nümunədə funksiyanın müxtəlif variantlarının hərəsinə ayrıca ad vermişik: `mysum_dict` və `mysum_list`. Ancaq, gördüyünüz kimi, bu adlar yeç yerdə istifadə olunmur, çünki biz sadəcə obyekti `register` ilə bağlayırıq vəssalam. Buna görə də adətən bu "aralıq" funksiyaları sadəcə `_` ilə adlandırırlar, çünki onsuz da funksiya variantının adından istifadə edilməyəcək, elə isə niyə boşuna adlar fəzasında adların sayını artıraq ki...

```
import functools
from typing import Union

@functools.singledispatch
def my_sum(arg):
    print("Unknown type")
    raise NotImplementedError("Unsupported type")

@my_sum.register
def _(arg: list):
    print("summing for list")
    return sum(arg)

@my_sum.register
def _(arg: dict):
    print("summing for dict values")
    return sum(arg.values())

@my_sum.register
def _(arg: int | float):
    print("summing for int")
    return arg

@my_sum.register
def _(arg: Union[set, tuple]):
    print("summing for set or tuple")
    return sum(arg)
```

```
print(my_sum([1, 2, 3]))
print(my_sum({"a": 1, "b": 2, "c": 3}))
print(my_sum(5))
print(my_sum({1, 2, 3}))
print(my_sum((1, 2, 3)))
print(my_sum("hello"))  # NotImplementedError: Unsupported type
```

```
summing for list
6
summing for dict values
6
summing for int
5
summing for set or tuple
6
summing for set or tuple
6
Unknown type
```

Gördüyünüz kimi, tip göstəricisi kimi bir neçə tipi işarə edən `typing.Union` və ya `|` operatorundan istifadə etmək olar, bu imkan Python 3.11-dən başlayaraq əlavə edilib. Əlavə olaraq, `register` metodundan dekorator kimi deyil, adi funksiya kimi çağırıb funksiya implementasiyasını qeyd edə bilərik. Bu zaman birinci arqument - hədəf tip, ikinci arqument isə - bu tipə bağlanacaq funksiya obyekti olacaq. Bundan istifadə edərək, funksiya obyekti kimi lambda ifadə də qeyd etmək olar:

```
my_sum.register(type(None), lambda arg: print("summing for None "))
my_sum(None)  # summing for None type
```

Tapşırıqlar

1. İki yolla fibonaçi ardıcıllığının n-ci həddini qaytaran funksiya yazın. Birinci `fibonacci_slow` funksiyası heç bir keşdən istifadə etməsin. İkinci, `fibonacci_fast` funksiyasında isə maksimal ölçüsü 128 olacaq `lru_cache` istifadə edin. Hər iki funksiyanı eyni ədədlərlə test edin. Aradakı fərqin səbəbi nədir?
2. Verilmiş mətn ismarıcını loq kimi çap edən `log_message` funksiyasını yazın. Loq ismarıcın dörd növü(dərəcəsi) olacaq: "INFO", "WARNING", "ERROR" və "DEBUG". Funksiya ismarıca onun dərəcəsini və çağrıldığı zamanı əlavə edib çap edəcək:

```
def log_message(message: str,
                level: Literal["INFO", "WARNING", "ERROR", "DEBUG"] = "INFO",
                timestamp: bool = True)->None:
    """
    Log a message with level and optional timestamp.

    Args:
        message: The message to log
        level: Log level (INFO, WARNING, ERROR, DEBUG)
        timestamp: Whether to include timestamp
    """

log_message("Done!", level="INFO")
# [2026-01-16 20:24:15] [DEBUG] some log
log_message("Upps, something wrong...", level="ERROR")
# [2026-01-16 20:26:02] [ERROR] Upps, something wrong...
```

3. `functools.partial` istifadə edərək, `log_message` funksiyasından uyğun olaraq ancaq DEBUG və INFO dərəcəli loqlar yazacaq `log_debug` və `loh_info` adlı funksiyaları yaradın.

Link: https://github.com/AzizNadirov/python-road-book/blob/master/code/tasks/std_lib_functools_9_4.ipynb

9.5 Pathlib - Pythonda fayl sistemi yolları ilə işləməyin müasir yolu

Sizinlə artıq fayllar ilə işləmişik. Ancaq, fayla çatmaq üçün, fayl sistemi ilə işləməyi bacarmaq lazımdır: hər hansı qovluqdakı faylların siyahısını almaq, onları açmaq, onlara yazmaq, silmək və sairə. İndiyə qədər fayl yolu bizim üçün sadəcə bir sətr idi, məsələn: `/home/aziz/Documents/books/python.pfd` . Deyək ki, bizə bu fayl yolunun ana qovluğu(bir səviyyə geriyə) lazımdır, yəni `/home/aziz/Documents/books/` qovluğu. Adi sətr ilə bunu almaq çətin olacaq. Əvəzində, Python 3.4-də əlavə edilən `pathlib` standart kitabxanasından istifadə etmək olar. Kitabxanadan istifadə etməyin ən böyük müsbət cəhətlərindən biri - əməliyyat sisteminin növünə görə asılılığı minimuma endirməsidir. Məsələn, `anar` adlı istifadəçinin `Documents` qovluğunun tam yolu Windows, Linux və Mac üçün belə olacaq:

- Windows: `C:\Users\anar\Documents`
- Linux: `/home/anar/Documents`
- MacOS: `/Users/anar/Documents`

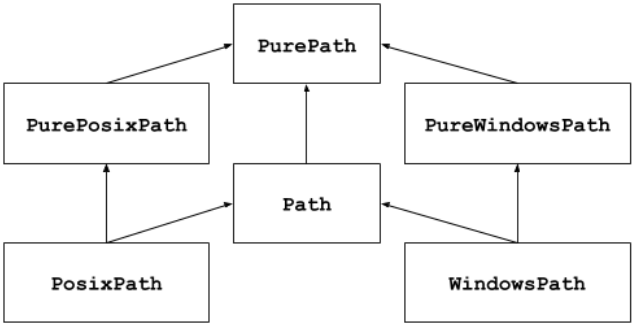
Ancaq, `pathlib.Path.home() / "Documents"` əmri bütün əməliyyat sistemlərində istifadəçinin `Documents` qovluğunu qaytaracaq.

9.5.1. Kitabxananın ümumi quruluşu

Pathlib kitabxanası OOP tərzində dizayn edilib və iki əsas qrupa bölünür:

- **Pure Path - xalis yol**: bu yol obyektləri yalnız fayl yolunun hissələri ilə işləyə bilir, kompüterin fayl sistemindən nəsə oxuya və ya yaza bilmirlər.
- **Path - yol**: pure path kimidir ancaq, əlavə olaraq fayl sisteminə yaza və oxuya bilir. Ümumilikdə, bizim üçün fayl yolları iki sinfə bölünür:

1. **WindowsPath**: windows ƏS fayl yolları ilə işləmək üçün
2. **PosixPath**: UNIX tipli(Linux, Mac) ƏS fayl yolları ilə işləmək üçün. Python-ın rəsmi dokumentasiyasından götürükmüş aşağıdakı şəkildən göründüyü kimi, iyerarxiyanın əvvəlində `PurePath` sinfi dayanır və bu sinifdən `PurePosixPath` və `PureWindowsPath` sinifləri törəyir. Daha sonra, özündə yazı-oxu bacarığını ehtiva edən `Path` sinfi də `PurePath` sinfindən törəyir, ona görə də bir növ `[PurePath + yazı/oxu bacarığı]` deməkdir. Daha sonra, bu sinfin daha konkret `PosixPath` və `WindowsPath` sinifləri gəlir. Biz əsasən `Path` sinfi üzərində işləyəcəyik, bu sinif yazı-oxu ehtiva edir və hər iki sinif üçün ümumidir.



Sadəcə yadınızda saxlayın ki, bundan sonrakı nümunələr Linux ƏS ilə icra edilib. `Path` sinfi isə Windows üçün `WindowsPath`, UNIX üçünsə `PosixPath` tipli yol qaytarır. Aşağıda eyni kodun Linux və Windows ƏS-lərində icra nəticəsi göstərilib:

```
>>> from pathlib import Path
>>> # Windows OS
>>> Path.home()
    WindowsPath('C:/Users/gvido')
>>> # Linux
>>> Path.home()
>>> PosixPath('/home/gvido')
```

9.5.2. `pathlib.Path` əsaslar

Gəlin əvvəlcə sadə bir yol üzərində, bu yollarla edə biləcəyimiz sadə əməliyyatlara baxaq.

```
book_path = Path("/home/tengo/Documents/irodov4.pdf")
print(book_path) # /home/tengo/Documents/irodov4.pdf
print(type(book_path)) # <class 'pathlib.PosixPath'>
```

`book_path` dəyişəni, `pathlib.PosixPath` (və ya `WindowsPath`) tipli obyektdir və gördüyünüz kimi bir pdf sənədə ünvanlanır. Gəlin növbə ilə bəzi əsas xüsusiyyət və metodlara baxaq, görək bu obyektlə nələr edə bilərik:

- `name`: yoldakı son qovluğun və ya faylın adını qaytarır. Əgər yol fayla aparırsa, faylın adına onun suffiksi - genişlənməsi də aid edilir.
- `stem`: yuxarıdakı `name` xüsusiyyətinin suffikssiz formasını qaytarır. Əslində isə sadəcə `name` xüsusiyyətinin sondan `.<str>` hissəni atır. Məsələn, `name` bizə `app.tar.gz` qaytarırsa, `stem` bizə `app.tar` verəcək, çünki suffix kimi `.gz` hissəsi atılır.
- `suffix`: faylın genişlənməsini qaytarır. Əgər yol fayl deyil, qovluğa aparırsa - suffix boş sətr olacaq. Əlavə olaraq, `suffixes` xüsusiyyəti də var və suffikslərin siyahısını qaytarır.
- `parent`: faylın və ya qovluğun ana qovluğunu qaytarır, yəni son faylı və ya qovluğu özündə saxlayan qovluq.
- `parts`: yolun tərkib hissələrini siyahı kimi qaytarır.
- `is_file()` və `is_dir()`: yolun fayl və ya qovluq olduğunu yoxlayır. Yuxarıdakı xüsusiyyətlər fayl sisteminə giriş tələb etmir, onlar sadəcə yolun sətr quruluşunu təhlil edir. Bu xüsusiyyətlər `PurePath`-dən gələn xüsusiyyətlərdir.
- `exists()`: yolun fayl sistemində mövcud olub-olmamasını yoxlayır.

```
print(f"file: {repr(book_path)}")
print(f"name: {book_path.name}")
print(f"stem: {book_path.stem}")
print(f"suffix: {book_path.suffix}")
print(f"parent: {book_path.parent}")
print(f"parts: {book_path.parts}")
print(f"exists: {book_path.exists()}")
print(f"is_file: {book_path.is_file()}")
print(f"is_dir: {book_path.is_dir()}")
```

```
file: PosixPath('/home/tengo/Documents/irodov4.pdf')
name: irodov4.pdf
stem: irodov4
suffix: .pdf
parent: /home/tengo/Documents
parts: ('/', 'home', 'tengo', 'Documents', 'irodov4.pdf')
exists: True
is_file: True
is_dir: False
```

`Path` sinfinin nüsxələri dəyişilməyən obyektlərdir və əməliyyat sisteminin özəlliklərinə riayət etməyə çalışır, məsələn Windows ƏS-də fayl və qovluqlarının adlarında hərflərin registri - böyük və ya kiçik hərflə yazılması fərq daşımır. Hərçən UNIX tipli ƏS-də bu fərq yaradır. Yəni, Windows ƏS-də `MyBook.pdf` və `mybook.pdf` eyni fayldır, UNIX-də isə fərqli. Odur ki, istifadə etdiyiniz sinfin `WindowsPath` və ya `PosixPath` olmasından asılı olaraq yol özünü həmin ƏS-ə xas olan şəkildə aparacaq:

```
win_path = pathlib.PureWindowsPath(r"C:\Users\tengo\Documents\app.tar.gz")
win_path_2 = pathlib.PureWindowsPath(r"C:\Users\tengo\documents\App.tar.gz")
print(f"win_path == win_path_2: {win_path == win_path_2}")
# out: win_path == win_path_2: True
```

Yuxarıdakı nümunədə, yolların birində `Documents` və `app.tar.gz` kimidir, digərində isə `documents` və `App.tar.gz`. Ancaq, söhbət Windows ƏS-dən getdiyi üçün, bu iki yolun eyni olduğu alırıq. Oxşar şeyi Unix ƏS yolları üçün nəzərdə tutulmuş siniflərlə yoxlasaq:

```
unix_path = pathlib.PurePosixPath("/home/tengo/Documents/app.tar.gz")
unix_path2 = pathlib.PurePosixPath("/home/tengo/documents/App.tar.gz")
print(f"unix_path == unix_path2: {unix_path == unix_path2}")
# out: unix_path == unix_path2: False
```

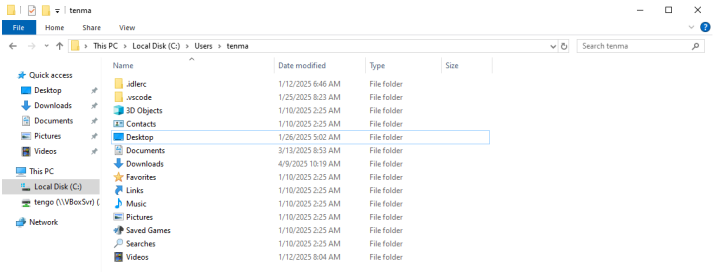
Əlavə olaraq, iki fərqli fayl sistemi üçün sinifləri də müqayisə edə bilərik, ancaq bu zaman `False` alacağıq, çünki fərqli fayl sistemlərindəki yollar eyni ola bilmir:

```
posix = pathlib.PurePosixPath(r"C:\Users\tengo\documents\App.tar.gz")
win = pathlib.PureWindowsPath(r"C:\Users\tengo\documents\App.tar.gz")
print(f"posix == win: {posix == win}")
# out: posix == win: False
```

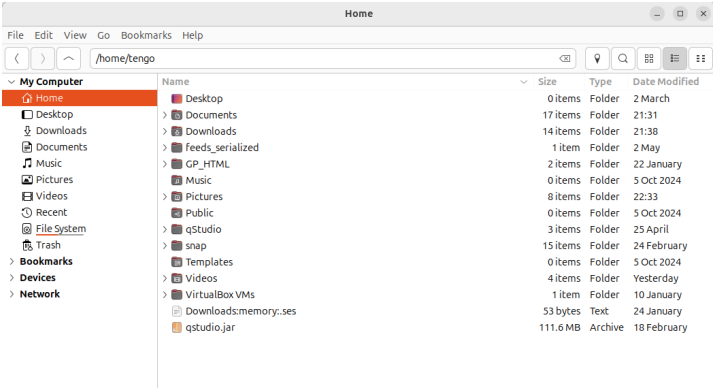
9.5.3. İstifadəçinin ev qovluğu və cari iş qovluğu

İstifadəçinin ev qovluğu: Müasir(əslində son 50-60 ildə belədir :D) əməliyyat sistemləri çox istifadəçili olur. Bu o deməkdir ki, bir bir kompüterdən bir neçə istifadəçi istifadə edə bilər. Hər bir istifadəçinnin öz qovluqları, quraşdırılmış proqramları və icazələr toplusu ola bilər - kimdə admin olur, kimsə qonaq qismində istifadə edə bilir və sairə. Nəzərə alsaq ki istifadəçinin bütün aktivliyi fayl sistemində saxlanılır(proqramlar, sənədlər və s.), ƏS tərəfindən hər bir istifadəçi üçün ayrıca qovluq yaradılır və artıq bu qovluqdan başlayaraq sizin "My Documents", "My Music" və sairə qovluqlarınız

yer alır. Bütün bunların kökündə isə sizin istifadəçinin ev qovluğu - *home directory* dayanır. Aşağıdakı şəkildə Windows 10 ƏS üçün `tenma` adlı istifadəçinin ev qovluğu göstərilib.



Aşağıda isə, Linux(Ubuntu) ƏS-də `tengo` adlı istifadıçinin ev qovluğu verilib.



Göürndüyü kimi, Linuxda istifadəçi qovluqları `home` qovluğunun içərisində yerləşir. Baxmayaraq ki bu iki əməliyyat sistemi kifayət qədər fərqlidirlər, görünür ki müəyyən məqamlarda oxşardırlar. Windows yaddaş qurğularını disk kimi fayl sisteminə qurur(*mount* edir) - `C` diski, `D` diski və sairə. Linuxda isə tək bir başlanğıc - kök (root) var. Hər bir qurğu fayl sisteminin müəyyən bir hissəsinə qurulur.

`pathlib.Path` sinfinin `home()` metodu bizə istifadəçinin ev qovluğunu qaytaracaq, Linux üçün bu `PosixPath` , Windows üçün isə `WindowsPath` olacaq:

```
>>> from pathlib import Path
>>> Path.home()
    PosixPath('/home/tengo')
```

Cari iş qovluğu: Kompüterdə hər hansı proqram, proses işə düşərkən, müəyyən bir qovluq içərisində işə düşür. Deyək ki, `home/tengo/Documents/scripts/snake_game.py` yolu python ilə yazdığımız `snake_game.py` skriptinə aparır. Biz bu skripti işə salsaq, bu proses üçün cari iş qovluğu - `home/tengo/Documents/scripts` olacaq. Nisbi yollar da adətən cari işçi qovluğuna əsasən tamamlanır. Məsələn, əgər cari iş qovluğu `/home/tengo` olarsa və burdaca yerləşən `/home/tengo/Documents` qovluğuna keçmək istəyiriksə, bunun üçün terminalda `cd /home/tengo/Documents` əvəzinə sadəcə `cd ./Documents` və ya `cd Documents` yazmaq kifayət

edir, çünki sistem bu cür nisbi yolları məhz cari iş qovluğuna nəzərən emal edir. Burada `.` - məhz cari iş qovluğunun qısa yazılışıdır. Cari iş qovluğunu terminalda `pwd` əmri, Pythonda isə - `pathlib.Path.cwd()` metodu ilə almaq mümkündür. Terminalda:

```
tengo@EWave:~/Documents/book_cs_with_python$ pwd
    /home/tengo/Documents/book_cs_with_python
```

Python üçün:

```
>>> import pathlib
>>> pathlib.Path.cwd()
PosixPath('/home/tengo/Documents/book_cs_with_python')
```

Sətrlərlə `/` operatoru:

Çox zaman, yolun sonuna hər hansı hissəni əlavə etmək lazım gəlir, məsələn, qovluq yolumuz var və yolun sonuna `mybook.pdf` faylını əlavə edərək falın yolunu almaq istəyirik. qovluğun yolu `Path` sinfinin nüsxəsidirsə - bunu sadəcə `"mybook.pdf"` sətrini həmin nüsxəyə `/` operatoru ilə birləşdirməklə etmək olar:

```
books_dir = Path.home() / "Documents" / "books"
informatics_book = books_dir / "informatics.pdf"
print(informatics_book)
# /home/tengo/Documents/books/informatics.pdf
```

Əla! `pathlib` mövzusu çox vacibdir və müasir Python-da fayl sistemi ilə işləməyi xeyli asanlaşdırır. Gəlin mövzunu davam etdirək və sonra qeyd etdiyiniz digər maraqlı başlıqlara keçək.

9.5.4. Fayl və Qovluqlarla Praktiki Əməliyyatlar

Öncə deyildiyi kimi, `PurePath` sinfindən fərqli olaraq, `Path` sinfi sistemə yazı/oxu imkanına malikdir. Gəlin görək praktiki olaraq bunu necə etmək olar.

Faylları Oxumaq və Yazmaq

`Path` obyektləri vasitəsilə faylları asanlıqla oxuya və yaza bilərik. Bunun üçün `read_text()`, `write_text()`, `read_bytes()` və `write_bytes()` kimi metodları var.

- `read_text(encoding=None, errors=None)` : faylın məzmununu mətn (sətr) olaraq oxuyur. `encoding` parametri ilə faylın kodlaşdırmasını göstərə bilərsiniz (məsələn, `"utf-8"`).
- `write_text(data, encoding=None, errors=None, newline=None)` : verilmiş `data` sətrini fayla yazır. Fayl mövcud deyilsə yaradılır, mövcuddursa üzərindən yazılır.
- `read_bytes()` : faylın məzmununu baytlar (bytes) olaraq oxuyur. Bu, şəkillər, videolar kimi qeyri-mətn faylları üçün faydalıdır.

- `write_bytes(data)` : verilmiş `data` baytlarını fayla yazır.

Gəlin baxaq:

```
from pathlib import Path
# Cari iş qovluğunda "salam.txt" mətn faylını yaradaq
txt_file = Path.cwd() / "salam.txt"

# Fayla mətn yazaq
txt_file.write_text("Salam, millət!\nPythonda pathlib öyrənirik...",
encoding="utf-8")

# Fayldan mətni oxuyaq
oxunan_mətn = txt_file.read_text(encoding="utf-8")
print(f"Fayldan oxunan: {oxunan_mətn}")

# İndi isə baytlarla işləyək
bayt_faylı = Path.cwd() / "data.bin"
# \xc9\x99 - utf-8 ilə ə hərfini verir
orijinal_baytlar = b"M\xc9\x99n bayt s\xc9\x99triy\xc9\x99m."
bayt_faylı.write_bytes(orijinal_baytlar)

oxunan_baytlar = bayt_faylı.read_bytes()
print(f"Bayt faylından oxunan: {oxunan_baytlar}")
print(f"Deşifrələnmiş: {oxunan_baytlar.decode('utf-8')}")
```

```
Fayldan oxunan: Salam, millət!
Pythonda pathlib öyrənirik...
Bayt faylından oxunan: b'M\xc9\x99n bayt s\xc9\x99triy\xc9\x99m.'
Deşifrələnmiş: Mən bayt sətriyəm.
```

Bundan öncə faylları oxuyarkən və ya yazarkən `with` kontekst menecerindən istifadə edirdik, bunun sayəsində faylımız sonda təhlükəsiz şəkildə bağlanırdı. Yuxarıdakı hazır metodlar isə bunu özləri edir. Həmçinin baytlarla nümunə kimi mətn istifadə etmişəm hərçənd, istənilən məzmunu bayt kimi fayla yazmaq olar - şəkli yükləyib, bayt kimi oxuyub, nəsə edib, bayt kimi `əla şəkil.jpg` kimi yaddaşda saxlamaq olar.

Qovluq Yaratmaq

Yeni qovluq yaratmaq üçün `mkdir(mode=0o777, parents=False, exist_ok=False)` metodundan istifadə olunur.

- `mode` : yaradılacaq qovluğun icazələrini təyin edir (UNIX sistemləri üçün daha aktualdır).
- `parents` : əgər `True` olarsa və yolda aralıq qovluqlar mövcud deyilsə, onlar da yaradılacaq. Məsələn, `Path("A/B/C").mkdir(parents=True)` əmri `A` və `A/B` yoxdursa, onları da yaradacaq. Susmaya görə `False` -dur, yəni ana qovluq mövcud olmalıdır.
- `exist_ok` : əgər `True` olarsa və qovluq artıq mövcuddursa, xəta baş verməyəcək. Susmaya görə `False` -dur, yəni qovluq mövcuddursa `FileExistsError` xətası alacağıq.

```
new_dir = Path.cwd() / "test_folder" / "inner_dir"

try:
    new_dir.mkdir(parents=True, exist_ok=False)
    print(f"'{new_dir}' yaradıldı.")
    new_dir.mkdir(parents=True, exist_ok=True) # xəta verməyəcək
    print(f"'{new_dir}' onsuz da var idi, exist_ok=True olduğu üçün problem
yoxdur.")
except FileExistsError:
    print(f"'{new_dir}' artıq mövcuddur!")
```

Qovluq Məzmunu Üzərində İterasiya

Bir qovluğun içindəki fayl və qovluqlara baxmaq üçün bir neçə metod var:

- `iterdir()` : qovluğun içindəki birbaşa elementlər (fayllar və alt qovluqlar) üzərində iterasiya etmək üçün iterator qaytarır. Rekursiv deyil, yəni qovluqdakı altqovluqların içərisinə girməyəcək.
- `glob(pattern)` : verilmiş `pattern` - şablona (nümunəyə) uyğun gələn fayl və qovluqları tapır. Nümunə olaraq `*` (istənilən sayda simvol) və `?` (bir simvol) kimi xüsusi simvollardan istifadə etmək olar.. Məsələn, `Path.cwd().glob("*.txt")` cari qovluqdakı bütün `.txt` fayllarını özündə saxlayacaq. Bu metod da rekursiv deyil.
- `rglob(pattern)` : yuxarıda haqqında danışdığımız `glob` kimidir, ancaq rekursivdir, yəni bütün alt qovluqların içərisində axtarış edəcək.

```
current_dir = Path.cwd()
print(f"\n'{current_dir}' içərisindəkilər:")
for element in current_dir.iterdir():
    if element.is_file():
        print(f"  Fayl: {element.name}")
    elif element.is_dir():
        print(f"  Qovluq: {element.name}")
print(f"Amma, iç qovluqlara baxa bilmədim...")
```

`rglob` metodundan istifadə edərək, bir "fayl axtarış" funksiyasını yaza bilərik:

```
def find_files(path: Path,
               pattern: str,
               recursive: bool=True)->list[Path]:
    """ search for files matching `pattern` in `path` directory """
    if not path.is_dir():
        raise ValueError(f"'{path}' must be a directory!")
    if recursive:
        return list(path.rglob(pattern))
    else:
        return list(path.glob(pattern))

# Documents qovluğunda bütün .pdf fayllarını tapaq
print(find_files(Path.home() / "Documents", "*.pdf", recursive=True))
```

Fayl və Qovluqları Silmək

- `unlink(missing_ok=False)` : faylı silir. Əgər yol qovluğa aparırsa, `IsADirectoryError` xətasını verəcək. `missing_ok=True` olarsa və fayl mövcud olmasa belə xəta baş verməyəcək.
- `rmdir()` : qovluğu silir. Əgər qovluq boş deyilsə `OSError` xətasını verəcək.

```
# Öncə yaratdığımız salam.txt faylını silək
if txt_file.exists():
   txt_file.unlink()
   print(f"'{txt_file}' silindi.")
else:
   print(f"'{txt_file}' tapılmadı.")
```

Qovluğu içindəki fayllar və digər qovluqlarla bilrilkdə silmək üçün, belə bir funksiya yazmaq olar. Funksiya - ötürülmüş qovluqdakı bütün elementləri silir: fayldırsa - `.unlink` metodu ilə silir, qovluqdursa - rekursiv olaraq özünü həmin qovluq üçün çağırır. Qovluqdakı bütün fayl və içqovluqlar silindikdən sonra artıq boş qalmış qovluğun özü də silinir. İçərisində nələrsə saxlayan qovluq bir növ ağac strukturunu xatırlatdığı üçün, bu cür əməliyyata bəzən "removing tree" - yəni ağac strukturunun silinməsi də deyilir:

```
def rm_tree(path: Path)->bool:
    if not path.is_dir():
        print(f"{path.as_posix()} must be a directory!")
        return False
    if not path.exists():
        print(f"{path.as_posix()} does not exist!")
        return False

    for child in path.iterdir():
        if child.is_file():
            print(f"Removing file: '{child}'")
            child.unlink()
        else:
            print(f"Removing directory: {child}")
            rm_tree(child)
    path.rmdir()
    print(f"'{path}' deleted.")
    return True

test_dir = Path.cwd() / "test_dir"
rm_tree(test_dir)
```

Əslində, Pythonda `shutil` kitabxanası və onun `rmtree` metodu var, bu metodun köməyi ilə həmçinin bo. olmayan qovluğu silmək mümkündür:

```
import shutil

path = "<silinəcək qovluğun yolu>"
shutil.rmtree(path)
```

Ad Dəyişmək və Köçürmək

- `rename(target)` : Fayl və ya qovluğun adını dəyişir və ya onu başqa yerə köçürür. `target` yeni yol və ya ad ola bilər.

```
kohne_ad = Path.cwd() / "data.bin"
yeni_ad = Path.cwd() / "binary_data.dat"
yeni_yer = Path.cwd() / "dir1" / "binary_data_moved.dat"

if kohne_ad.exists():
    kohne_ad.rename(yeni_ad)
    print(f"'{kohne_ad.name}' adı '{yeni_ad.name}' olaraq dəyişdirildi.")

if yeni_ad.exists():
    yeni_ad.rename(yeni_yer) # həm adını dəyişir, həm də dir1-ə köçürür
    print(f"'{yeni_ad.name}' '{yeni_yer}' ünvanına köçürüldü və adı
dəyişdirildi.")
```

Mütləq Yol

- `resolve(strict=False)` : Simvolik linkləri (symbolic links) həll edir, `..`, `.` kimi nisbi yol elementlərini normallaşdıraraq yolun mütləq (absolute) formasını qaytarır. `strict=True` olarsa və yol mövcud deyilsə `FileNotFoundError` verəcək. Yadınıza salın ki, `.` - cari qovluq, `..` - isə bir səviyyə yuxarı qovluq, valideyn qovluq olacaq.

```
nisbi_yol = Path("./dir1/../dir2/./sub_dir/fayl2.txt")
print(f"Nisbi yol: {nisbi_yol}")
print(f"Mütləq yol (resolve): {nisbi_yol.resolve()}")
```

Əlavə olaraq, önünüzə `~` işarəsi də çıxa bilər. Bu işarə istifadəçinin ev qovluğunu ifadə edir. Ancaq, bu işarə yolun bir hissəsi olduqda, `resolve` metodu onu düzgün emal edə bilmir. Özündə bu simvolu saxlayan yolu mütləq yola çevirmək, onu açmaq üçün - `expanduser()` metodundan istifadə etmək olar:

```
tengo@EWave:~$ pwd
    /home/tengo
tengo@EWave:~$ python3
>>> from pathlib import Path
>>> Path.cwd()
    PosixPath('/home/tengo')
>>> Path("~/Documents")
    PosixPath('~/Documents')
>>> Path("~/Documents").resolve()
    PosixPath('/home/tengo/~/Documents')
>>> Path("~/Documents").expanduser()
    PosixPath('/home/tengo/Documents')
```

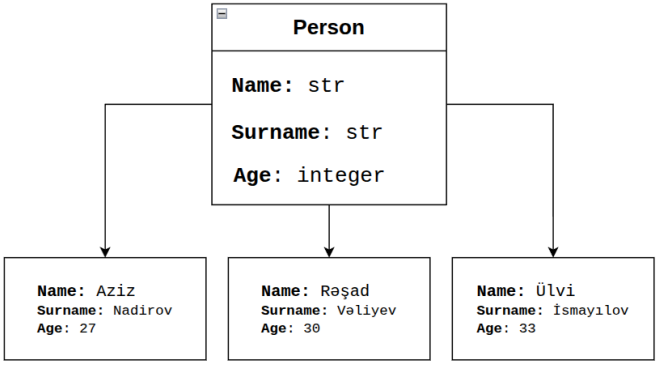
Tapşırıqlar

1. Bir path alaraq, onun haqqında məlumatları çap edən `get_info` funksiyasını yazın. Aşağıdakıları çap etməyə çalışın:
   - Yolun tipi - fayl ya qovluq olması
   - həcmi - MB, KB və sairə
   - Yaradılma tarixi
   - Son dəyişmə tarixi
2. Qovluğun yolunu alıb, içərisindəki elementləri nömrələnmiş və "gözəl" şəkildə format edilmiş halda çap edəcək `ls` funksiyasını yazın.
3. Öncəki `get_info` və `ls` funksiyalarından istifadə edərək, bir `file_browser` funksiyasını yazın. Bu funksiya ilə, ötürülmüş yoldakı faylların siyahısına baxmaq(`ls`), sıra nömrəsinə görə seçilmiş qovluğa keçib içərisinə baxmaq olar. Seçilən qovluqdursa, içərisindəkiləri `ls` edirik, fayldırsa - `get_info` ilə məlumatları çap edirik. İstifadəçi hər zaman "b" daxil edərək valideyn qovluğuna keçə, "q" daxil edərək menyunu tərk edə və ya cari nömrələnmiş(əgər qovluqda `ls` nəticəsinə baxırsa) siyahıdan elementin sıra nömrəsini daxil edə bilər.

Link: https://github.com/AzizNadirov/python-road-book/blob/master/code/tasks/std_lib_pathlib_9_5.ipynb

9.6 Data Siniflər

İndiyə qədər datanı necəsə strukturlaşdırmaq üçün nələrdən istifadə etmişik - lüğətlər, siyahılar, hətta siniflərdən belə istifadə etmişik. Siniflərdən istifadə etməyin ən böyük müsbət cəhəti - verilənləri *verilənlər modeli*-nə uyğun saxlamaqdır. *Verilənlərin Modeli* və ya *Data Model* - verilənlər qrupunu bir sinif kimi təsvir etməyə imkan verir. Model deyərkən, bu modelə uyğun verilənlərin hansı strukturda, hansı sahələrdən ibarət olacağını özündə əks etdirən sinif nəzərdə tutulur. Məsələn, ən sadə şəxs - `Person` modelini qrafik formada belə təsvir etmək olar:



Yuxarıda - `Person` blokunda model təyin olunur, bu modelə əsasən - verilənin 3 sahəsi olmalıdır: sətr tipli `Name` ,s sətr tipli `Surname` və tam ədəd tipli `Age` . Modelin hər bir nüsxəsi bu sahələrə malik olacaq. Pythonda bu cür modelləri təyin etmək yollarından biri - *data siniflərdir*. Data siniflər, verilənlərin modellərini təsvir etmək üçün cilalanmış adi siniflərdir. Yuxarıdakı `Person` modelini data siniflər vasitəsilə təyin edək:

```
from dataclasses import dataclass

@dataclass
class Person:
    Name: str
    Surname: str
    Age: str
```

İlk öncə `dataclass` dekoratorunu import edirik. Daha sonra, modeli `@dataclass` ilə bükərək adi sıravi sinif kimi təyin edirik. Heç bir metod təyin etmədən, data modelin sahələrini adi sinif xüsusiyyətləri kimi sadalayırıq. Fikir verin, model adi sinifdir, heç bir irsiyyət tələb olunmur, sadəcə tək bir dekorator ilə. İndi isə, bu model əsasında, bir neçə data nüsxə yaradaq:

```
aziz = Person("əziz", "Nadirov", 27)
reshad = Person(Name="Rəşad", Surname="Vəliyev", Age=30)
ulvi = Person("Ülvi", "İsmayılov", 33)

print(aziz)
```

```
print(reshad)
print(ulvi)
```

```
Person(Name='Əziz', Surname='Nadirov', Age=27)
Person(Name='Rəşad', Surname='Vəliyev', Age=30)
Person(Name='Ülvi', Surname='İsmayılov', Age=33)
```

Sinif nüsxələrinin çap olunma tərzinə fikir verin - adi siniflərdən fərqli olaraq, daha oxumlu şəkildə görünür, çünki data siniflərin `__repr__`, `__str__` metodları dataclass tərəfindən dəyişdirilərək bu hala gətirilir. Biz bu nüsxələrin xüsusiyyətlərinə nöqtə müraciəti ilə yanaşa bilərik:

```
print(aziz.Name)    # Əziz
```

Data Siniflərin müqayisəsi. Data sinif nüsxələrini bir-biri ilə müqaisə edə bilərik. Əgər iki nüsxənin bütün sahələri bərabərdirsə, o zaman nüsxələr də bərabər hesab edilir:

```
aziz = Person("Əziz", "Nadirov", 27)
aziz2 = Person("Əziz", "Nadirov", 27)
print(aziz == aziz2)    # True
```

Onların yalnız bərabərliyini deyil, həmçinin dəyər olaraq böyük və ya kiçik olmasını yoxlamaq olar. Nüsxələr üçün `>`, `<` istifadə etməkçün modeli bükdüyümüz dekoratorda `order=True` ötürmək lazımdır: `@dataclass(order=True)`. Belə modelin nüsxələrini müqayisə edərkən, bütün sahələr birincidən sonuncuya qədər müqayisə ediləcək:

```
@dataclass(order=True)
class Car:
    engine_volume: float
    door_count: int

bmw = Car(2.0, 4)
merc = Car(2.4, 2)
print(f"BMW > Merc: {bmw > merc}")
print(f"BMW < Merc: {bmw < merc}")
print(sorted([merc, bmw], reverse=True))
```

```
BMW > Merc: False
BMW < Merc: True
[Car(engine_volume=2.4, door_count=2), Car(engine_volume=2.0, door_count=4)]
```

Datasiniflərin dəyişiləbilənliyi barədə. Datasinif - data nüsxələri yaratmaq üçün bir qəlibdir. Hər bir nüsxə texniki olaraq özü-özlüyündə data sinfin bir nüsxəsidir. Yadınızdadırsa, biz nüsxənin

xüsusiyyətlərini mənimsətmə yolu ilə dəyişə bilirdik:

```
class SimpleClass:
       def __init__(self):
           self.a = 1

instance = SimpleClass()
print(f"[1] instance.a: {instance.a}" ) # [1] instance.a: 1
instance.a = -1     # nüsxənin xüsusiyyətini dəyişirik.
print(f"[2] instance.a: {instance.a}" ) # [2] instance.a: -1
```

Eyni şeyi əslində datasiniflərin nüsxələri ilə də edə bilirik:

```
@dataclass
class Person:
    name: str
    age: int

isi = Person("Ismayil", 26)
print(isi.age)  # 26
isi.age = 99    # data nüsxənin sahəsinin dəyərini dəyişirik.
print(isi.age)  # 99
```

İndi isə, fikirləşək - data nüsxənin sahələrini dəyişmək bizə necəsə xələl yetirə bilər? Dəyişiləbilən və dəyişiləbilməyən tiplər barədə danışarkən qeyd etmişdik ki, dəyişiləbilməyən tiplərdən istifadənin müsbət cəhətlərindən biri - səhvən dəyişilmənin mümkünsüzlüyüdür. Çünki, təsadüfən hər hansı funksiya daxilində obyekti dəyişib özümüzçün tapılması çətin bir xəta yarada bilərik. Data modelin nüsxələri üçün isə bu - verilənlərin təhrifi kimi sayıla bilər. Odur ki, data siniflərdə onun nüsxələrinin dəyişilə bilib-bilməməsini təyin etmək olar. Bunun üçün `@dataclass` dekoratoruna `frozen=True` ötürmək lazımdır. Bu zaman, bu sinfin nüsxələrinin sahələri bir növ "dondurulacaq" və dəyişilməyən olacaq:

```
@dataclass(frozen=True)
class Person:
    name: str
    age: int

isi = Person("Ismayil", 26)
print(isi.age)  # 26
isi.age = 99    # FrozenInstanceError: cannot assign to field 'age'
```

Yuxarıdakı nümunədə, `frozen=True` olduğu üçün, `isi.age = 99` kimi nüsxənin sahəsini dəyişməyə çalışdıqda `FrozenInstanceError` xətasını alırıq. Amma, unutmaq olmaz ki, sahə dayişiləbilən tipli obyekt olarsa, o zaman onu birbaşa dəyişə bilirik:

```
@dataclass(frozen=True)
class Person:
    name: str
    age: int
    some_list: list
```

```
isi = Person(name="Ismayil", age=26, some_list=[1, 2, 3])
isi.some_list.append(4)  # birbaşa siyahı obyektini dəyişirik.
print(isi.some_list)     # [1,2,3,4]
```

Əgər datasinif dondurulmuş olarsa(`frozen=True` kimi təyin olunarsa) və heç bir dəyişiləbilən tipli sahə olmazsa, o zaman data nüsxələr həş kodlara sahib olacaqlar və bu o deməkdir ki, biz onları lap çoxluq içərisində belə istifadə edə bilərik:

```
@dataclass(frozen=True)
class Person:
    name: str
    age: int

isi = Person(name="Ismayil", age=26)
isi2 = Person(name="Ismayil", age=26)
isi_past = Person(name="Ismayil", age=25)

print(hash(isi) == hash(isi2))  # True, eynidirlər.
print(hash(isi) == hash(isi_past))  # False

print(set([isi, isi2, isi_past]))   # təkrarlar(eyni həşlilər) atılır
# {Person(name='Ismayil', age=25), Person(name='Ismayil', age=26)}
```

Dəyişiləbilən tipli sahələrin susmaya görə dəyərləri. Datasinifdə sahələrin susmaya görə dəyərlərini təyin edə bilərik. Bu zaman, nəzərə almaq lazımdır ki, funksiyaların başlıqlarında olduğu kimi, burda da ilk öncə - susmaya görə dəyərləri olmayanlar(zərurilər), daha sonra susmaya görə dəyərləri olan sahələri(şərtilər) sadalamaq lazımdır.

```
@dataclass(frozen=True)
class Person:
   name: str
   age: int = 0

p = Person("John")
print(p)    # Person(name='John', age=0)
```

Susmaya görə dəyər dəyişiləbilən tipli olarsa, python `ValueError` xətasını atacaq. Çünki, bu zaman data sinifdə təyin etdiyimiz tək bir obyekti bütün yaradılacaq nüsxələrə bağlamaq təhlükəsi yarana bilər. Buna görə də, susmaya görə dəyər dəyişiləbilən olarsa, xəta alırıq. Aşağıdakı nümunədə `order_ids` sahəsi üçün susmaya görə dəyəri boş siyahı obyekti `[]` kimi təyin etməyə çalışırıq, siyahı isə dəyişiləbilən obyekt olduğu üçün xəta alacağıq:

```
@dataclass
class User:
    name: str
    order_ids: list[int] = []

john = User("John", [1, 2, 3])
```

```
# ValueError: mutable default <class 'list'> for field order_ids is not
allowed: use default_factory
```

Bəs nə etmək olar? Əslində, sonda aldığımız xəta ismarıcı özü bizə çıxış yolunu göstərir - `default_factory` istifadə etmək! Məsələ burasındadır ki, biz sahələri `dataclasses.field` funksiyası ilə də təyin edə bilərik. Burada bizə maraqlı olan funksiyanın ilk iki arqumentidir: `default` və `default_factory`. Ümumilikdə `field` özündən datasinfin bir sahəsini ehtiva edir. `default` - sahənin susmaya görə dəyəri olacaq. `field` istifadə etməyəndə biz sadəcə susmaya görə dəyəri sahəyə mənimsədirik. `default_factory` isə, susmaya görə dəyəri qaytaran, onu yaradan bir funksiya obyektidir. Yəni, əgər susmaya görə dəyəri almaq üçün hər hansı funksiyanı çağırmaq lazımdırsa, onu `default_factory` kimi təyin edirik. Bundan istifadə edərək, `order_ids` sahəsinin susmaya görə dəyərini `field` ilə belə təyin edə bilərik:

```
from dataclasses import dataclass, field

@dataclass
class User:
    name: str
    order_ids: list[int] = field(default_factory=list)

john = User("John", [1, 2, 3])
print(john) # User(name='John', order_ids=[1, 2, 3])
```

Yadda saxlamaq lazımdır ki, `default_factory` ilə göstərilən funksiya heç bir parametr almamalıdır.

Daha bir nümunə. Gəlin, daha bir model yaradaq.

```
from dataclasses import dataclass, asdict, field

def get_default_order()->list['Order']:
    return [Order(title='Take your gift!', id=1, product_id=0), ]

@dataclass
class Order:
    title: str
    id: int
    product_id: int

@dataclass
class User:
    name: str
    surname: str
    orders: list[Order]=field(default_factory=get_default_order)
```

Yuxarıdakı nümunədə `User` datasinfi vasitəsilə model təyin olunub. Modelin son `orders` (sifarişlər) sahəsinin özü `Order` (sifariş) nüsxələrindən ibarət bir siyahıdırş Yəni, modelin sahəsi - başqa bir modelin nüsxəsi ola bilər. Sahənin susmaya görə dəyəri `get_default_order` funksiyası ilə təyin olunur, funksiya bir növ tək hədiyyə sifarişindən ibarət siyahı qaytarır. Fikir verin:

1. `User.orders` sahəsinin susmaya görə dəyəri olduğu üçün `User` içərisində ardıcıllıqda ən son təyin edirik.
2. `User.orders` sahəsinin tipi siyahı - dəyişiləbilən olduğu üçün `default_factory` istifadə etdik. Susmaya görə dəyər qaytaran fabtik funksiyamız isə heç bir arqument qəbul etmir və gözlədiyimiz `list[Order]` qaytarır. İndi isə, bir user - istifadəçi yaradaq:

```
john = User(name='John', surname='Doe')
print(john)
# User(name='John', surname='Doe', orders=[Order(title='Take your gift!', id=1,
product_id=0)])
```

Göründüyü kimi, `john` nüsxəsini təyin edərkən, `orders` sahəsi üçün heç bir dəyər ötürmədik və susmaya görə dəyər yerini aldı və `john.orders` susmaya görə dəyəri `[Order(title='Take your gift!', id=1, product_id=0)]` oldu, tip olaraq sifarişlər siyahısı - `list[Order]` . Daha bir istifadəçi yaradıb, bu dəfə mövcud sifarişləri ötürək:

```
ali = User(name='Ali',
           surname='Aliyev',
           orders=[
              Order(title='Phone + case bundle', id=1, product_id=2),
              Order(title='Retro lovers: Nokia 6233', id=5, product_id=56)
           ])
print(ali)
# User(name='Ali', surname='Aliyev', orders=[Order(title='Phone + case bundle',
id=1, product_id=2), Order(title='Retro lovers: Nokia 6233', id=5,
product_id=56)])
```

Yuxarıda datasinfi təyin edərkən import etdiyimiz `asdict` funksiyası ilə data nüsxələri python lüğətlərinə çevirə bilərik:

```
ali_dict = asdict(ali)
print(ali_dict)
```

```
{'name': 'Ali',
 'surname': 'Aliyev',
 'orders': [
     {'title': 'Phone + case bundle', 'id': 1, 'product_id': 2},
     {'title': 'Retro lovers: Nokia 6233', 'id': 5, 'product_id': 56}
    ]
}
```

Datasiniflər olduqca güclü, müasir və rahat alətlərdir. Onlardan yerli-yerində istifadə edərək, böyük, açarlarının yadda saxlanılması tələb olunan lüğətləri məhz data siniflərlə əvəzləmək olar. Üstəlik, nəzərə alsaq ki VS Code, PyCharm kimi müasir İDE-lər onların vasitəsilə istifadəçilərə ipucular verir, datasiniflərdən istifadə etmək daha da asanlaşır.

Warning

- Data sinifllərin sahələrini təyin edərkən tip göstəricilərindən istifadə edirik. Ancaq, bu göstəricilər sadəcə məsləhət xarakteri daşıyır. Yəni, sahəni təyin edərkən bir tip göstərib, daha sonra nüsxəni yaradarkən tamam başqa bir tip ötürsək, python tərəfindən heç bir xəta almayacağıq.
- Yuxarıdakı nümunələrdə "nələrsə siyahısı" göstərmək üçün `list[...]` kimi yazmışam, məsələn, `list[str]` sətrlər siyahısı deməkdir. Lakin, `list` kimi qurma funksiya-tiplərdən bu cür tip göstərici kimi istifadə etmək yalnız Python 3.9 və daha sonrakı versiyalarda mümkündür. Daha öncəki versiyalarda isə `typing.List` istifadə etmək lazımdır, Yəni `from typing import List` import edib `List` (ilk hərfi böyük) istifadə etmək.

Tip

Datasiniflərin `__post_init__` adlı xüsusi metodu var. Bu metod nüsxə yaradıldıqdan sonra çağrılır. Bu metodu öz data siniflərimizdə təyin edərək müxtəif yoxlamalar aparmaq olar:

```
@dataclass
class Order:
   title: str
   id: int
   product_id: int

    def __post_init__(self):
         if self.id < 0:
             raise ValueError("Incorrect order id")
```

Ancaq, data modellərini bu cür yoxlamaq, validasiya etmək üçün üçüncü tərəf kitabxanası - `pydantic` daha məsləhətlidir.
Rəsmi veb sayt: https://pydantic.dev/

Tapşırıqlar

1. Aşağıdakı annotasiyadan istifadə edərək sadə kitab - `Book` modeli yazın. Model sadə müqayisə əməllərini dəstəkləməlidir.

```
from dataclasses import dataclass

@dataclass
class Book:
    """
    Book model with basic information.

    Attributes:
        title: Book title
        author: Author name
        year: Publication year
        pages: Number of pages
    """
    pass


book1 = Book("1984", "George Orwell", 1949, 328)
book2 = Book("1984", "George Orwell", 1949, 328)
book3 = Book("Animal Farm", "George Orwell", 1945, 112)

# Test equality
print(f"book1 == book2: {book1 == book2}")  # True
print(f"book1 == book3: {book1 == book3}")  # False

# Test string representation
print(f"Book info: {book1}")
# Book(title='1984', author='George Orwell', year=1949, pages=328)
```

2. Tələbə-qiymət modeli yaradın. Tələbənin qiymətləri - 4 dənə 60-100 arası ədəddən ibarət siyahıdır. Tələbənin orta qiymətini və bu orta qiymətə görə hərfi bal qaytaran metodlar yazın.

```
from dataclasses import dataclass

@dataclass(order=True)
class Student:
    """
    Student model with grades.

    Students are compared by average_grade first, then by name.
    """
    name: str
    surname: str
    grades: list[int]  # List of grades (60-100)

    def calculate_average(self) -> float:
        """Calculate average grade"""
        pass

    def get_letter_grade(self) -> str:
```

```
            """
            Convert average to letter grade.

            90-100: A
            80-89: B
            70-79: C
            60-69: D
            """
            pass


    students = [
        Student("Ali", "Aliyev", [85, 90, 88, 92]),
        Student("Aysel", "Mammadova", [95, 98, 93, 96]),
        Student("Kamran", "Hasanov", [70, 75, 72, 68]),
        Student("Leyla", "Ismayilova", [85, 90, 88, 92])  # Same avg as Ali
    ]

    for student in students:
        avg = student.calculate_average()
        letter = student.get_letter_grade()
        print(f"{student.name} {student.surname}: {avg:.2f} ({letter})")

    # Sort students by grade (highest first)
    sorted_students = sorted(students, reverse=True)
    print("\n=== Top Students ===")
    for i, student in enumerate(sorted_students, 1):
        print(f"{i}. {student.name} {student.surname}:
{student.calculate_average():.2f}")
```

Link: https://github.com/AzizNadirov/python-road-book/blob/master/code/tasks/std_lib_dataclasses_9_6.ipynb

10. Pythonda modullar və Paketlər

> **Info**
>
> Bu bölmədən etibarən Pythonda daha professional və tətbiqi hissə olan paketlər və onların idarəedilməsi haqqda danışacağıq. İlk öncə paketin nə olduğunu və paketlər daxilində necə import edə biləcəyimizi öyrənəcəyik. Daha sonra, virtual mühitlər vasitəsilə hər bir layihə üçün necə "ayrıca müstəqil" python interpretatoru istifadə edə bilərik - bu haqqda danışacağıq. Bundan sonra, kənar kitabxanaları necə yükləyib-silə biləcəyimizə baxacağıq. Son olaraq sizi, bütün bu söhbəti asanlaşdıracaq kənar bir alət ilə tanış edəcəm.

10.1 Pythonda Modullar

İndiyə qədər yazmış olduğumuz bütün kodlar demək olar ki bir modul, bir `.py` skripti içərisində idi. Yalnız standart kitabxanadan hər hansı aləti `import`, `from` vasitəsilə öz kodumuza əlavə edirdik, "import" edirdik. Sizə bir sirr açım - biz təkcə standart kitabxana alətlərini yox, başqa modullarda yerləşən öz adlarımızı da import edə bilərik.

Proqram məntiqinin modulyarlığı

Artıq "parçala və hökm sür" yanaşması ilə tanışsınız. İndiyədək kifayət qədər qəliz adlandırıla biləcək proqram məntiqi yazmışıq. Uzun və mürəkkəb proqram kodu yazarkən, bu kodu aydın və müstəqil funksiyalar, siniflər şəklində yazmağa çalışmışıq. Bir modul daxilində olan kodun dizaynı məhz bu cür olmalıdır. Niyə "bir modul daxilində olan" ? Çünki, daha mürəkkəb - müxtəlif məntiqli əməliyyatları icra edən hissələrdən - komponentlərdən ibarət olan proqram yazanda, artıq bütün bu məntiqi tək bir modulun içərisində saxlamaq mümkün olmayacaq. Təsəvvür edin ki tək bir `my_calc.py` adlı modulunuz var və bu modulun içərisində ağıllı valyuta hesablayıcı proqramı yazmısınız, hesablayıcı - müxtəlif qabaqcıl riyazi əməliyyatları icra edə bilir, yeri gələndə məzənnəyə uyğun olaraq valyuta ilə də əməliyyatlar apara bilir. Belə bir meqa-hesablayıcı sizcə tək bir modul içərisində yazılarsa, neçə sətr yer tutar? - Çox. Əslində, kod sətrlərinin sayı heç də əsas motiv deyil, əsas məsələ - kodu məntiqi komponentlərə görə parçalamaqdır. Deyək ki, öncəki meqa-hesablayıcı proqramının arxitekturasını dizayn edirsiniz. İlk baxışdan görünür ki, burda ən azından sırf riyazi funksiyalar və məzənnə ilə əlaqəli məqamlar olacaq. Deməli bu məntiqləri fərqli fayllarda yazıb, sonra isə üçüncü - əsas modulda birləşdirmək olar. Deyək ki, proyekti `my_calc_app` qovluğunda başlatmışıq və riyazi məntiqi `math_logic.py`, məzənnə ilə bağlı məntiqi isə `currency_logic.py` modulunda yazmışıq. O zaman proyektin strukturu belə olar:

```
| my_calc_app/
    | math_logic.py
    | currency_logic.py
    my_calc.py
```

Modullar:

```
# my_calc_app/math_logic.py
def do_math():
```

```
    pass
```

```
# my_calc_app/currency_logic.py
def do_currency():
    pass
```

```
# my_calc_app/my_calc.py
from math_logic import do_math
from currency_logic import do_currency

def calc():
    do_math()
    do_currency()
```

Yuxarıdakı oyuncaq nümunədə iki modul: `math_logic` və `currency_logic` var, hər modulda bir boş funksiya yazılıb. Proqramın giriş nöqtəsi - `my_calc.py` modulu olacaq. Burda `from ... import ...` konstruksiyası ilə eyni qovluqda olan modullardan funksiyalarımızı import edirik.

`import` ya `from ... import`

Funksiyalar haqqında danlşarkən LEGB qaydası, adlar fəzaları haqqında danışmışdıq. Bu zaman biz bir modul daxilində yer alan, bir modulun adlar fəzası haqqında danışırdıq. İmport mexanizmi isə imkan verir ki, bir modulun adlar fəzasını - orada təyin olunmuş adları öz modulumuza əlavə edək. Bunu etməyin iki üsulu var: digər modulun bütün adlarını import etmək və digər modulun yalnız sadalanmış, lazım olan adlarını import etmək. `import ...` - digər modulun bütün adlarını modulumuza import edir. Bu zaman həmin modulun adlarından istifadə etmək üçün `<import_olunan_modulun_adı>.ad` kimi yazılışdan istifadə etmək lazımdır. Bu onun üçün edilmişdir ki, import edilmiş adlarla modulun öz adları konflikt yaratmasın(əslində, sadəcə hər sonra yaradılmış ad, əvvəlkini örtəcəkdi - overwriting baş verəcəkdi). Məsələn, öncəki `my_calc_app` nümunəsini belə yazmaq olardı:

```
# my_calc_app/my_calc.py
import math_logic
import currency_logic

def calc():
    math_logic.do_math()
    currency_logic.do_currency()
```

`from <modul_name> import <name>` isə moduldan ancaq lazım olan ad və ya adları import edir, bir neçə ad import etmək lazım olduqda vergül ilə sadalamaq olar.

```
# my_calc_app/my_calc.py
from math_logic import do_math
from currency_logic import do_currency

def calc():
```

```
    do_math()
    do_currency()
```

Əlavə olaraq, import zamanı import olunan ada ləqəb - *alias* təyin etmək olar, bunun üçün import olunan addan sonra `as` operatoru ilə ləqəb təyin edilir:

```
# my_calc_app/my_calc.py
from math_logic import do_math as math
from currency_logic import do_currency as currency

def calc():
    math()
    currency()
```

Nəticədə, import olunan ad, cari modulun adlar fəzasına öz adı ilə deyil, təyin olunan ləqəb ad kimi düşür və istifadə olunur. Cari modulda artıq belə bir ad varsa və ya import olunan ad çox uzundursa, ləqəbli import yardımçı ola bilər.

Dolayısı icra və `__name__` dəyişəni

Biz hər hansı kənar moduldan ad import etdikdə, həmin adın yaranması üçün import olunan modul bir dəfə icra edilməlidir. Təsəvvür edin ki, iki ədəd modul var:

```
# app/b.py
print("running module 'b'...")
B_MESSAGE = "Message: " + "b"

def greeting():
    return f"Hi! {B_MESSAGE}"
```

və

```
# app/main.py
from b import greeting
```

`b` modulu özündə iki ədəd ad təyin edir: `B_MESSAGE` adlı sətr və `greeting` adlı funksiya. Göründüyü kimi, funksiyanın işləməsi üçün də `B_MESSAGE` sətrindən istifadə olunur, çünki funksiyanın qaytardığı sətrin bir hissəsidir. Biz əsas `main.py` modulunda `greeting` funksiyasını import etdikdə, Python modu olan adları almaq üçün, həmin modulu bir dəfə icra edir. Modul icra olunandan sonra, içərisindəki adlar təyin olunur və import edən modul - `main.py` artıq hesablanmış, təyin olunmuş adları modulumuza təqdim edir - yalnız bundan sonra `from ... import ...` və ya `import ...` istifadə etməyimizdən asılı olaraq yazacağımız və ya bütün adlar modulumuzun adlar fəzasına daxil ediləcək. `b.py` icra edilərsə, `print` ismarıcımızı görməliyik. Əgər `main.py` modulunu işə salsaq: `running module 'b'...` görəcəyik. Hərçənd, biz `main.py` modulunda sadəcə tək bir import etmişik. Öncə dediyim kimi, Python import olunan moduldan adları almaq üçün, həmin modulu icra edir. Üstəlik, modul icra edildiyinə görə, import olunan adlar, onlara tələb olunan məlumatları alır: `greeting` funksiyasının işləməsi üçün `B_MESSAGE` sətri hesablanmalı idi, əks halda funksiyamız çox güman ki çalışmayacaqdı. Burdan yadda saxlanılmalı

olan əsas məqam: import zamanı, import olunan modul icra edilir! Hətta, əgər import olunan modulun özündə də importlar varsa, onlar da icra edilir. Məsələn, `main` modulu `a` istifadə edir, `a` modulu da deyək ki `b` modulundan nə isə import edir. Belə olan təqdirdə, biz `main` modulunu icra etdikdə, əvvəlcə `a` icra olunmağa çalışacaq, daha sonra `a` özü `b` modulundan asılı olduğu üçün, oradan nəsə import etdiyi üçün `b` modulu da icra ediləcək. Bu da bir növ icra zənciri yaratmış olur.

Modulun `__name__` attributu: Bilirik ki, modulu import edərkən, import olunan modul icra edilir. Deməli, hər hansı modul icra olunursa, deməli bu modulu ya özümüz işə salıb icra edirik, ya da bu modulu icra etdiyimiz başqa bir modula import etmişik. Bəs necə bilə bilərik ki, modul hansı səbəbdən icra olunur? Ola bilsin ki, modulda müyəyyən bir funksiya çağrılır və biz istəyirik ki bu funksiya yalnız və yalnız modulun özü bir başa icra edilərsə işə düşsün, yəni import səbəbilə modul işə düşübdürsə, funksiyamız çağrılmasın. Biz bunu, modul səviyyəsində istifadə oluna bilən `__name__` dəyişəni vasitəsilə təyin edə bilərik. Məsələ burasındadır ki, əgər biz hər hansı modulu birbaşa icra ediriksə, bu zaman `__name__` dəyişəninin dəyəri `__main__` olacaq, əks halda isə, modulun fayl adı olacaq. Deməli, əgər biz `b.py` modulundan nəsə import ediriksə, `b` modulunda `__name__` dəyişəninin dəyəri `__main__` deyil, `b` olacaq. Gəlin, `b.py` moduluna bir neçə sətr kod da əlavə edək:

```
# b.py
print("running module 'b'...")
print("__name__ for b: ", __name__)
B_MESSAGE = "Message: " + "b"

def greeting():
    return f"Hi! {B_MESSAGE}"

greeting()
```

Gördüyünüz kimi, modulun sonunda `greeting` funksiyasının çağırışı var. `main.py` isə dəyişməz olaraq qalsın:

```
# app/main.py
from b import greeting
```

Biz `main.py` modulunu icra etsək, importa görə `b` modulu icra ediləcək və oradakı `greeting()` işə düşəcək:

```
# python main.py
running module 'b'...
__name__ for b:  b
Hi! Message: b
__name__ for 'main':  __main__
```

İndi isə, fərz edək ki, biz `b` modulundakı `greeting` funksiyasının çağrılmasını ancaq o zaman etmək istəyirik ki, istifadəçi birbaşa olaraq `b` modulunu işə salmış olsun, yəni import zamanı dolayısı icra zamanı deyil. Elə olan halda, `__name__` dəyişəninin yoxlamaq olar, əgər `__name__ == "__maim__"` olarsa, bu o deməkdir ki, istifadəçi birbaşa olaraq `b` modulunu icra edib, məsələn `python b.py` kimi:

```
# b.py
print("running module 'b'...")
print("__name__ for b: ", __name__)
B_MESSAGE = "Message: " + "b"

def greeting():
    return f"Hi! {B_MESSAGE}"

if __name__ == "__main__":
    print(greeting())
```

Bundan sonra `main.py` modulunu icra etsək alarıq:

```
# python main.py
running module 'b'...
__name__ for b:  b
__name__ for 'main':  __main__
```

Yəni, bu dəfə `b` modulu dolayısı icra edildiyi üçün adı `"__main__"` deyil, öz adı - `"b"` olacaq. Praktikada `if __name__ == "__main__":` yazılışını çox görəcəksiniz, bunnu adətən modulun funksionalını yalnız birbaşa icra zamanı işə salmaq üçün edirlər, yəni importlar moduldakı funksionallığı təsadüfən işə salmasın deyə.

Əla yazıdır! Modullar mövzusunu çox aydın və anlaşılan dildə izah etmisiniz. Eyni tərzdə, Python-da Paketlər mövzusunu davam etdirə bilərik:

10.2 Pythonda Paketlər

Modulların proqram məntiqini ayrı-ayrı fayllara bölməyə necə kömək etdiyini artıq gördük. Bu, kiçik və orta ölçülü proyektlər üçün əla bir yanaşmadır. Bəs proyektimiz daha da böyüyərsə - onlarla, hətta yüzlərlə moduldan ibarət olarsa? Belə bir vəziyyətdə bütün modulları tək bir qovluqda saxlamaq yenə də qarışıqlığa səbəb ola bilər. Məhz bu zaman Python-un paketləri köməyimizə çatır. Paket - özündə Pyhton modullarını saxlayan adi qovluqdur. Sadəcə, qovluğa `__init__.py` faylı əlavə edilir və bu zaman python anlayacaq ki, bu qovluq paketdir. Bu fayl barədə bir azdan daha ətraflı danışacağıq.

Proyekt strukturunun paketlərə bölünməsi

Təsəvvür edin ki, əvvəlki `my_calc_app` nümunəmizdəki meqa-hesablayıcımız daha da mürəkkəbləşir. Bəlkə də, riyazi məntiqin özü də bir neçə alt hissəyə bölünməlidir: sadə əməliyyatlar, mürəkkəb funksiyalar, statistik hesablamalar və s. Yaxud, valyuta məntiqi ilə yanaşı, proqramımıza istifadəçi interfeysi (UI) və ya verilənlər bazası ilə işləmək üçün yeni komponentlər əlavə etmək istəyirik. Bütün bu fərqli məntiqlərə aid modulları eyni səviyyədə saxlamaq əvəzinə, onları məntiqi qovluqlarda – yəni paketlərdə – qruplaşdıra bilərik.

Gəlin meqa-hesablayıcı proyektimizi paketlərlə yenidən strukturlaşdıraq. Deyək ki, hesablayıcının əsas məntiqi üçün `calculator_core` adlı bir paketimiz və yardımçı funksiyalar üçün `utils` adlı başqa bir paketimiz olacaq. Proyektin strukturu təxminən belə olacaq:

```
| calc_app/
   | calculator_core/      # 'calculator_core' paketi
   |   | __init__.py
   |   | arithmetic_ops.py
   |   | currency_ops.py
   | utils/                # 'utils' paketi
   |   | __init__.py
   |   | formatter.py
   |   | logger.py
   main_app.py             # əsas giriş nöqtəsi
```

Burada `calculator_core` və `utils` qovluqları içərisində `__init__.py` faylı olduğu üçün onlar Python paketləri hesab olunur. `arithmetic_ops.py` və `currency_ops.py` modulları `calculator_core` paketinin bir hissəsidir, `formatter.py` və `logger.py` isə `utils` paketinin.

Paketlərin içindəki modullar:

```python
# calc_app/calculator_core/arithmetic_ops.py
def add(a, b):
   return a + b

def subtract(a, b):
   return a - b
```

```python
# calc_app/calculator_core/currency_ops.py
def convert_usd_to_eur(amount_usd, rate=0.92):
   return amount_usd * rate
```

```python
# calc_app/utils/formatter.py
def format_result(value, precision=2):
   return f"Result: {value:.{precision}f}"
```

`__init__.py` faylları hələlik boş ola bilər:

```python
# calc_app/calculator_core/__init__.py
# Bu fayl calculator_core qovluğunu paketə çevirir.
# Hələlik boşdur.
```

```python
# calc_app/utils/__init__.py
# Bu fayl utils qovluğunu paketə çevirir.
# Hələlik boşdur.
```

İndi isə `main_app.py` modulunda bu paketlərdəki funksiyaları import edək:

```python
# calc_app/main_app.py
from calculator_core.arithmetic_ops import add
from calculator_core.currency_ops import convert_usd_to_eur
```

```
from utils.formatter import format_result

sum_result = add(10, 5)
print(format_result(sum_result))

converted_amount = convert_usd_to_eur(100)
print(format_result(converted_amount, precision=3))

# Alternativ import üsulu
import calculator_core.arithmetic_ops
import utils.formatter

diff_result = calculator_core.arithmetic_ops.subtract(20, 7)
print(utils.formatter.format_result(diff_result))
```

Gördüyünüz kimi, paketlərdən import edərkən `paket_adı.modul_adı` sintaksisindən istifadə edirik. Bu, adlar fəzasını daha səliqəli, ayrıca saxlamağa kömək edir və fərqli paketlərdə eyni adlı modulların olmasına imkan yaradır (məsələn, `project_a.utils` və `project_b.utils`).

`__init__.py` faylının əlavə imkanları

`__init__.py` faylı təkcə qovluğu paketə çevirmir, həm də paket import edildikdə icra olunur. Bu o deməkdir ki, paket üçün hər hansı ilkin quraşdırma və ya başlanğıc kodunu bu faylda yerləşdirə bilərsiniz. Bundan əlavə, `__init__.py` faylı paketdən nələrin "asanlıqla" import edilə biləcəyini təyin etmək üçün istifadə oluna bilər. Məsələn, `calculator_core` paketinin istifadəçilərinin `add` funksiyasını birbaşa `calculator_core.add` kimi çağıra bilməsini istəyiriksə, `calculator_core/__init__.py` faylını belə dəyişə bilərik:

```
# calc_app/calculator_core/__init__.py
print("calculator_core paketi yüklənir...")
from .arithmetic_ops import add, subtract
from .currency_ops import convert_usd_to_eur

__all__ = ["add", "subtract", "convert_usd_to_eur"]
```

`__all__` siyahısı `from calculator_core import *` üçün nələrin import ediləcəyini təyin edir. Buradakı `from .arithmetic_ops import add` sətrindəki nöqtə (`.`) cari paketi (yəni `calculator_core`-u) işarə edir. Bu, nisbi import adlanır. İndi `main_app.py` faylında importu belə də etmək olar:

```
# my_mega_calculator_project/main_app.py
from calculator_core import add, convert_usd_to_eur # __init__.py sayəsində
from utils.formatter import format_result

sum_result = add(10, 5) # Birbaşa calculator_core-dan gəlir
print(format_result(sum_result))

converted_amount = convert_usd_to_eur(100) # Birbaşa calculator_core-dan gəlir
print(format_result(converted_amount, precision=3))
```

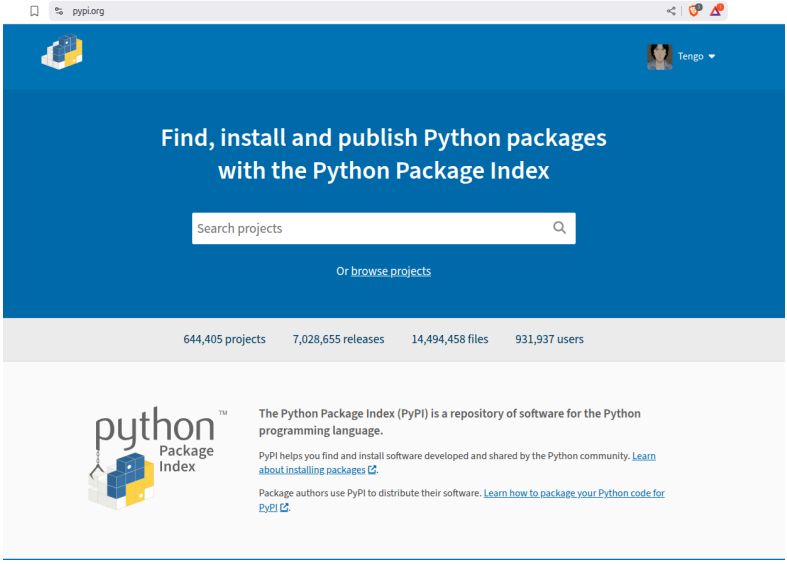
`__init__.py` -da `__all__` adlı xüsusi bir siyahı təyin etməklə, `from calculator_core import *` ifadəsi işlədildikdə hansı adların import ediləcəyini idarə edə bilərik. `__all__` təyin olunmayıbsa, `import *` yalnız `__init__.py` -da açıq şəkildə təyin olunan və ya import edilən adları (`_` ilə başlamayanları) import edəcək. Ümumiyyətlə, `import *` istifadəsindən qaçmaq lazımdır. Çünki, bu zaman moduldan nələrin import olunduğunu aydın görmək olmur, bu isə hər hansısa adın təsadüfən modul tərəfindən əvəzlənməsinə gətirib çıxarda bilər. Bunu da qismən `__init__.py` modulunda `__all__` siyahısını tənzimləməklə həll etmək olar.

10.3 Pythonda pip paket meneceri və üçüncü tərəf kitabxanalar

> **Info**
>
> Bu bölmədə Python dilində yazılmış üçüncü tərəf kitabxanaları haqqda, onların necə yüklənib quraşdırılması, silinməsi, asılılıqları haqqında danışacağıq. Üçüncü tərəf kitabxanası - Pythondan asılılığı olmayan, müstəqil tərtibatçıların yazdığı kitabxanalardır. Niyə üçüncü tərəf - çünki, bu kitabxanaları nə biz, nə də rəsmi Python tərəfindən yazılmayıb, üçüncü tərəf - hər hansı tərtibatçı, kompaniya(Google, Microsoft və s.) tərəfindən yazılıb. Əlavələr hissəsində bəzi faydalı və "janrının klassikası" sayılan kitanxanaların qısa təsviri olacaq.

Üçüncü tərəf kitabxanaları və `PyPI`

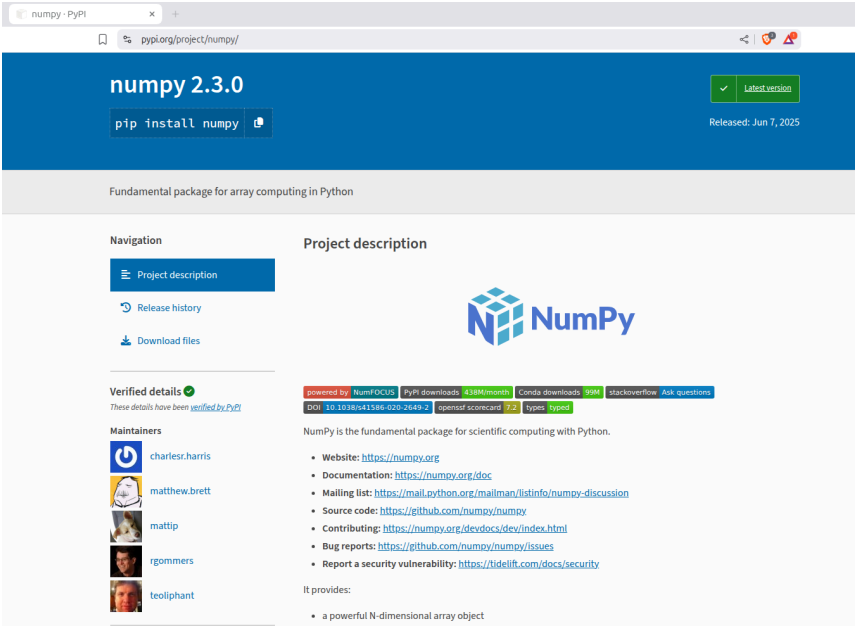
Başlıqda deyildiyi kimi, üçüncü tərəf kitabxanaları(*third party libraries*) - müstəqil tərtibatçıların yazdığı, Python standart kitabxanasına daxil olmayan kitabxanalardır.Tərtibatçılar - həm təklikdə yaşayıb yaradan qızıl insanlar, həm də hər hansı şirkət ola bilər. Məsələn, İT şirkətləri müəyyən servislər təqdim edir və bu servislərlə işləmək üçün Python kitabxanaları yazırlar. Yaxşı, bəs bu kitabxanaları necə əldə etmək olar? Bunun üçün Pythonın rəsmi reyst saytı - `pypi.org` var:



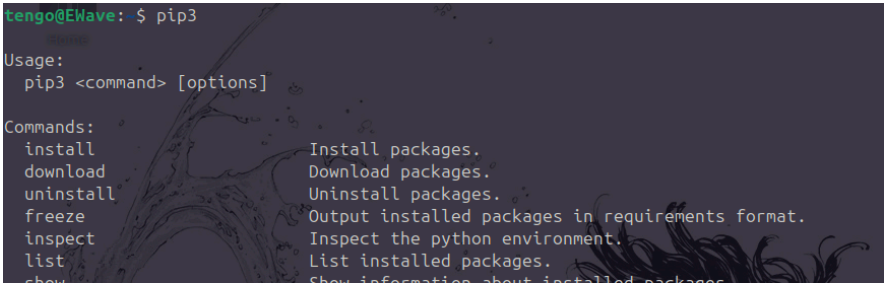
Bu sayt, platforma bir növ - reyestr, indekser rolunu oynayır. Tərtibatçılar buradan qeydiyyatdan keçərək, öz kitabxanalarını yükləyirlər. Hər bir kitabxana minimal yoxlanışdan keçir və sistemə əlavə edilir. Bundan sonra, istənilən Python istifadəçisi bu kitabxananı quraşdıra bilər.

`pip` paket meneceri

Yuxarıda öyrəndik ki, üçüncü tərəf kitabxanaları pypi reyestrində yer alır, sual yaranır - bəs, bu kitabxanaları öz lokal sistemimizə necə yükləyə bilərik? Bunun üçün Pythonda `pip` adlı paket meneceri var. Niyə məhz "paket"? Məsələ burasındadır ki, kitabxanalar paket şəklində qurulur və bu kitabxanaları quraşdırmaq, silmək və ya sadalamaq üçün `pip` istifadə edirik. Kitabxana kodunun özü "wheel" genişlənməli fayllar olur, bu fayllar quraşdırıldıqdan sonra, istifadə olunan Pythonın qovluğuna yazılır və sonrakı istifadə zamanı import oluna bilir. Gəlin, nümunə üçün `numpy` adlı kitabxananı quraşdırmağa çalışaq. PyPI-da axtarsaq alarıq:



Saytdan görə bilərik ki, kitabxana Pythonda elmi hesablamalar üçün nəzərdə tutulub, son versiyası `2.3.0`-dır və `pip install numpy` əmri ilə quraşdırıla bilər. Hələki bu əmri harasa yazmağa tələsməyək. Deməli, hər bir kitabxananın unikal, təkrarlanmaz adı və versiyası olur. Biz kitabxananı quraşdırarkən, quraşdırmaq istədiyimiz versiyanı da göstərə bilərik, amma əgər bunu etməsək, Python susmaya görə quraşdırıla biləcək ən son versiyanı seçəcək. Quraşdırmaq istədiyimiz kitabxananın adı "`numpy`"-dır. Quraşdırmaq üçün `pip` proqramından istifadə edirik və bu proqram Pythonla birlikdə quraşdırılır. İşdir, əgər pip yoxdursa(Linuxda ola bilər), belə olan halda terminalda `python -m ensurepip` əmrini icra etmək olar, bundan sonra terminalda `pip` və ya `pip3` yazaraq istifadə etmək olar:



Windows üçün əmrləri powershell və ya cmd əmrlər pəncərəsində yazmaq olar. Yuxarıdakı şəkildən də göründüyü kimi, `pip` üçün müəyyən əmrlər var: `install`, `download`, `uninstall` və sairə. Bizim istifadə edəcəyimiz `install` - paketi yükləyib quraşdırmaq üçün və `uninstall` - artıq qurulmuş paketi silmək, `list` - quraşdırılmış paketlərin siyahısını almaq üçün istifadə olunur. `install` və `uninstall` əmrlərindən sonra, quraşdırmaq və ya silmək istədiyimiz paketin adı yazılmalıdır. Məsələn `numpy` adlı kitabxananı quraşdırmaq istəyiriksə, `pip install numpy` kimi yazmalıyıq. İlk öncə, Pythonda quraşdırılan paketlərin siyahısına baxa bilərik:

```
(.tmp) tengo@EWave:~/Documents/book_cs_with_python$ pip list
Package    Version
---------- -------
pip        24.0
setuptools 65.5.0

[notice] A new release of pip is available: 24.0 -> 25.1.1
[notice] To update, run: pip install --upgrade pip
(.tmp) tengo@EWave:~/Documents/book_cs_with_python$
```

Mənim halımda ancaq iki paket var: `pip` və `setuptools`. Bu iki paket adətən öncədən quraşdırılmış olur, `pip` elə paket menecmentini həyata keçirən paketdir. İndi isə, `numpy` kitabxanasını quraşdırmağa çalışaq. Bunu edəndən sonra buna bənzər proses görəcəyik:

```
(.venv) tengo@EWave:~/Documents/book_cs_with_python$ pip install numpy
Collecting numpy
  Downloading numpy-2.3.0-cp311-cp311-manylinux_2_28_x86_64.whl.metadata (62 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 62.1/62.1 kB 546.5 kB/s eta 0:00:00
Downloading numpy-2.3.0-cp311-cp311-manylinux_2_28_x86_64.whl (16.9 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 16.9/16.9 MB 2.4 MB/s eta 0:00:00
Installing collected packages: numpy
Successfully installed numpy-2.3.0

[notice] A new release of pip is available: 24.0 -> 25.1.1
[notice] To update, run: pip install --upgrade pip
(.venv) tengo@EWave:~/Documents/book_cs_with_python$
```

Əmri icra etdikdə, ilk öncə paketin icra oluna biləcək versiyası tapılır, daha sonra isə həmin versiyalı paket yüklənməyə başlayır. Bundan sonra, yüklənmiş paket Python sisteminə quraşdırılır. İndi yenidən paketlərin siyahısına baxsaq:

```
(.tmp) tengo@EWave:~/Documents/book_cs_with_python$ pip list
Package    Version
---------- -------
numpy      2.3.0
pip        24.0
setuptools 65.5.0

[notice] A new release of pip is available: 24.0 -> 25.1.1
[notice] To update, run: pip install --upgrade pip
(.tmp) tengo@EWave:~/Documents/book_cs_with_python$
```

Şəkildən göründüyü kimi, `numpy` paketinin `2.3.0` yəni son versiyası quraşdırılmışdır. Bu o deməkdir ki, biz artıq Pythonı işə salıb, `numpy` kitabxanasını import edib işlədə bilərik:

```
(.tmp) tengo@EWave:~/Documents/book_cs_with_python$ python
Python 3.11.12 (main, Apr  9 2025, 08:55:55) [GCC 13.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy
>>> numpy.__version__
'2.3.0'
>>> numpy.__name__
'numpy'
>>>
```

Pythonda paketlərin asılılıqları - package dependencies

Yüklədiyimiz üçüncü tərəf paketlər də adi python kodudur və əksər hallarda bu paketlərin özləri hansısa başqa bir üçüncü tərəf Python paketindən istifadə edirlər. Təsəvvür edin ki, `pandas` adlı bir kitabxana var(irəlidə görəcəysiniz ki həqiqətən də var :D) və bu kitabxana öz kodunda `numpy` və `pytz` adlı digər kitabxanalardan istifadə edir. Bu o deməkdir ki, əgər biz `pandas` istifadə etmək istəyiriksə, onun istifadə etdiyi `numpy` və `pytz` kitabxanalarını da yükləməliyik. Bu cür tələblər - asılılıqlar adlanır. Əslində burda həmçinin paketlərin versiyaları da rol oynayır. Çünki, kitabxanalar yeniləndikcə, kodları dəyişdikcə funksionallıqları da dəyişir. Buna görə də, kitabxanalar bir-birinin konkret versiyalarından(və ya versiya aralıqlarından) asılı olur. Məsələn `pandas 2.0` paketi `numpy 2.7` tələb edə bilər. `pip` paket meneceri də bu cür asılılıqları həll etməyə çalışır. Hər hansı paketi quraşdırmağa çalışarkən, `pip` həmin paketin asılı olduğu digər paketlərin siyahısını da alır və bütün yığılacaq paketlər asılılıqlarını elə həll etməyə çalışır ki həm bütün asılılıqlar ödənsin, həm də ən son versiya kitabxanaları qurşdırmış olsun. Gəlin, öncə quraşdırdığımız `numpy` kitabxanasını silib, `pandas` kitabxanasını quraşdıraq:

```
(.tmp) tengo@EWave:~/Documents/book_cs_with_python$ pip uninstall numpy
Found existing installation: numpy 2.3.0
Uninstalling numpy-2.3.0:
  Would remove:
    /home/tengo/Documents/book_cs_with_python/.tmp/bin/f2py
    /home/tengo/Documents/book_cs_with_python/.tmp/bin/numpy-config
    /home/tengo/Documents/book_cs_with_python/.tmp/lib/python3.11/site-packages/numpy-2.3.0.dist-info/*
    /home/tengo/Documents/book_cs_with_python/.tmp/lib/python3.11/site-packages/numpy.libs/libgfortran-040039e1-0352e75f.so.5.0.0
    /home/tengo/Documents/book_cs_with_python/.tmp/lib/python3.11/site-packages/numpy.libs/libquadmath-96973f99-934c22de.so.0.0.0
    /home/tengo/Documents/book_cs_with_python/.tmp/lib/python3.11/site-packages/numpy.libs/libscipy_openblas64_-56d6093b.so
    /home/tengo/Documents/book_cs_with_python/.tmp/lib/python3.11/site-packages/numpy/*
Proceed (Y/n)? y
  Successfully uninstalled numpy-2.3.0
(.tmp) tengo@EWave:~/Documents/book_cs_with_python$ pip install pandas
Collecting pandas
  Downloading pandas-2.3.0-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (91 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 91.2/91.2 kB 552.0 kB/s eta 0:00:00
Collecting numpy>=1.23.2 (from pandas)
  Using cached numpy-2.3.0-cp311-cp311-manylinux_2_28_x86_64.whl.metadata (62 kB)
Collecting python-dateutil>=2.8.2 (from pandas)
  Using cached python_dateutil-2.9.0.post0-py2.py3-none-any.whl.metadata (8.4 kB)
Collecting pytz>=2020.1 (from pandas)
  Using cached pytz-2025.2-py2.py3-none-any.whl.metadata (22 kB)
Collecting tzdata>=2022.7 (from pandas)
  Using cached tzdata-2025.2-py2.py3-none-any.whl.metadata (1.4 kB)
Collecting six>=1.5 (from python-dateutil>=2.8.2->pandas)
  Using cached six-1.17.0-py2.py3-none-any.whl.metadata (1.7 kB)
Downloading pandas-2.3.0-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (12.4 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 12.4/12.4 MB 3.4 MB/s eta 0:00:00
Using cached numpy-2.3.0-cp311-cp311-manylinux_2_28_x86_64.whl (16.9 MB)
Using cached python_dateutil-2.9.0.post0-py2.py3-none-any.whl (229 kB)
Using cached pytz-2025.2-py2.py3-none-any.whl (509 kB)
Using cached tzdata-2025.2-py2.py3-none-any.whl (347 kB)
Using cached six-1.17.0-py2.py3-none-any.whl (11 kB)
Installing collected packages: pytz, tzdata, six, numpy, python-dateutil, pandas
Successfully installed numpy-2.3.0 pandas-2.3.0 python-dateutil-2.9.0.post0 pytz-2025.2 six-1.17.0 tzdata-2025.2

[notice] A new release of pip is available: 24.0 -> 25.1.1
[notice] To update, run: pip install --upgrade pip
```

Şəklə fikir verin: `numpy`, `python-dateutil`, `pytz`, `tzdata`, `six` sonda isə `pandas` özü yüklənir. Məsələ burasındadır ki, `pandas` bu kitabxanalardan asılıdır: `pandas` kitabxanası `pytz` tələb edir,

`pytz` özü isə `tzdata` tələb edir və sairə. `pandas` quraşdırıldıqdan sonra paketlərin siyahısına baxsaq:

```
tengo@EWave:~$ pip list
Package         Version
--------------- -----------
numpy           2.3.0
pandas          2.3.0
pip             24.0
python-dateutil 2.9.0.post0
pytz            2025.2
setuptools      65.5.0
six             1.17.0
tzdata          2025.2
```

Pythonda virtual mühitlər - venv

Artıq üçüncü tərəf kitabxanalarını quraşdıra və silə bilirik. Kitabxanalar kifayət qədər çox və olduqca fərqli təyinatlıdırlar, eynilə Pythonla edə biləcəyimiz şeylər kimi - biz paralel olaraq bir neçə proyekt apara bilərik, birində bir kitabxananı, digərində isə başqa bir qrup kitabxananı istifadə edə bilərik. Belə olan halda, biz məcburuq ki, dayanmadan bütün bu kitabxanaları sistemimizdə olan Pythona yükləyək, silək... Nəzərə alsaq ki kitabxanalar arasında asılılıqlar məsələsi də var - vəziyyət qəlizləşir. Vəziyyətdən çıxış yollarından biri - virtual mühit(*virtual environment*) yaratmaq ola bilər. Virtual mühit özündən - təcrid edilmiş ayrıca Python mühiti ehtiva edir. Məsələn biz layihəmizin qovluğunda bir Python virtual mühiti yaradaraq, layihə üçün lazım olan kitabxanaları məhz bu mühitə quraşdıra, layihəni icra etmək üçün də məhz bu mühitdən istifadə edə bilərik. Layihəni bitirdikdən sonra isə, mühiti silə bilərik. Beləliklə, əsas sistem Python yaddaşını lazım olmayacaq saysız kitabxanalarla zibilləməmiş oluruq.

Virtual mühiti yaratmaq üçün terminalı və ya Windows üçün əmrlər pəncərəsini mühiti yaratmaq istədiyimiz qovluqda açırıq. Daha sonra `python -m venv <yaradılacaq mühitin adı>` daxil edərək virtual mühiti yaradırıq. Aşağıda Linux üçün virtual mühitin yaradılması və bundan sonra qovluqda peydə olan fayllara baxış kodu verilib:

```
tengo@EWave:~/Documents/new_py$ pwd
    /home/tengo/Documents/new_py
tengo@EWave:~/Documents/new_py$ python3 -m venv my_venv
tengo@EWave:~/Documents/new_py$ ls
    my_venv
tengo@EWave:~/Documents/new_py$ ls ./my_venv/
    bin  include  lib  lib64  pyvenv.cfg
tengo@EWave:~/Documents/new_py$ ls ./my_venv/bin/
    activate      activate.fish  pip   pip3.12  python3
    activate.csh  Activate.ps1   pip3  python   python3.12
```

Yuxarıda nə baş verir: terminal `~/Documents/new_py` qovluğunda açılıb. Daha sonra, `python3 -m venv my_venv` əmri ilə `my_venv` adlı virtual mühit yaradılır, əməliyyat 1-3 saniyə zaman alır. Virtual mühit yarandırdan sonra, layihə qovluğunda virtual mühitlə eyni adda qovluq yaradılır, bizi maraqlandıran - qovluğun içərisindəki `bin/activate` faylıdır. Linuxda bu binar fayl vasitəsilə virtual

mühiti aktivləşdirmiş oluruq. Bunun üçün: `source <virtual_mühitin_adı>/bin/activate` əmrini icra edirik:

```
tengo@EWave:~/Documents/new_py$ source my_venv/bin/activate
(my_venv) tengo@EWave:~/Documents/new_py$
```

Fikir verin, virtual mühiti aktivləşdirdikdən sonra əmr sətrinin əvvəlinə `(my_venv)` sətri əlavə edildi. Bu onu göstərir ki, hal-hazırda biz öz virtual mühitimiz içərisində işləyirik. Virtual mühiti söndürmək, deaktiv etmək üçün `deactivate` əmrindən istifadə etmək olar:

```
(my_venv) tengo@EWave:~/Documents/new_py$ deactivate
tengo@EWave:~/Documents/new_py$
```

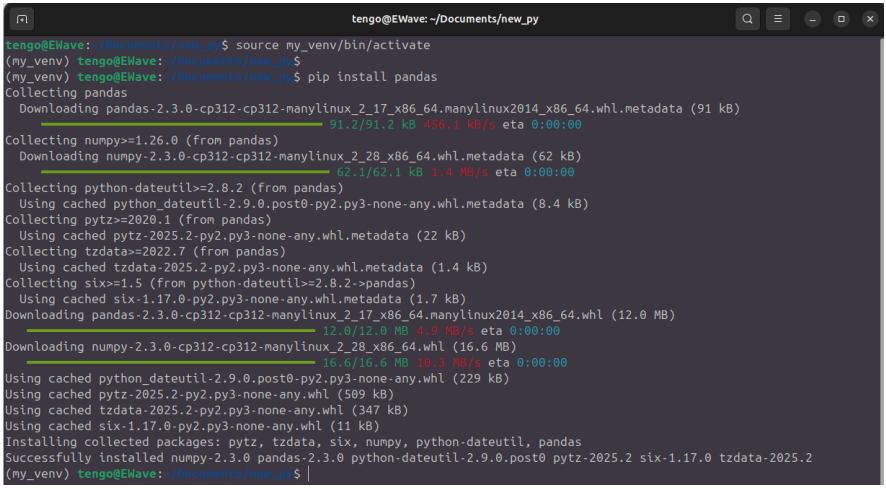
Gördüyünüz kimi, `deactivate` əmrindən sonra `(my_venv)` yazısı yox oldu, bu onu göstərir ki, virtual mühit deaktiv edildi. Linuxda olan `which` adlı əmrin vasitəsilə(WIndows üçün bu `where` olacaq), daxil edilən əmri icra edən binar proqramın harda olduğunu öyrənə bilərik. Məsələn virtual mühiti söndürdükdən sonra, biz terminalda `python` daxil edərkən işə düşən Python interpretatorunun harda yerlışdiyini belə öyrənə bilərik:

```
tengo@EWave:~/Documents/new_py$ which python3
    /usr/bin/python3
```

Bu o deməkdir ki, mənim sistem Python interpretatorum `/usr/bin/python3` ünvanında yerləşir. İndi isə, öncə yaratdığım virtual mühitit yenidən aktivləşdirib, eyni əmri bir daha icra etsəm:

```
tengo@EWave:~/Documents/new_py$ source my_venv/bin/activate
(my_venv) tengo@EWave:~/Documents/new_py$ which python3
    /home/tengo/Documents/new_py/my_venv/bin/python3
```

Gördüyünüz kimi, virtual mühiti aktivləşdirəndən sonra, `which python3` əmri mənə sistem interpretatorunun yolunu deyil, virtual mühitimizin ünvanını qaytarır. Bu bir daha onu göstərir ki, bizim icra edəcəyimiz Python məhz yaratdığımız virtual mühit daxilində olacaq. İndi isə, bu virtual mühitə bir neçə paket quraşdıraq:



Pythonı işə salıb, paketin versiyasına baxmağa çalışaq:

```
(my_venv) tengo@EWave:~/Documents/new_py$ python
    Python 3.12.3 (main, Feb  4 2025, 14:48:35) [GCC 13.3.0] on linux
    Type "help", "copyright", "credits" or "license" for more information.
>>> import pandas as pd
>>> pd.__version__
    '2.3.0'
```

Rəsmi sənədləşmədən götürülən aşağıdakı cədvəldə, müxtəlif platformalarda yaradılmış `<venv>` mühitini necə aktivləşdirilə biləcəyimiz verilib:

Platform	Shell	Command to activate virtual environment
POSIX	bash/zsh	`$ source <venv>/bin/activate`
	fish	`$ source <venv>/bin/activate.fish`
	csh/tcsh	`$ source <venv>/bin/activate.csh`
	pwsh	`$ <venv>/bin/Activate.ps1`
Windows	cmd.exe	`C:\> <venv>\Scripts\activate.bat`
	PowerShell	`PS C:\> <venv>\Scripts\Activate.ps1`

`requirements.txt` faylı:

`pip freeze` əmri bizə cari mühitdə(hər hansı virtual mühit aktiv olarsa həmin mühitin, əks halda sistem Python mühiti üçün) quraşdırılmış bütün paketlərin siyahısını onların versiyaları ilə birlikdə qaytarır:

```
(my_venv) tengo@EWave:~/Documents/new_py$ pip freeze
    numpy==2.3.0
    pandas==2.3.0
    python-dateutil==2.9.0.post0
```

```
    pytz==2025.2
    six==1.17.0
    tzdata==2025.2
```

Biz bu məlumatı bir fayla yazıb, daha sonra başqa bir mühitdə bu fayldan istifadə edərək, fayldakı bütün paketləri - verilən versiyalar ilə bir əmrlə quraşdıra bilərik. Linuxda hər hansı əmrin çıxışını fayla yönəltmək üçün `>` və ya `>>` istifad' olunur. `>` əmri - fayl mövcud deyilsə onu yaradır və içərisinə yazır, faylın içərisində öncədən nəsə varsa - silinir və əvəzinə yeni verilənlər yazılır, yəni fayllar bölməsindən öyrəndiyimiz `overwriting` baş verir. `>>` əmri isə, faylın içərisindəkiləri silməyir, sonuna əlavə edir, yəni `append` edir. Bizə lazım olan `>` əmridir. Gəlin, öncə yaratdığımız `my_venv` mühitinin paketlərini `requirements.txt` faylına yazaq və daha sonra həmin faylın içərisinə baxaq(bunun üçün Linuxda `cat` əmrindən istifadə edəcəyik):

```
(my_venv) tengo@EWave:~/Documents/new_py$ pip freeze > requirements.txt
(my_venv) tengo@EWave:~/Documents/new_py$ ls
    my_venv  requirements.txt
(my_venv) tengo@EWave:~/Documents/new_py$ cat ./requirements.txt
    numpy==2.3.0
    pandas==2.3.0
    python-dateutil==2.9.0.post0
    pytz==2025.2
    six==1.17.0
    tzdata==2025.2
```

Özünüz də, layihənin qovluğunu açıb, orada sözü gedən faylın yarandığına və içərisində məhz bu paketlərin olduğuna əmin ola bilərsiniz. Nəzərə alsaq ki mən bu nümunələri `Python 3.12.3` üzərində icra edirəm, sizdə fərqli Python versiyası olacağı təqdirdə, kitabxanaların da versiyası fərqlənə bilər, bu barədə narahat olmayın. İndi isə, virtual mühiti deaktivləşdirib, mühiti silək. Mühiti silmək üçün, mühitin öz adında yaratdığı qovluğu silmək kifayətdir. Terminal ilə bunu `rm -rf <venv name>` ilə etmək olar, ancaq siz hər hala qarşı sadəcə qovluğa keçib klik edərək silə bilərsiniz:

```
(my_venv) tengo@EWave:~/Documents/new_py$ deactivate
tengo@EWave:~/Documents/new_py$ rm -rf my_venv
tengo@EWave:~/Documents/new_py$ ls
    requirements.txt
```

Virtual mühiti sildikdən sonra geriyə ancaq öncə yaratdığımız `requirements.txt` faylı qalır. İndi isə, gəlin `.venv` adlı mühit yaradaq və `pip install -r ./requirements.txt` əmri ilə fayldakı bütün paketləri quraşdıraq:

```
tengo@EWave:~/Documents/new_py$ python3 -m venv .venv
tengo@EWave:~/Documents/new_py$ la
requirements.txt  .venv
tengo@EWave:~/Documents/new_py$ source .venv/bin/activate
(.venv) tengo@EWave:~/Documents/new_py$ which python
/home/tengo/Documents/new_py/.venv/bin/python
(.venv) tengo@EWave:~/Documents/new_py$ pip install -r requirements.txt
Collecting numpy==2.3.0 (from -r requirements.txt (line 1))
  Using cached numpy-2.3.0-cp312-cp312-manylinux_2_28_x86_64.whl.metadata (62
```

```
kB)
Collecting pandas==2.3.0 (from -r requirements.txt (line 2))
  Using cached pandas-2.3.0-cp312-cp312-
manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (91 kB)
Collecting python-dateutil==2.9.0.post0 (from -r requirements.txt (line 3))
  Using cached python_dateutil-2.9.0.post0-py2.py3-none-any.whl.metadata (8.4
kB)
Collecting pytz==2025.2 (from -r requirements.txt (line 4))
  Using cached pytz-2025.2-py2.py3-none-any.whl.metadata (22 kB)
Collecting six==1.17.0 (from -r requirements.txt (line 5))
  Using cached six-1.17.0-py2.py3-none-any.whl.metadata (1.7 kB)
Collecting tzdata==2025.2 (from -r requirements.txt (line 6))
  Using cached tzdata-2025.2-py2.py3-none-any.whl.metadata (1.4 kB)
Using cached numpy-2.3.0-cp312-cp312-manylinux_2_28_x86_64.whl (16.6 MB)
Using cached pandas-2.3.0-cp312-cp312-
manylinux_2_17_x86_64.manylinux2014_x86_64.whl (12.0 MB)
Using cached python_dateutil-2.9.0.post0-py2.py3-none-any.whl (229 kB)
Using cached pytz-2025.2-py2.py3-none-any.whl (509 kB)
Using cached six-1.17.0-py2.py3-none-any.whl (11 kB)
Using cached tzdata-2025.2-py2.py3-none-any.whl (347 kB)
Installing collected packages: pytz, tzdata, six, numpy, python-dateutil,
pandas
Successfully installed numpy-2.3.0 pandas-2.3.0 python-dateutil-2.9.0.post0
pytz-2025.2 six-1.17.0 tzdata-2025.2
```

Adətən nöqtə ilə başlayan fayl və qovluq adları gizli hesab olunur, bunu adətən proqram qovluqlarını gözdən uzaq etmək üçün edirlər. Bizim virtual mühitin adı - `.venv` da nöqtə ilə başladığı üçün, bu virtual mühot üçün yaradılmış qovluq da gizli qovluq olacaq. Odur ki, gizli fayl və qovluqları da görmək üçün `ls` yox, `la` istifadə etdik. Mühiti aktişləşdirdikdən sonra `which python` əmri ilə əmin olduq ki, biz `python` adına müraciyət etdikdə, həqiqətən də virtual mühitimizdəki python binar faylı işə düşəcək. Daha sonra, `pip install -r requirements.txt` əmri ilə `requirements.txt` faylındakı bütün paket-versiyaları yükləyib quraşdırırıq. Öncə, biz `install` əmrindən `pip install <paketin adı>` kimi istifadə edirdik, ancaq `-r` əlavə etsək, tək paket adı əvəzinə - özündə bütün tələbləri(quraşdırılmalı olan paktləri) saxlayan mətn faylının yolunu göstərə bilərik. Əmri icra etdikdən sonra, fayldakı bütün paketlər quraşdırılır. Proyektin paketlərini `requirements.txt` faylında saxlamaq yaxşı vərdişdir - işdir, layihəni digər tərtibatçılara ötürsəniz, bunu virtual mühitsiz edəcəksiniz, çünki virtual mühit ancaq yaradıldığı qovluqda işləyəcək, başqa qovluğa və ya kompüterə köçürüb istifadə etmək alınmayacaq. `requirements.txt` faylının köməyiylə isə, digər tərtibatçı rahatlıqla bütün tələb olunan paketləri bir əmrə quraşdıra bilər.

> **Tip**
>
> Virtual mühitlər yaradıldıqları maşının ƏS-dən asılı olur. Odur ki, bir maşından digərinə köçürmək, hətta bir qovluqdan digər qovluğa köçürməyin mənası yoxdur, çünki mühitin konfiqurasiya fayllarında konkret yollar qeyd olunur və biz mühiti bir qovluqdan digərinə köçürdükdə bu yolları "sındırmış" oluruq. Odur ki, çalışın `requirements.txt` faylını aktual saxlayın. Daha bir məsələ - virtuai mühitin adlandırılması ilə bağıldır. Adətən mühiti `.venv` adlandırırlar, çünki yaradılacaq qovluq proqram qovluğu olacağı üçün, onu adi istifadəçinin gözündən gizlətməyə çalışırlar. Bu istifadəçinin təsadüfən oraya daxil olub, nəyisə silmə

ehtimalını azaltmaq üçün edilir. Ümumilikdə isə, bu bir konvensiyadır - virtual mühiti - `.venv`, tələblər faylını isə `requirements.txt` adlandırırlar.

Epiloq

Kitabı başlayıb, bura qədər gəlmisənsə - səni təbrik edirəm! Tam könül rahatlığı ilə - "Mən Pythonu bilirəm!" deyə bilərsən! Bu səhifələr boyu səninlə əslində uzun bir yol keçdik və son olaraq yenə bir neçə məsləhət vermək istəyirəm.

- **Səmt seçimi**: kitab boyu Python ilə proqramlaşdırmağa çalışırdıq. İndi isə, bu ümumi bilikləri tətbiq etmək üçün sahə seçmək lazımdır. Süni intellekt, Veb proqramlaşdırma, Proqram mühəndisliyi, Kiber təhlükəsizlik və sairə. Seçəcəyin sahədən asılı olaraq, həmin sahənin domen biliklərini öyrənəcəksən, daha sonra isə, həmin sahədə istifadə olunan Python kitabxanalarını. Məsələn, Data Science üçün `pandas`, `numpy`, `scikit-learn` kimi kitabxanalar istifadə olunur. Domen sahəni və proqramlaşdırma bacarığı olduqdan sonra kitabxanadan istifadə etmək, öz proqram məntiqini qurmaq - kifayət qədər asandır. Bu o deməkdir ki, nə qədər hazır kitabxanalardan istifadə etsən belə, proqramlaşdırma biliyi və təcrübəsi olmadan onlardan "səmərəli", əzbərsiz istifadə etmək çətin olacaq.
- **Proqramlaşdırmanın gündəlik həyata tətbiqi:** proqramlaşdırmaq - məsələni, problemi həll etmək üçün qurulmuş dəqiq alqoritmin reallaşdırılmasıdır. Özünüz də artıq bilirsiniz ki, ən çətini kodu yazmaq deyil, yazacağın kodu düşünüb tapmaqdır. Bunun üçün isə, məsələ üzərində soyuq, məntiqlə düşünüb - ən effektiv, problemi həll edəcək alqoritmi düşünüb koda çevirmək lazımdır. Bu cür yanaşma gündəlik həyatımıza da rahat şəkildə tətbiq oluna bilir. Harda və necə yaşamağımızdan asılı olmadan, hamımızın həyatında müəyyən çətinliklər - həll etməli olsuğumuz problemlər olur. Əgər bu problemlərə də "həll edilməli alqoritmik məsələ" kimi baxsaq, soyuq və məntiqli şəkildə həll edə bilərik. Odur ki, nə qədər "çətin məsələ" olsa belə, alqoritmi tapmaq mümkündür, hər şey sadəcə alqoritmin zaman - time complexity(O) xüsusiyyətindən və bizim iradəmizdən asılıdır.
- **Ümumi olaraq sonra nə öyrənim**: ümumi Python olaraq öyrənə biləcəyiniz neçə-neçə kitabxana və alətlər var. Onların hamısını bu kitabda hallandırıb əbəs yerə səni çaşdırmamaq üçün, yalnızca burada sadalayacam. Ola bilsin ki artıq onların bir çoxu barədə bizim Youtube kanalımızda məlumat tapa bilərsən:
  - `re` - Pythonda regex - mütəmadi ifadələr vasitəsilə şablona görə sürətli mətn axtarışı edir.
  - `pandas` - cədvəl faylları(xlsx, csv faylları və sairə) ilə işləmək üçün istifadə olunur. Orta ölçülü cədvəllərlə(bir neçə milyon sətr) sürətli şəkildə işləyə bilir. Artıq bildiyiniz kimi - üçüncü tərəf kitabxanasıdır.
  - `pydantic` - Pythonda dataklasslar, type hinting kimi fəndlərdən istifadə edərək, verilənləri validasiya etməyə kömək edir.
  - `unittest` və `pytest` Pythonda test yazmaq üçün istifadə olunan kitabxanalardır.
  - `logger` (standart kitabxana) və `loguru` (üçüncü tərəf kitabxanası) - loq aparmaq, proqramda baş verənləri qeyd etmək üçün kitabxanalardır.
  - SQL - verilənlər bazası ilə işləmək, sorğular yazmaq üçün dildir.
  - Git - üzərində işlədiyiniz proyekti
  - `sqlalchemy` - verilənlər bazaları ilə SQL yox, Python kodu ilə işləmək üçün istifadə olunan kitabxanadır.
  - `streamlit` - qısa müddətdə sadə veb tətbiq yaratmaq üçün kitabxanadır.
  - Veb Freymvörklər(Web Frameworks) - veb tətbiqləri yaratmaq üçün istifadə olunurlar. Freymvörk kitabxanaya bənzəyir, ancaq ondan fərqli olaraq özündən bütöv-ardıcıl və böyük

bir mexanizm ehtiva edir. Veb Freymvörklərə ən aktual misal kimi - `FastAPI` və `Django` göstərmək olar.

Son olaraq: Sahədən asılı olmayaraq, ən yaxşılardan olmaq istəyirsənsə, davamlı olaraq özünü inkişaf etdirməlisən - yeni alətlər öyrənmək, mövcudları daha yaxşı öyrənmək, bu sahədə baş verən xəbərləri izləmək - bunların hamısı şərtdir. Hər şeyi birdən-birə etmək mümkün deyil. Hər şey üçün zaman və əzm tələb olunur. Sən də bura qədər gəlmisənsə, afərin sənə! Deməli, sən bir əjdahasan! Necə deyərlər - sükanı belə saxla, sənə bol-bol uğurlar!

Bəzi Faydalı linklər:

- Pythonda hansı obyektlər yaddaşda nə qədər yer tutur, hesablama sürəti nə qədər olur:
  - https://mkennedy.codes/posts/python-numbers-every-programmer-should-know/