ZÁKLADNÍ KONSTRUKCE JAZYKA PYTHON

PYTHON OBJEKTOVĚ

KOLEKCE

DATOVÉ TYPY V PYTHONU

[bool] - Boolean (booleovský typ) nabývá buď hodnoty **True** nebo **False**.

[int], [float] - Čísla mohou být celá (**int**eger; 1 a 2), reálná (**float**; 1.1 a 1.2), zlomky (fraction; 1/2 and 2/3), nebo dokonce čísla komplexní.

[str] - Řetězce jsou posloupnosti Unicode znaků. Tuto podobu může mít například html dokument.

Bajty a pole bajtů, například soubor s obrázkem ve formátu jpeg.

Seznamy jsou uspořádané posloupnosti hodnot.

N-tice jsou uspořádané, neměnné posloupnosti hodnot.

Množiny jsou neuspořádané kolekce hodnot.

Slovníky jsou neuspořádané kolekce dvojic klíč-hodnota.

METODY TEXTOVÝCH ŘETĚZCŮ

(pokud funkce náleží nějakému objektu – datovému typu - nazýváme ji terminologicky metodou)

Zjištění, zda určitý řetězec obsahuje určitý podřetězec:
(když chceme zjistit, zda řetězec obsahuje (kdekoliv) nějaký podřetězec, použijeme k tomu operátor **in**)

operátor [***in***] – zjišťuje v objektu se nachází hledaný objekt
str.***startswith(****'hledaný_text'****)*** – zjišťuje, zda objekt začíná řetězcem v závorce
str.***endswith(****'hledaný_text'****)*** – zjišťuje, zda objekt končí řetězcem v závorce

Rozlišování mezi malými a velkými písmeny:
(v Pythonu záleží, zda je písmeno zapsané jako malé, nebo velké, takže když budeme chtít porovnávat významově nějaké dva řetězce, u kterých si nejsme jisti velikosti písmen, je dobré je před porovnáním převést na stejnou velikost)

str.***upper()*** – změní všechny znaky na velké

str.***lower()*** – změní všechny znaky na malé

str.***capitalize()*** – změní první písmeno ve větě na velké

str.***title()*** – změní první písmeno u každého slova na velké

Odstranění „bílých znaků“
(bílé znaky jsou veškeré mezery před a za textem a všechny netisknutelné znaky, formování textu)

str.***strip()*** – odstranění všech bílých znaků v řetězci

str.***lstrip()*** - odstranění všech bílých znaků na začátku řetězce

str.***rstrip()*** - odstranění všech bílých znaků na konci řetězce

Nahrazení části řetězce jiným textem
(řetězce jsou neměnné a tak, je-li řetězec měněn, vrací se výsledek v podobě nového řetězce)

replace() – metodě zadáváme dva parametry – část kterou chceme najít a nahradit, a část kterou má být nahrazena, a dále také můžeme určit, nejvyšší počet nahrazení, která mají být provedena, pokud metoda nenalezne hledaný podřetězec, vrátí celý původní řetězec

Formátovací operátor
(umožňuje použít zástupných znaků k vložení více proměnných na různá místa v řetězci)

[***%***] – zapisuje se v řetězci před znaky, které reprezentují určité proměnné, a které chceme nahradit:
%d – celá čísla (int)
%s – řetězce (str)
%f – desetiná čísla (float)

příklad:
a, b, c = 10, 20, a+b
s = "Když sečteme %d a %d, dostaneme %d" % (a, b, c)
>>> Když sečteme 10 a 20, dostaneme 30
(TYP: když potřebujeme v řetězci zapsat znak % můžeme použít zástupného znaku zpětného lomítka s procentem (\ %), nebo zapsat znak % dvakrát za sebou)

Vkládání „bílých znaků“
(hodí se zejména při formátování vzhledu v textu – například seznamu, nebo tabulky, kde chceme, aby všechen text byl stejně dlouhý, či stejně odsazen)

str.***ljust(****číslo****)*** – dle udaného počtu, přidá chybějící znaky, jako mezery, za text

str.***rjust(****číslo****)*** – dle udaného počtu, přidá chybějící znaky, jako mezery, před text

str.***center(****číslo****)*** – dle udaného počtu, přidá chybějící znaky, jako mezery, z obou stran textu

Zjištění počtu znaků v řetězci
(jedná se o globální funkci vracející číslo ***int***)

len(*str****)*** – funkce vrací délku celého řetězce

Metody is*()
(Metody is*() jsou metody řetězců, které vrací *True* nebo *False* v závislosti na tom, zda řetězec splňuje určitou podmínku, a využívá se k ošetřování uživatelských vstupů)

*str**.isalnum()*** - vrací, zda řetězec obsahuje pouze alfanumerické znaky (u prázdných řetězců vrací *False*)

*str**.isalpha()*** - vrací, zda řetězec obsahuje pouze znaky abecedy

*str**.isdecimal()*** - vrací, zda řetězec obsahuje pouze decimální znaky

*str**.isdigit()*** - vrací, zda řetězec obsahuje pouze číslice

*str**.islower()*** - vrací, zda je řetězec celý napsaný malými písmeny

*str**.isupper()*** - vrací, zda je řetězec celý napsaný velkými písmeny

*str**.isnumeric()*** - vrací, zda řetězec obsahuje pouze numerické znaky

*str**.isspace()*** - vrací, zda řetězec obsahuje pouze bílé znaky (mezery, tabulátory atd...)

*str**.istitle()*** - vrací, zda řetězec je titulek (každé první písmeno je velké)

TEXTOVÉ ŘETĚZCE
(textové řetězce se vytvářejí pomocí jednoduchých, nebo dvojitých apostrofů a slouží k uchování textu, pokud budeme chtít řetězec posuzovat z pohledu logiky, tak prázdný řetězec představuje hodnotu *False* a neprázdný *True*, řetězce se skládají z jednotlivých znaků a jsou tedy iterovatelné)

Speciální znaky
(zapisují pomocí takzvané escepe sekvencí, které začínají zpětným lomítkem)

n – nový řádek

t – tabulátor (zarovnává slova, aby každé začalo na pozici, která je násobkem 8)

a – zvonek (hodí se, pokud chceme v nějaké konzolové aplikaci zapípat)

**** - zpětné lomítko

r'str' – zapíšeme-li před textový řetězec r, řetězec bere znak po znaku a zpětná lomítka jsou brány pouze jako zpětná lomítka, a za nimi je jen další znak

\' \" – uvozovky (dále je možné použít v řetězci opačné uvozovky, než používáme na označení řetězce)

''' – trojité uvozovky – používají se pro zápis víceřádkových řetězců (zde samostatné uvozovky jdou)

Operace s řetězci

len(**str**) – délka řetězce (počet znaků včetně prázdných míst)

*str **+** str* - slučování řetězců do jednoho

*str ***** číslo* - násobení řetězce

ASCII
(zastaralé ukládání znaků, nahrazeno UTF-8, a jediná výhoda ASCII je, že základní znaky jsou uloženy za sebou podle abecedy, takže se dá použít k dohledávání sousedících znaků anglické abecedy – např. pro Césarovu šifru)

ord(**str**) – vrátí hodnotu znaku v ASCII

chr(**číslo**) – vrátí znak dle hodnoty v ASCII

Porovnání řetězců:

*str **==** str* - rovnost

*str **>** str* - je větší

*str **<** str* - je menší

*str **>=** str* - je větší nebo rovno

*str **<=** str* - je menší nebo rovno

*str **!=** str* – nerovná se

***!**str* – obecná negace

Řezání řetězců:
(řetězce jsou seznamy znaků, proto k vybírání znaků používáme hranaté závorky a indexu)

*str**[index]*** - vybere jediný znak

*str**[idx_začátku : idx_konce]*** - vybere znaky v rozsahu

*str**[idx_začátku : idx_konce : krok]*** - z rosahu vybere znaky dle uvedeného kroku

*str**[:]*** - vybere celý řetězec

*str**[::-1]*** - převrátí řetězec

Metody:

*str**.count(str)*** – vrátí počet podřetězců v jiném řetězci

*str**.find(str)*** – vrátí index prvního pozice podřetězce, pokud není nalezen, vrací -1

*str**.index(str)*** – stejná jako *find*, akorát pokud nenalezne hodnotu vyvolá výjimku

*str**.isalpha()*** – porovná zda jsou všechny znaky řetězce písemné znaky a vrátí *True/Folse*

*str**.isdigit()*** - porovná zda jsou všechny znaky řetězce číslice a vrátí *True/Folse*

*str**.islower()*** - porovná zda jsou všechny znaky malé písmena a vrátí *True/Folse*

*str**.isupper()*** - porovná zda jsou všechny znaky velká písmena a vrátí *True/Folse*

*str**.lower()*** – převede text na malá písmena

*str**.upper()*** – převede text na velká písmena

*str**.replace(str_co, str_čím)*** – nahradí v řetězci podřetězec jiným podřetězcem a vrátí výsledek jako nový řetězec

*str**.startswitch(str)*** - porovná zda řetězec začíná podřetězce a vrátí *True/Folse*

*str**.endswitch(str)*** – porovná zda řetězec končí podřetězce a vrátí *True/Folse*

Formátování řetězců:
(umožňuje nám do řetězců vkládat proměnné pomocí zástupných znaků, docílíme tak přehlednějšího kódu, než kdybychom řetězce nastavovali)

*"srt **{0}** str**".format(**xxx**)*** – do kudrnatých závorek píšeme číslo indexu, dle kterého jsou pak vybýrané hodnoty, z kulatých závorek za slovem *format*

Rozdělování a spojování řetězců CSV souborů:
(SCV = Comma-Separated Values = čárkou oddělené hodnoty, je soubor kde jednotlivé části textu - hodnoty, jsou v textu oddělené čárkou)

*str**.split('separátor')*** – rozdělí původní řetěze podle separátorů na sekvenci podřetězců, které vrátí, defaultním separátorem je mezera

*str**.join('separátor')*** – spojí sekvenci podřetězců do jednoho řetězce, hodnoty odděluje separátorem

SEZNAMY
(můžeme si ho představit, jako řadu indexy očíslovaných přihrádek, do kterých můžeme vkládat a vybírat obsah, seznamy mohou obsahovat položky různých typů)

Deklarace seznamu
(seznam deklarujeme pomocí hranatých závorek, nebo metodou *list()*)

list() – vytvoří seznam a vloží do něj objekt, je-li objekt sekvenční, rozebere objekt na jednotlivé sekvence a každou z nich postupně uloží do polí seznamu

Vybírání části seznamu
(ze seznamu položky vybíráme skrze indexy, potřebujeme-li ale vybrat více polí, využíváme zápis výběru pomocí indexů, kroku a dvojteček)

Možnosti:
*seznam**[idx]*** – vybere určitou položku dle čísla indexu (1. položka má index 0)

*seznam**[:]*** – vybere celý seznam

*seznam**[idx_od :]*** – vybere část seznamu, od uvedeného indexu, až do konce

*seznam**[: idx_do]*** – vybere část seznamu, od začátku, až do uvedeného indexu (počet je idx_do -1)

*seznam**[idx_od : idx_do]*** – vybere část seznamu, od uvedeného indexu, až do uvedeného indexu

*seznam**[: : krok]*** – vybere z celého seznamu pouze položky s uvedeným krokem

*seznam**[idx_od : : krok]*** – varianta výběru od uvedeného indexu s krokem

*seznam**[: idx_do : krok]*** – varianta výběru do uvedeného indexu s krokem

*seznam**[idx_od : idx_do : krok]*** – varianta výměru části s krokem

Metody pro práci se seznamy:

*seznam**.append(**hodnota**)*** – vloží položku do nového pole na konec seznamu

*seznam**.pop()*** – vrací poslední položku ze seznamu a zároveň ji maže

*seznam**.insert(**index, hodnota**)*** – vloží položku na danou pozici

*seznam**.extend(**hodnota**)*** – připojí všechny položky dané sekvence do seznamu

*seznam**.clear()*** – odebere všechny položky ze seznamu (stejné jako del(seznam[:]), nebo seznam = [])

*seznam**.remove(**hodnota**)*** – odebere danou položku ze seznamu, není-li nalezena, nastane výjimka

*seznam**.reverse()*** – otočí pořadí položek v seznamu

*seznam**.count(**hodnota**)*** – vrátí počet výskytů dané položky v seznamu

*seznam**.sort()*** – seznam seřadí, text podle abecedy, čísla podle hodnoty

*seznam**.index(**hodnota**)*** – vrací index prvního výskytu hledané položky, není-li nalezena, nastane výjimku, je také možné nastavit index počátku a konce oblasti hledání

Globální funkce pro práci se seznamy:

len(**seznam**) – vrací počet položek seznamu

min(**seznam**) – vrací položku s nejmenší hodnotou

max(**seznam**) – vrací položku s nejvyšší hodnotou

sum(**seznam**) –vrací součet všech položek seznamu (pouze u číselných typů)

sorted(**seznam**) – vrací setříděnou kopii seznamu, původní seznam je nedotčen

all(**seznam**) - funkce vrací True pokud jsou všechny položky seznamu vyhodnoceny jako True

(nenulová čísla, neprázdné řetězce i seznamy atd.)

any(**seznam**) - funkce vrací True, pokud je v seznamu alespoň jedna hodnota vyhodnocena jako True (nenulová čísla, neprázdné řetězce i seznamy atd.)

del(**seznam[idx]**) – maže položky seznamu

PODMÍNKY A VĚTVENÍ V PYTHONU

(větvení programu nastává ve chvíli, kdy program se v určitém bodě reaguje na určité situace – například vstup od uživatele, a k větvení dochází na základě splnění, či nesplnění podmínek)

Podmínky

(podmínku vytvoříme pomocí klíčového slova *if*, za kterým následuje nějaký logický výraz ukončený dvojtečkou, pokud je výraz v podmínce pravdivý (*True*), provede se blok programu určený touto podmínkou, pokud výraz pravdivý není, tj. je nepravdivý (*False*), celý blok podmínky se vynechá a program pokračuje dále)

Klíčová slova:

if – základní (první) podmínka

elif – pokud první podmínka nebyla splněna, můžeme skrze tento příkaz, zadat další podmínky

else – tímto příkazem určujeme, co se má udělat v případě, když nejsou podmínky splněny

Operátoři podmínek:

== - rovnost

> - je větší

< - je menší

>= - je větší nebo rovno

<= - je menší nebo rovno

!= - nerovná se

not – obecná negace

Složené podmínky pro sloučení více podmínek:

and – musí platit podmínky před i podmínka za

or – musí platit alespoň jedna z podmínek

Ternární výraz (operátor)
(konstrukt, který umožňuje vyhodnocení podmínky a vrácení jednoho ze dvou možných výrazů, termální výraz je odvozeno od toho, že má 3 části, ternární výraz je zpravidla určený pro výběr hodnoty, nepoužívá se k rozlišení volání metod bez návratového typu)

Zápis:
'hodnota_True' if (výraz_podmínky) else 'hodnota_False'
(pokud se výraz podmínky vyhodnotí jako True, bude použita hodnota před ním, a pokud jako False, bude použita hodnota za ním, výraz může, ale i nemusí být v závorkách – ty jsou zde hlavně pro větší přehlednost)

Vnořování:
(termální výraz lze zanořovat dovnitř sebe a tím reagovat na tři a více hodnot, nicméně zanoření ve většině případů kód znepřehlední, a tak je dobré k vnořování přistupovat s rozvahou)

Příklad:
pohlavi = "nevim" # nějaká proměnná udávající pohlaví
nazevPohlavi = ***"muž" if (pohlavi == "muz") else "žena" if (pohlavi == "zena") else "nezname"***
print(nazevPohlavi)

Konstrukce match - case:
(funguje od verze 3.10, a slouží pro výběr volby a následného příkazu, nahrazuje tak větvení *if – elif - else*, a zároveň tímto větvením může sám být nahrazen)

match(*volba****):*** – v závorce je vstupní hodnota, která se porovnává s hodnotami case (jako *if*)

case *„možný výsledek“****:*** – je varianta předpokládaného výsledku,
za kterou se do bloku píše úkon, který se má při shodě provést (jako *elif*)

case _: - odkazuje na úkon, který se má provést, když nedošlo k shodě (jako *else*)

Match s propadáváním
(používáme, když chceme, aby jedna *case* dokázala obsloužit více stavů naráz, je to standardní součást gramatiky Pythonu a v zdrojových kódech se běžně vyskytuje)

Konstrukt:
(pokud potřebujeme ve více blocích *case* provádět stejný kód, stačí jejich hodnoty vložit za sebe a oddělit je svislou čárou |)

Příklad:
```
case 1 | 2 | 3:
    print("Je první čtvrtletí.")
```

CYKLY
(slouží pro opakování určitého úkonu, či sledu úkonů)

Cyklus WHILE
(nejednoduší cyklus – opakuje své úkony, dokud podmínka je pravdivá – *True*, může i obsahovat nepovinou větev *else*, pro případ kdy podmínka již není *True*.)

Příklad:
```
i = 1
while (i <= 10):
        print(i, end = ' ')
        i += 1
>>> 1 2 3 4 5 6 7 8 9 10
```

Nekonečný cyklus while True:
(používá se ve chvíli, kdy předem nevíme daný počet opakování, jako podmínky bude jakýkoliv logický výraz, který se dá vyhodnotit jako *True*, nebo samotný příkaz *True*, cyklus v tu chvíli bude běžet, dokud bude podmínka platná, nebo dokud se nezmění příkaz *True* na *False*, k tomu se používají buď mechanizmy mimo cyklus, a nebo je podmínka ošetřena a hlídána uvnitř cyklu, všeobecně se ale doporučuje nekonečným cyklům se vyhýbat)

Cyklus FOR-IN
(tento cyklus obsahuje vždy pevný počet opakování, který se rovná počtu prvků v sekvenci, cyklus projde všechny prvky v sekvenci, prvek zkopíruje do proměnné cyklu a provede s ním příkazy cyklu)

Příklad:
```
slovo = "ahoj"
for znak in slovo:
        print(znak, end=' ')
>>> a h o j
```

Funkce range()
(funkce vrací vygenerovaná čísla nějaké sekvence, např ve formě rozsahu)

Funkci je možné zapsat s jedním až třemi parametry:

range(*n****)*** - vrátí čísla od nuly do n-1 (do n, které už zahrnuto není)

range(*m, n****)*** - vrátí čísla od m do n-1

range(*m, n, i****)*** - vrátí čísla od m a každé další i-té číslo do n-1

Parametry:

n ***- konec*** – určuje na kterém indexu se končí (tento index už ale není zahrnutý)

m - ***začátek*** – určuje na kterém indexu se začíná (výchozí hodnota je 0)

i - ***krok*** – specifikuje hodnotu kroku procházení (výchozí hodnota je 1)

Vnořování cyklů
(pokud máme určitý cyklus hodnot a každou hodnotou cyklu, použijeme v dalším cyklu, mluvíme o vnořování cyklů, což je vhodné zejména pro tabulky)

Příklad:

```
print("Tabulka malé násobilky:")
for j in range(1, 11):
    for i in range(1, 11):
        x = str(i * j)
        print(x.rjust(3), end = " ")
    print()
```

Přeskočení a přerušení cyklu:

pass
(je tzv. *no-op* /no operation/ výraz, který se používá k označení prázdného bloku kódu, sám nemá žádnou funkcionalitu a používáme ho převážně v případech, kdy potřebujeme tyto části kódu z nějakého důvodu doplnit později)

break
(příkaz ukončuje aktuální cyklus a používá se nejčastěji v situaci, kdy pomocí cyklu nalezneme nějakou položku a dále již v procházení nechceme pokračovat, v praxi je ale často nahrazován příkazem *return*, který také přeruší cyklus, ale navíc vrací poslední hodnotu)

continue
(příkaz je podobný příkazu *break*, ale používá se k ukončení aktuálního průběhu cyklu, a ne celého cyklu – cyklus po té přechází na další iteraci tento příkaz často použijeme při validování položek při procházení nějaké kolekce)

return
(příkaz podobně jako *break* ukončuje cyklus, ale vrací poslední hodnotu)

FUNKCE
(je programovací paradigma – způsob jak něco naprogramovat – kdy program si rozdělíme na menší části – podproblémy – které řešíme samostatně funkcemi, funkce obvykle přijímá argumenty – data, která zpracuje – a něco vrací – např. výslednou hodnotu – ale také nemusí vracet nic a jen vykoná daný úkon)

Základní synaxe:
(funkce mají podobnou synaxi jako větvící příkazy *if*, *elif*, *else*, definujeme ji pomoci klíčového slova *def*, a výslednou hodnotu vracíme příkazem *return*, za ***def*** se píše ***mezera*** a po té ***název funkce***, za který se píší ***jednoduché závorky***, do kterých se píší ***názvy jednotlivých argumentů***, pokud je funkce má, a za konec závorky se píše dvojtečka, která znamená, že tělo funkce bude odskočeno o jeden tabulátor)

Argumenty:
(v Pythonu máme 2 druhy argumentů:
Poziční argumenty - jsou argumenty, které mají svoji pozici uvnitř závorek a na jejich místo se dosadí hodnoty
Klíčové argumenty - mají již předem nastavenou hodnotu, která se dá změnit, a jsou označovány klíčovým slovem s rovnítkem, tyto argumenty není třeba zadávat, pokud nechceme měnit přednastavenou hodnotu)

Operátor *
(je-li použit před názvem argumentu při deklaraci funkci, způsobí, že funkce bude očekávat iterovatelnou hodnotu uloženou jako n-tici, použijeme-li ale hvězdičku samostatně, znamená to, že nechceme aby bylo možné zadávat argumenty pozičně a je třeba u všech argumentů uvést i jejich název, a nakonec, jsou-li uvedeny jako argument dvě hvězdičky, znamenají násobení)

Rekurze
(rekurze je zápis kódu, kdy v určitém místě, funkce volá sama sebe a vytváří tak cyklus, při rekurzi si musíme dát pozor na to, aby se funkce někdy ukončila, jinak program spadne na přetečení zásobníku)

print()
(i pro funkci print můžeme kromě textu, který se má vytisknout uvádět argumenty:
sep – udává mezery mezi jednotlivými prvky /výchozí = ‘’/
end – definuje, čím se zápis ukončí / výchozí = ‘\n‘ – nový řádek)

OŠETŘENÍ CHYB TRY-EXCEPT

(používá se k zachycení možné chyby, tím že příkaz se nejprve provede a zkontroluje se zda prošel bez toho, aby vyvolal nějakou chybu – výjimku, pokud projde, posílá se hodnota dále, pokud neprojde, zkontroluje se, zda vyvolaná výjimka je ošetřena v některé z větví except, a pokud zde nenajde pro danou chybu řešení, pokusí se hledat řešení o úroveň výše , atd, a pokud nikde není zachycena, dojde k vyvolání výjimky)

Zápis:

try:
  #blok příkazů
except jmeno_prvni_vyjimky:
  #blok příkazů
except jmeno_dalsi_vyjimky ***as*** chyba::
  #blok příkazů
  #text vyjímky se uloží do proměnné chyba
#zde je buď konec, nebo zachycení dalších výjimek

Nejčastější výjimky:

SyntaxError - chyba je ve zdrojovém kódu
ZeroDivisionError - pokus o dělení nulou
TypeError - nesprávné použití datových typů - např. sčítání řetězce a čísla
ValueError - nesprávná hodnota

Příklad na zachycení výjimky typu u vstupu:

```
def nacti_cislo(text_zadani, text_chyba):
  spatne = True
  while spatne:
    try:
      cislo = float(input(text_zadani))
      spatne = False
    except ValueError:
      print(text_chyba)
    else:
      return číslo
```

Příklad na zachycení výjimky u zadání ano/ne:

```
def dalsi_priklad():
  nezadano = True
  while nezadano:
    odpoved = input("\nPřejete si zadat další příklad? y / n: ")
    if (odpoved == "y" or odpoved == "Y"):
      return True
    elif (odpoved == "n" or odpoved == "N"):
      return False
    else:
      pass
```

KNIHOVNY

(knihovny, nebo-li moduly nám poskytují užitečný zdroj datových typů, funkcí a různých nástrojů, díky nim, nemusíme psát už to, co někdo napsal před námi, programi pak tvoříme za pomoci existujících modulů, a díky tomu je jejich vývoj pohodlnější a rychlejší)

Import knihovny:

abychom mohli funkce z knihovny použít, musíme ji nejprve naimportovat – zpřístupnit:
import nazev_modulu

a po té voláme funkci modulu, stejně, jako by to byla metoda modulu:
nazev_modulu.nazev_funkce()

ne vždy je praktické importovat celou knihovnu funkcí, a tak když chceme importovat jen určitou funkci, děláme to za pomocí příkazu *from*:
from nazev_modulu import nazev_funkce

v tu chvíli se z funkce modzulu stane naše funkce a pro její volání stačí napsat jen:
nazev_funkce()

pokuď bychom chtěli z modulu zpřístupnit vše, použijeme hvězdičky:
from nazev_modulu import *
tento způsob se ale nedoporučuje, protože nám tak velmi nabobtná jmenný systém a může také dojít ke kolizi názvů

pokud používáme importu z balíčků seskupení modulů, můžeme použít pro konkrétní modul vlastní název:
import nazev_modulu as vlastni_nazev

Knihovna math

(je modul pro rozšířené matematické funkce a konstanty)

Některé příklady:
math.pi() – pí

math.e() – Eulerovo číslo
math.sin() – sinus uváděný v radiánech (na stupně je potřeba je dodatečně převést)
math.cos() – cosinus uváděný v radiánech (na stupně je potřeba je dodatečně převést)
math.tan() – tangen uváděný v radiánech (na stupně je potřeba je dodatečně převést)
math.asin() – opačná funkce k sin() (vrátí uhle v radiánech pro danou hodnotu *sin*)
math.acos() – opačná funkce k cos() (vrátí uhle v radiánech pro danou hodnotu *cos*)
math.atan() – opačná funkce pro tan() – (vrátí uhle v radiánech pro danou hodnotu *tan*)
math.degrees() – pomocí této funkce převedeme hodnotu úhlu z radiánů na stupně
math.ceil() – funkce provede zaokrouhlení na nejbližší celé číslo směrem nahoru
math.floor() - funkce provede zaokrouhlení na nejbližší celé číslo směrem dolu
math.fabs() – funkce nám vrací absolutní hodnotu desetinného čísla
(vestavěná funkce abs() nám vrací absolutní hodnotu celého čísla)
math.factorial() – vrací faktoriál daného čísla
math.pow() – funkce umocní první předané číslo na druhé předané číslo (stejné jako X**Y)
math.sqrt() – vrací druhou mocninu daného čísla (stejné jako X**2)
math.hypot() – funkce vrací eukleidovskou vzdálenost tj. math.sqrt(x*x + y*y)
(funkce vypočítá na základě pythagorovy věty přeponu pravoúhlého trojúhelníka)
math.log() – je-li zadán, funkce vrátí logaritmus čísla o základu, jinak vrátí hodnotu přirozeného logaritmu (se základem Eulerova čísla)
math.log10() – vrátí z daného čísla logaritmus o základu 10
math.log2() – vrátí z daného čísla logaritmus o základu 2

Knihovna random
(poskytuje mnoho užitečných funkcí, které nám umožňují generovat pseudo náhodná čísla, míchat seznamy, vybírat z nich náhodné prvky, atd)

random.randint() – vybere náhodné číslo v rozsahu od prvního do druhého čísla
random.choice() – vybere náhodný prvek ze seznamu, nebo jiné kolekce
random.suffle() – zamíchá pořadím prvků seznamu, nebo jiné kolekce
random.sample() – vybere daný počet náhodných prvků ze seznamu, nebo jiné kolekce

Easter egge
(jsou skryté a zábavné funkčnosti, které do jazyka schovali jeho tvůrci)

import this – báseň „The Zen of Python, by Tim Peters“
import antigravity – odkaz na stránky https://xkcd.com/353/
import __hello__ - ???
from __future__ import braces - ???

NEJČASTĚJŠÍ CHYBY PŘI PSANÍ KÓDU

Kdy program funguje správně?
- když funguje
- když dodržuje dobré praktiky
- když je otestovaný

Jak správně pojmenovávat proměnné:
- proměnné vždy pojmenováváme podle toho, co obsahují, a ne k čemu slouží
- nepoužívat zkratky
- všechny písmena by měla být malá, bez diakritiky a oddělená podtržítkem
- proměnné v jednom kódu pojmenováváme jedním jazykem (nemíchat angličtinu a češtinu)
- pro více variant nepoužíváme rozčíslování, ale podtržítkem a slovem rozšířený název
- seznamy (kolekce) by měli být pojmenované v množném čísle
- v podmínkách neduplikujeme bool hodnotu: *if ((vek >= 18) == True): // if (vek >= 18):*
- měli bychom se vyhýbat přebytečných podmínek na stavy, které jsou již jasné
- nikde v programu by se neměl kód opakovat DRY (do not repete yourself)
- namísto tabulátoru používat 4 mezer

Refaktoring – jsou následné úpravy kódu, které zachovají jeho funkčnost, ale zkvalitní návrh programu

PYTHON OBJEKTOVĚ

Evoluce metodik:

1) ***Strojový kód*** - je soubor instrukcí, kde není možnost pojmenovávat proměnné nebo zadávat matematické výrazy a je specifický pro daný hardware (procesor).

2) ***Nestrukturované paradigma*** - je podobné assemblerům, jedná se o soubor instrukcí, který se vykonává odshora dolů. Zdrojový kód již nebyl závislý na hardwaru a byl lépe čitelný pro člověka.

3) ***Strukturované programování*** - je první paradigma, které se udrželo delší dobu a opravdu chvíli postačovalo pro vývoj nových programů. Programujeme v něm pomocí cyklů a větvení.

4) ***Modulární programování*** - umožňuje zapouzdřit určitou funkcionalitu do modulů, stále však neexistuje způsob, jak již napsaný kód modifikovat a znovu využít.

5) **Objektově orientované programování** - Jedná se o filozofii a způsob myšlení, designu a implementace, kde klademe důraz na **znovupoužitelnost**. Poskládání programu z komponent je výhodnější a levnější. Můžeme mu věřit, je otestovaný (o komponentách se ví, že fungují, jsou otestovány a udržovány). Pokud je někde chyba, stačí ji opravit **na jednom místě.**

Jak OOP funguje:

V tomto paradigmatu se snažíme se nasimulovat realitu tak, jak ji jsme zvyklí vnímat. Můžeme tedy říci, že se odpoutáváme od toho, jak program vidí počítač (stroj) a píšeme program spíše z pohledu programátora (člověka). Jako jsme kdysi nahradili assembler lidsky čitelnými matematickými zápisy, nyní jdeme ještě dál a nahradíme i ty. Jde tedy o určitou úroveň abstrakce nad programem. To má značné výhody už jen v tom, že je to pro nás přirozenější a přehlednější.

Základní konstrukce:

Základní jednotkou je objekt, který odpovídá nějakému objektu z reálného světa.

Každý ***objekt*** má své ***atributy*** a ***metody***.

Atributy objektu jsou vlastnosti neboli data, která uchovává. Jedná se o prosté proměnné, se kterými jsme již stokrát pracovali. Někdy o nich hovoříme jako o vnitřním stavu objektu.

Metody jsou schopnostmi, které umí objekt vykonávat. Metody mohou mít parametry a mohou také vracet nějakou hodnotu.

Třída je vzor, podle kterého se objekty vytváří. Definuje jejich vlastnosti a schopnosti.

Instance je objekt, který se vytvoří podle třídy. Instance mají stejné rozhraní jako třída, podle které se vytváří, ale navzájem se liší svými daty (atributy).

OOP stojí na základních třech pilířích: ***Zapouzdření***, ***Dědičnost*** a ***Polymorfismus***.

Zapouzdření umožňuje skrýt některé metody a atributy tak, aby zůstaly použitelné jen pro třídu zevnitř. Nemůžeme tedy způsobit nějakou chybu, protože využíváme a vidíme jen to, co tvůrce třídy zpřístupnil. Rozhraní (interface) třídy rozdělí na veřejně přístupné a její vnitřní strukturu.

Vytvoření třídy a přidání metody:

```
class NazevTridy:
    def nazev_metody_tridy(self):
        print("Hello object world!")
```

Název třídy píšeme vždy velbloudí notací bez mezer a na rozdíl od proměnných má každé slovo v názvu velké první písmeno.)

Deklarace metody v Pythonu je podobná deklaraci funkce. Za klíčovým slovem `def` následuje samotný název metody. Metody píšeme stejně jako proměnné a funkce malými písmeny. V případě víceslovného názvu použijeme podtržítka. **Závorka s parametry je povinná.** První povinný poziční argument je `self`. Do něj se vloží "odkaz" na objekt, do kterého metoda náleží. Tento argument tam vloží sám objekt. Do těla metody zapíšeme kód pro výpis do konzole.

Založení instance třídy:

nazev_instance = *NazevTridy* ()

nazev_instance je objekt, se kterým budeme pracovat. Objekty se ukládají do proměnných, název třídy slouží jako datový typ. Instance má zpravidla název třídy, jen má první písmeno malé. Deklarujme si tedy proměnnou a následně v ní založme novou instanci třídy.

Při vytvoření nové instance se zavolá tzv. ***konstruktor***. To je speciální metoda na třídě, proto při vytvoření instance píšeme ty prázdné závorky, jelikož voláme tuto "vytvářecí" metodu. Konstruktor zpravidla obsahuje nějakou inicializaci vnitřního stavu instance (např. dosadí výchozí hodnoty do proměnných). Pokud v kódu žádný konstruktor nedeklarujem, Python si vytvořil tzv. *implicitní prázdný konstruktor*. Vytvoření instance objektu je tedy podobné volání metody.

Na instanci třídy pak můžeme zavolat libovolnou metodu třídy:
nazev_instance. ***nazev_metody_tridy***()

Přidání parametru
syntaxe parametru metody je stejná, jako syntaxe proměnné. Jednotlivé parametry oddělujeme čárkou.

Přidání atributu
Atributy se definují stejně jako proměnné. Před jejich název píšeme `self`.

Navracení hodnoty - RETURN
Vzhledem k objektovému návrhu není nejvhodnější, aby si každý objekt ovlivňoval vstup a výstup jak se mu zachce. **Každý objekt by měl mít určité kompetence, které by neměl překračovat**. Výhodou takto navrženého objektu je vysoká univerzálnost a znovupoužitelnost. Bude pak pouze na jeho příjemci, jak s ním naloží. Takto můžeme výstup ukládat do souborů, psát na webové stránky nebo dále zpracovávat. K návratu hodnoty použijeme příkaz `return`. ***Return*** metodu ukončí a navrátí její hodnotu. Jakýkoli další kód v těle metody se po `return` již neprovede!

Přidání komentářů
Jestliže do interaktivní konzole v IDLE zadáme příkaz `help()` a jako parametr napíšeme *název třídy*, zobrazí její popis i popis všech jejích metod. Tyto komentáře vytváříme tak, že je píšeme pod název třídy a pod název každého atributu a metody. K jejich zápisu použijeme tři dvojité uvozovky (`"""`).

Načítání třídy v jiném souboru
Po té co třídu zapíšeme, můžeme ji i uložit jako py soubor a v jiném py souboru si ji můžeme naimportovat pomocí formule:

from *nazev_souboru_s_tridou* ***import*** *NazevTridy*

Volitelně můžeme za NazevTridy dopsat: ***as*** *nazev_instance*
a tím si rovnou pojmenujeme i instanci, kterou pak dále v našem souboru budeme používat

Soukromé atributy
jsou atributy u kterých nechceme aby se dali zvenčí modifikovat***. Začínají jedním, nebo dvěma podtržítky. Jedním podtržítkem*** není přístup odepřen, ale dává najevo, že daný prvek se nemá z vnější používat. ***Dvě podtržítka*** způsobí, že k atributu nelze normálně přistupovat. Oba přístupy jsou používané. Při návrhu třídy tedy použijeme pro atributy podtržítka. Nepoužijeme je pouze v případě, že něco bude opravdu potřeba vystavit.

Konstruktor
je metoda, která se zavolá ve chvíli vytvoření instance objektu a slouží k nastavení vnitřního stavu objektu a k provedení případné inicializace. Při vytváření instance třídy, je název třídy konstruktor, který když v sobě neobsahuje pokyny, vygeneruje si sám prázdnou metodu.

__new__() a __init__()
jsou metody kterými můžeme deklarovat konstruktor. Metoda ***__new__()*** se volá při vytváření objektu. Většinou si ale vystačíme s metodou ***__init__(),*** která se volá při inicializaci objektu. Pokud chceme při inicializaci zavést nějaké atributy, zapisujeme je v moduly ***__init__()*** vždy se slovem ***self*** a ***tečkou.***

Zapouzdření
atributy, které nechceme aby někdo měnil je dobré nastavit jako neveřejné (soukromé), nicméně abychom se dostali k hodnotě atributu a mohli ji zobrazit, vytvoříme v třídě metodu, která bude vracet hodnotu atributu. Docílíme tím toho, že atribut bude read only.

Parametr konstruktoru
pokud při inicializaci chceme aby některé údaje specifikoval uživatel, přidáme je jako parametry do konstruktoru **__init__(self, parametr),** pokud použijeme za uvedením paramnetru rovnítko s určitou hodnotou, bude tato hodnota brána jako výchozí a použita i bez udání parametru – takovému parametru s předdefinovanou hodnotou se říká ***klíčový parametr***.

Import modulu uvnitř třídy
Budemeli používat uvnitř třídy nějaký externí modul (například random) naimportujeme si ho vnitřně, za použití podtržítka. Při importování modulů se Python podívá, jestli byl již modul importován. Pokud ano, Python ho znovu neimportuje.

Překrývání metody __str__
tuto metodu obsahuje každý objekt a je určena k tomu, aby vrátila textovou reprezentaci instance a hodí se tak ve všech případech ,kdy si instanci potřebujeme vypsat a nebo s ní pracovat jako s textem.
Tuto metodu mají např. i čísla, v Pythonu pak funguje **implicitní konverze**. Jakmile tedy budeme chtít do konzole vypsat číslo nebo kterýkoli jiný objekt, Python na něm zavolá metodu `__str__` a vypíše její výstup. Nikdy bychom si neměli dělat vlastní metodu, např. něco jako `vypis()`, když už máme v Pythonu připravenou cestu. ***Překrývání*** se tomuto procesu říká proto, že v původním __str__ objektu je přednastaven určitý úkon (cesta k naší třídě), nicméně, když metodu __str__ použijeme a zapíšeme do ní vlastní výstup, tak tímto ***překryjeme*** její původní nastavení a metoda bude odteď vykonávat to, co jsme zadali.

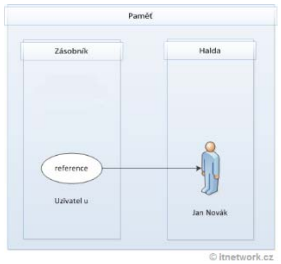
PAMĚŤ

Aplikace (resp. její vlákno) má operačním systémem přidělenou paměť v podobě tzv. zásobníku (stack). Jedná se o velmi rychlou paměť s přímým přístupem. Její velikost aplikace nemůže ovlivnit, protože prostředky jsou přidělovány operačním systémem.

Zásobník a halda

Zásobník i halda se nacházejí v paměti RAM. Rozdíl je v přístupu a velikosti. Halda je prakticky neomezená paměť, ke které je však přístup složitější a tím pádem pomalejší. Naopak zásobník je paměť rychlá, ale velikostně omezená.

Objekty jsou v paměti uloženy vlastně nadvakrát, jednou v zásobníku a jednou v haldě. V zásobníku je uložena pouze tzv. reference, tedy odkaz do haldy, kde se poté nalézá opravdový objekt.



Výhody tohoto systému:
• Místo ve stacku je omezené.
• Když budeme chtít použít objekt vícekrát (např. ho předat jako parametr do několika metod), nemusíme ho v programu předávat jako kopii. Předáme pouze referenci na objekt místo toho, abychom obecně paměťově náročný objekt kopírovali. Toto si za chvíli ukážeme.

Chování dat v paměti

Proměnná reference je uložena v zásobníku a adresou odkazuje na objekt uložený v haldě. Pokud vytvoříme druhou proměnou, která bude odkazovat na náš objekt, pak změnou dat v jedné proměnné měmíme data v objektu a tím pádem i v druhé proměné. Abychom tomuto předešli, pak při vytváření kopie je potřeba zvolit metodu, která vytvoří v haldě nový objekt.

Garbage collector a dynamická správa paměti

Paměť můžeme v programech alokovat ***staticky*** nebo v ***dinamické*** správě paměti.
V statické alokaci je předem potřeba určit, kolik paměti budeme používat, a o tuto paměť požádat operační systém. Ten ji vyhradí a předá nám na ni adresu. Na toto místo pak v programu odkazují tzv, pointery (neboli přímé ukazatele do paměti), přes které jsme s pamětí pracovali.

Problém byl, že nikdo nehlídal, co do paměti dáváme (ukazatel směřoval na začátek vyhrazeného prostoru). Když jsme tam dali něco většího, zkrátka se to stejně uložilo a přepsala se data za naším prostorem, která patřila třeba jinému programu nebo operačnímu systému a často jsme si v paměti přepsali nějaká další data našeho programu a program se začal chovat chaoticky.

Když naopak nějaký objekt přestaneme používat, musíme po něm místo sami uvolnit. Pokud to neuděláme, paměť zůstane blokovaná. Pokud toto děláme např. v nějaké metodě a zapomeneme paměť uvolňovat, naše aplikace začne padat, případně zasekne celý operační systém. Taková chyba se opět špatně hledá.

Za cenu mírného snížení výkonu vznikly **řízené jazyky** (managed) **s tzv. garbage collectorem, jedním z nich je i Python**.

C++ se samozřejmě nadále používá, ale pouze na specifické programy, např. části operačního systému nebo 3D enginy komerčních her, kde je potřeba z počítače dostat maximální výkon. Na 99 % všech ostatních aplikací se více hodí Python nebo jiný řízený jazyk. Mimo jiné právě kvůli automatické správě paměti.

Garbage collector

je vlastně **program, který běží paralelně s naší aplikací** v samostatném vlákně. Občas se aktivuje a podívá se, na které objekty již v paměti nevedou žádné reference. Ty potom odstraní. Protože je jazyk řízený a nepracujeme s přímými pointery, **není vůbec možné paměť nějak narušit**, nechat ji přetéct a podobně. Interpret se o paměť automaticky stará.

Hodnota None

Referenční typy mohou, na rozdíl od hodnotových, nabývat speciální hodnoty `None`. `None` je klíčové slovo a označuje, že **reference neukazuje na žádná data**.

Kopírování objektů

Jak tedy můžeme vytvořit opravdovou kopii objektu? Můžeme objekt znovu vytvořit pomocí konstruktoru a dát do něj stejná data. Druhou možností je použít hlubokou kopii.

Metoda __repr__()
medoda vrací řetězec, který můžeme poté předat funkci `eval()`. Tato funkce se používá **k dynamickému provádění kódu**. Nechávat uživatele zadat výraz není bezpečné a tak se používá metoda __repre__(), kde konstruktor sami definujeme:

```
def __repr__(self):
    return str(f"NázevTřídy({název_aktuální_instance})"
```

Kód pro vytvoření kopie:

```
kopie_instance = eval(repr(název_aktuální_instance))
```

Nebo můžeme použít modul copy. A z něj funkci deepcopy():

```
kopie_instance = copy.deepcopy(název_aktuální_instance)
```

ZAPOUZDŘENÍ, DĚDIČNOST, POLYMORFIZMUS

Zapouzdření
umožňuje skrýt některé metody a atributy tak, aby zůstaly použitelné jen pro třídu zevnitř. Nemůžeme tedy způsobit nějakou chybu, protože využíváme a vidíme jen to, co tvůrce třídy zpřístupnil. Rozhraní (interface) třídy rozdělí na veřejně přístupné a její vnitřní strukturu. Atributy, které nechceme aby někdo měnil je dobré nastavit jako neveřejné (soukromé), nicméně abychom se dostali k hodnotě atributu a mohli ji zobrazit, vytvoříme v třídě metodu, která bude vracet hodnotu atributu. Docílíme tím toho, že atribut bude read only.

Soukromé atributy
jsou atributy u kterých nechceme aby se dali zvenčí modifikovat**.**
Začínají jedním, nebo dvěma podtržítky.
Jedním podtržítkem není přístup odepřen, ale dává najevo, že daný prvek se nemá z vnější používat.
Dvě podtržítka způsobí, že k atributu nelze normálně přistupovat. Oba přístupy jsou používané. Při návrhu třídy tedy použijeme pro atributy podtržítka. Nepoužijeme je pouze v případě, že něco bude opravdu potřeba vystavit.

Dědičnost
Dědičnost je jedna ze základních vlastností OOP a slouží k tvoření nových datových struktur na základě starých. Když použijeme dědičnost, definujeme základní třídu tak, aby z ní odvozená třída dědila. Zděděné atributy a metody tedy již nemusíme znovu definovat, Python je sám do nové třídy dodá. Ke zdědění používáme závorky. Mezi závorky píšeme třídy, od kterých naše třída dědí. V anglické literatuře najdete dědičnost pod slovem inheritance.

V potomkovi nebudou přístupné privátní atributy, ale pouze veřejné atributy a metody. Soukromé atributy a metody jsou chápány jako speciální logika konkrétní třídy, která je potomkovi utajena, i když ji vlastně používá, nemůže ji měnit. Abychom dosáhli požadovaného výsledku, použijeme nový modifikátor přístupu a to (nečekaně) jedno podtržítko. V Pythonu se **atributy a metody s jedním podtržítkem** nazývají **vnitřní**. Pro ostatní programátory, nebo objekty to znamená: "Toto je sice zvenčí viditelné, ale, prosím, nehrabejte mi na to!"

Výhody dědění jsou jasné, nemusíme opisovat oběma třídám ty samé atributy, ale stačí dopsat jen to, v čem se liší. Zbytek se podědí. Přínos je obrovský, můžeme rozšiřovat existující komponenty o nové metody a tím je znovu využívat. Nemusíme psát spousty redundantního (duplikovaného) kódu. A hlavně - když změníme jediný atribut v mateřské třídě, automaticky se tato změna všude podědí. Nedojde tedy k tomu, že bychom to museli měnit ručně u 20ti tříd a někde na to zapomněli a způsobili chybu. Jsme lidé a chybovat budeme vždy, musíme tedy používat takové programátorské postupy, abychom měli možností chybovat co nejméně.

O mateřské třídě se někdy hovoří jako o předkovi (zde Uzivatel) a o třídě, která z ní dědí, jako o potomkovi (zde Administrator). Potomek může přidávat nové metody nebo si uzpůsobovat metody z mateřské třídy (viz dále). Můžete se setkat i s pojmy nadtřída a podtřída.

Testování typu třídy

Můžeme použít vestavěnou funkci ***type():***
type(a) == int
>>> True

Ale lepší je použít vestavěnou funkci ***isinstance()****:*
isinstance(a, int)
>>> True

Dále můžeme zjistit nadtřídu/nadtřídy objektu (třídy).
a.__class__.__base__
>>> <class 'object'>

Všechny objekty v Pythonu dědí ze třídy objekt. V Pythonu 2.X bylo nutné uvádět, že námi vytvořená třída dědí ze třídy objekt. Nyní si to Python "doplní" sám.

Jazyky, které dědičnost podporují, buď umí dědičnost jednoduchou, kde třída dědí jen z jedné třídy, nebo vícenásobnou, kde třída dědí hned z několika tříd najednou, mezi ty patří i Python.

Polymorfismus

Nenechte se vystrašit příšerným názvem této techniky, protože je v jádru velmi jednoduchá. Polymorfismus umožňuje používat jednotné rozhraní pro práci s různými typy objektů. Mějme například mnoho objektů, které reprezentují nějaké geometrické útvary (kruh, čtverec, trojúhelník). Bylo by jistě přínosné a přehledné, kdybychom s nimi mohli komunikovat jednotně, ačkoli se liší. Můžeme zavést třídu *GeometrickyUtvar*, která by obsahovala atribut *barva* a metodu *vykresli()*. Všechny geometrické tvary by potom dědily z této třídy její interface (rozhraní). Objekty kruh a čtverec se ale jistě vykreslují jinak. **Polymorfismus nám umožňuje přepsat si metodu** *vykresli()* **u** každé **podtřídy tak, aby dělala, co chceme**. Rozhraní tak zůstane zachováno a my nebudeme muset přemýšlet, jak se to u onoho objektu volá.

Polymorfismus bývá často vysvětlován na obrázku se zvířaty, která mají všechna v rozhraní metodu speak(), ale každé si ji vykonává po svém.

Podstatou polymorfismu je tedy metoda nebo metody, které mají všichni potomci definované se stejnou hlavičkou, ale jiným tělem.

STATIKA V PYTHONU

jsou třídní proměnné a metody definované v samotné třídě a jsou nezávislé na instanci. OPP je obsahuje jen pro speciální případy a obecně platí, že vše jde napsat i bez statiky, vždy je tedy třeba zvážit, zda statiku potřebujeme.

Třídní proměnné
Jako třídní můžeme označit různé prvky. Začněme u proměnných. Jak jsem se již v úvodu zmínil, statické prvky patří třídě, nikoli instanci. Data v nich uložená tedy můžeme číst bez ohledu na to, zda nějaká instance existuje. V podstatě můžeme říci, že třídní proměnné jsou společné pro všechny instance třídy, ale není to přesné, protože s instancemi doopravdy vůbec nesouvisí.

Pozor! Při změně třídní proměnné přes instanci změníme pouze hodnotu pro danou instanci.

Jako další praktické využití třídních proměnných se nabízí číslování uživatelů. Budeme chtít, aby měl každý uživatel přidělené unikátní identifikační číslo. Bez znalosti statiky bychom si museli hlídat zvenčí každé vytvoření uživatele a počítat je. My si však můžeme vytvořit přímo na třídě *Uzivatel* privátní statickou proměnnou *dalsi_id*, kde bude vždy připraveno číslo pro dalšího uživatele.

Statické metody
se volají na třídě. Jedná se zejména o **pomocné metody**, které potřebujeme často používat a nevyplatí se nám tvořit instanci.

@staticmethod** vs **@classmethod

Pozor! Díky tomu, že je metoda statická, nemůžeme v ní přistupovat k žádným instančním proměnným. Tyto proměnné totiž neexistují v kontextu třídy, ale instance.

Python obsahuje kromě, statických i **třídní metody.** Tato metody navíc dostávají jako první parametr třídu. **Třídní metody se hodí v tom případě, že budeme třídu dědit a chceme mít v potomkovi jinou hodnotu třídní proměnné.** Jinak je lepší použít statickou metodu.

Pokud použijeme @classmethod, pak první parametr obsahující odkaz na třídu se podle konvencí pojmenovává **cls**. Za pomoci tohoto parametru potom voláme třídní proměnné, podobně jako se self.

Od Pythonu 3 lze navíc slučovat "obyčejné funkce" do tříd. Vypadá to podobně, jako by byly funkce obsažené v nějakém modulu.

DATUM A ČAS V PYTHONU

Knihovna TIME poskytuje informace o aktuálním čase a datu, a pomáhá s přehledným vypsáním pro uživatele a obsahuje krom jiného i tyto muduly:

time()
je základní funkcí modulu time, její návratová hodnota (označovaná jako tick) označuje počet sekund, které uběhly od 1.1.1970. Hodnota jako taková má mnoho využití. Pokud například kdekoliv v našem programu založíme proměnnou s aktuální hodnotou času, později můžeme vytvořit druhou "aktuálnější" hodnotu. Posléze po odečtení obou hodnot zjistíme, kolik času mezi jednotlivými body v programu proběhlo. Podobným způsobem jde například vytvořit i časovaný `while` loop:

```
startTime = time.time()
while (time.time()-startTime) <= 3:     # vlastní kód, který bude probíhat 3 vteřiny
```

clock()
podobně lze využít i metodu `time.clock()`. Ta se od `time.time()` liší tím, že její hodnota udává počet sekund uběhlých od spuštění aktuálního procesu (našeho vlastního programu).

localtime()
Tato metoda nám poskytuje aktuální informace z prostředí uživatele. Všechny informace jsou ve speciálním objektu `struct_time`. Tento objekt (*časová struktura*) je tvořen 9 čísly:

Index	Vlastnost	Hodnota
0	Rok	2019
1	Měsíc	1 - 12
2	Den	1 - 31
3	Hodina	0 - 23
4	Minuta	0 - 59
5	Sekunda	0 - 61
6	Den v týdnu	0 - 6 (0 = Pondělí)
7	Den v roce	1 - 366 (Juliánský den)
8	Letní čas	0 / 1

Jeden způsob, jak data z tohoto objektu dostat, je přímým výběrem pomocí indexu:

```
local = time.localtime()
print(f"Dnes je {local[2]}. {local[1]}. {local[0]}.")
```

asctime()
Pro stručně a čitelně definované formátování můžeme použít funkci `asctime()`. Do funkce vložíme časový objekt (`struct_time`) a vrátí se nám text (*string*) s vypsanými hodnotami v předem určeném pořadí:

```
local = time.localtime()
print(f"Datum a čas: {time.asctime(local)} ")
```

sleep()
Užitečná funkce `sleep()` nám umožňuje náš program na nějakou dobu pozdržet a to na přesný časový úsek, který předáme jako parametr:

```
print("Start")
time.sleep(3) #program se na 3 vteřiny zastaví ("uspí")
print("Konec")
```

Knihovna CALENDAR
přidává nové možnosti formátování výpisu dat pro uživatele a obsahuje krom jiného i tyto muduly:

month()
Při použití musíme specifikovat jaký rok a měsíc si přejeme vybrat. Funkce nám poté vrátí několik řádků textu s velmi přehledným kalendářem

```
print("Kalendář na tento měsíc")
print(calendar.month(2019, 11))
```

Pokud bychom chtěli podobný kalendář pro celý rok, můžeme použít funkci `calendar()` a jako parametr ji předat rok. Ta ale vrátí velké množství textu. Je tedy asi lepší používat jednotlivé měsíce, u kterých, pokud bude potřeba, můžeme dělat vlastní změny.

monthrange(rok, měsíc)
Když budeme o daném měsíci potřebovat více informací, dobře využijeme `monthrange()`. Dostaneme z ní dvě čísla a to ve tvaru *(co je první den v měsíci za den v týdnu, celkový počet dní v měsíci)*:

```
print(calendar.monthrange(2019, 11))
```

U určování dne v týdnu musíme dát pozor na 2 věci:

- *Pondělí má index 0 a neděle index 6*
- *Leden má index 1, prosinec index 12*

weekday()
Funkce `weekday()` nám bude užitečná, pokud budeme chtít rychle zjistit, jakým dnem v týdnu bylo/je/bude nějaké datum. Parametry jsou rok, měsíc a den. Funkce vrátí jediné číslo a to index dne v týdnu odpovídající zadanému datu:

```
dny_v_tydnu={0:"pondělí", 1:"úterý", 2:"středa", 3:"čtvrtek", 4:"pátek", 5:"sobota", 6:"neděle"}
den5 = calendar.weekday(2019, 11, 5)
print("5. listopadu bylo " + dny_v_tydnu[den5])
```

isleap()
Pokud budeme chtít rychle zjistit, jestli je daný rok přestupný nebo ne, poslouží nám funkce `isleap()`. Funkce nám vrátí jako odpověď pouze `True/False`.

```
leap1 = calendar.isleap(2019)
print("2019 přestupný? {}".format(leap1))
leap2 = calendar.isleap(2020)
print("2020 přestupný? {}".format(leap2) + '\n')
```

Knihovna DATETIPE
používá se, vedle modulů `time` a `calendar`, pro práci s datem a časem a podporuje příjemnější, objektový přístup. Pracuje totiž s vlastními specializovanými `datetime` objekty. To nám snadno umožní od sebe jednotlivá data například odečítat nebo je navzájem porovnávat bez nutnosti zaobírání se počtem sekund od roku 1970 nebo indexy v tuple. Nyní si ukážeme, jak se tyto objekty zakládají a jak s nimi dál pracovat v jednotlivých modulech.

datetime()
Velmi významnou třídou v této knihovně je stejnojmenná třída `datetime` a její metoda `datetime()`. Tu můžeme používat pro vytváření instancí, se kterými budeme dále pracovat. Také poskytuje užitečné metody, které budeme hojně využívat.
Vytvořme si instanci data a času z určitého roku, měsíce a dne:

```
# datetime.datetime(rok, měsíc, den, hodina, minuta, sekunda, mikrosekunda)
print(datetime.datetime(2019, 11, 24))
```

Pokud bychom chtěli používat pouze část knihovny, která se vztahuje ke třídě `datetime`, můžeme pozměnit import do varianty: from *datetime* import *datetime*

Metody na třídě datetime:

now()
Metoda `now()` nám jednoduše umožní získat aktuální datum a čas:
print(datetime.datetime.now())
Tato metoda vytvoří instanci předem zmíněné třídy `datetime.datetime`, ve které můžeme najít aktuální informace. K jednotlivým údajům se dostaneme velmi lehce a to pomocí atributů `year, month, day, hour, minute` a `second`:

```
now = datetime.datetime.now()
print("Příklad vypsání data rozděleně:")
print(f"Rok: {now.year}")
print(f"Měsíc: {now.month}")
print(f"Den: {now.day}")
print(f"Čas: {now.hour}:{now.minute}:{now.second}")
```

weekday()
Metodu opět voláme na předem vytvořené instanci a zjistí nám její den v týdnu. Zkusme si ji zavolat na aktuálním datu a času, který také již umíme zjistit:

```
dny_v_tydnu={0:"pondělí", 1:"úterý", 2:"středa", 3:"čtvrtek", 4:"pátek", 5:"sobota", 6:"neděle"}
den = dny_v_tydnu[now.weekday()]
print(f"Den v týdnu: {den}")
```

(Opět zde musíme dát pozor na správné značení: Pondělí = `0` - Neděle = `6`.)

strftime()

Tato metoda nám umožní vytvořit použitelný a lehce čitelný formát data a času pro uživatele:

```
now = datetime.datetime.now()
print(f'text2: {now.strftime("Hodin:%H, Minut:%M, Sekund:%S")}')
print(f'text3: {now.strftime("Den:%d, Měsíc:%m, Rok:%Y")}')
```

Při použití musíme uvnitř metody `strftime()` definovat, jak má náš chtěný výstup vypadat:

%d – den
%m – měsíc
%Y – rok
%H – hodina
%M – minuta
%S – sekunda
%a - zkratka dne v týdnu
%A - den v týdnu
%w - den v týdnu - číselná hodnota
%b - zkratka měsíce (ne číselná podoba)
%B - měsíc (celý název)

strptime()

Opačným způsobem pracuje funkce `strptime()`. Do té vložíme datum a čas zapsaný v textové podobě, např. vložený uživatelem. Poté musíme specifikovat, jaký formát je použit a funkce nám vrátí instanci `datetime`:

```
info = "25.5.2000 12:35"
print(f'>>> strptime:\n{datetime.datetime.strptime(info, "%d.%m.%Y %H:%M")}\n')
```

V této metodě se pro nás ukrývá mnoho možností, a je pouze na nás, jaký formát pro získávání data a času si zvolíme.

Práce s daty

To, že vše ukládáme jako instance té samé třídy, nám přináší při práci s daty mnoho výhod:

- S daty můžeme provádět například základní matematické operace jako je sčítání a odčítání.
- Dvě data můžeme mezi sebou i porovnávat a to obyčejnými operátory, které bychom použili při porovnávání čísel.

timedelta

Pro další možnosti práce s časem je zde `timedelta`. Pokud od sebe dvě data například odečteme, získáme instanci třídy `datetime.timedelta`, což není konkrétní datum, ale interval (rozdíl) mezi nějakými dvěma daty. Tento objekt si můžeme také sami založit. Můžeme tak třeba lehce zjistit, jaký den bude za 10 dní:

```
now = datetime.datetime.now()
td10 = datetime.timedelta(10)
d1 = now + td10
d2 = now - td10
print(f"Datum za 10 dní: {d1}")
print(f"Datum před 10 dny: {d2}")
```

Konstruktor *timedelta()* nemusí být vždy použit s celými dny. Pokud chceme pracovat s jinou hodnotou, máme tu možnost. Při zakládání objektu *timedelta* můžeme navolit mnoho parametrů:

```
datetime.timedelta(days=0, seconds=0, microseconds=0, milliseconds=0, minutes=0, hours=0, weeks=0)
```

Instance timedelta má na sobě vlastnosti jako days a další. Bohužel na sobě nemá počet let, protože rok není plnohodnotná časová jednotka a nemá pevně daný počet dní (některé roky jsou přestupné).

replace()

Velmi užitečná je také metoda replace(). Ta nám, podobně jako timedelta(), umožňuje měnit již vytvořené časové objekty. Zvyšme např. rok a den na objektu datetime o 1:

```
print(f"Původní datum: {now.date()}")
r1 = now.replace(year = now.year + 1)
d1 = now.replace(day =  now.day + 1)
print(f'Změnění roku: {r1.date()}')
print(f'Změnění dne: {d1.date()}\n')
```

Na jakýkoli časový objekt můžeme použít metodu replace(). V ní musíme specifikovat co chceme měnit (např. year = ...) a říci na jakou hodnotu se má změnit. Jde pouze o změnu hodnoty již založené proměnné.

date()

Všechny metody, které jsme si zmínili dříve, pracují s objekty datetime (datum a čas). Pokud bychom chtěli využívat pouze datum, je vhodné místo tohoto používat objekt date. Ten poté nemá informaci o datu a času, která by nám jinak mohla znepříjemňovat např. porovnávání. Objekt date získáme z objektu datetime pomocí metody date():

```
print(f"Dnešní datum: {datetime.date.today()}")
d = datetime.date(2000, 12, 24)
print(f"Jiné datum: {d}")
print(f"Výpis pouze roku: {d.year}")
print(f"Výpis pouze měsíce: {d.month}")
print(f"Výpis pouze dnu: {d.day}")
```

Není na tom nic složitého. Jediná změna od *datetime* je v použití metody *today(),* která přímo nahrazuje *now().* Veškeré metody používané výše jsou s tímto formátem naprosto kompatibilní. Pokud tedy ve svých programech nepotřebujeme časové informace, můžeme se jich poměrně lehce zbavit.

VLASTNOSTI V PYTHONU

Pokud nechceme aby nějaké naše atributy byli volně měnitelné zvenčí, můžeme je změnit dvěma podtržítkami před názvem atributu, na soukromé. Tímto zamezíme uživateli v přímém přístupu k atributu. Pokud se jedná o atribut, který chceme mýt soukromý, ale zároveň chceme jeho hodnotu mít k dispozici, či ji poupravovat, pak pro daný atribut musíme nastavit tzv. metody **setter** a **getter.**

Ruční psaní getterů a setterů může být zdlouhavé a tak v Pythonu lze použít kratší a lepší zápis, u kterého pak už nemluvíme o **atributech**, ale **vlastnostech**.

Syntaxe vlastnosti je velmi podobná metodě:

```
@jmeno_dekoratoru
def nejaka_metoda(self):
    ...
```

Vytvoříme metodu, která má stejný název jako požadovaný název vlastnosti. V těle metody vrátíme atribut spojený s vlastností. Metodu dekorujeme **dekorátorem property**. Tím se z metody stane vlastnost.

Ve skutečnosti jsou dekorátory objekty podporující volání (např. funkce, metody nebo objekty s metodou __call__()), které vrací upravenou (dekorovanou) verzi metody nebo funkce. Proto lze použít i následující syntaxi:

```
puvodni_nazev_metody = jmeno_dekoratoru()
```

V Pythonu máme privátní atribut a k němu máme dvě metody, které podle kontextu Python volá (pozná dle situace, zda čteme nebo zapisujeme). Když do vlastnosti nepřidáme metodu pro setter, nepůjde měnit ani zevnitř, ani zvenčí. Vlastnost se poté používá podobně jako normální atribut:

```
print(objekt.nazev_vlastnosti) # čtení
objekt.nazev_vlastnosti = hodnota # zápis
```

Pokud si přejeme, aby se vlastnost uměla i zapisovat, než jen číst, musíme si dodefinovat i setter:

```
@property
def vek(self):
    return self.__vek

@vek.setter
def vek(self, hodnota):
    self.__vek = hodnota
    self.__plnolety = (hodnota > 18)
```

Zprvu je nutné si vytvořit privátní proměnnou **__vek**, ve které bude hodnota ve skutečnosti uložena. V setteru použijeme další parametr, do kterého se uloží hodnota pro přiřazení. S *vek* nyní pracujeme opět stejně, jako s atributem. Nenápadné přiřazení do věku vnitřně spustí další logiku k přehodnocení atributu *plnolety.*

Stejně můžeme pochopitelně implementovat i vlastní getter a například něco někam logovat:

```
@nazev_vlastnosti.getter
def nazev_vlastnosti(self):
    return soukromy_atribut_vlastnosti
```

Pokud chceme, aby se getter choval jinak, tak si tělo metody upravíme. Ovšem getter musí stále něco vracet, jinak by to nebyl getter.

__dict__ a __slots__

Pokud děláme vlastnosti, tak můžeme použít jako "úložiště" atributu buď privátní atribut (viz výše), nebo veřejný atribut. Pokud ale použijeme veřejný atribut, tak se nám překryjí názvy atributu a metody vlastnosti a program upadne do rekurze. Jde to ale obejít. Veškeré atributy objektů se uchovávají ve slovníku spojeném s objektem. Dostaneme se k němu takto:

```
nazev_objektu.__dict__
```

Poté můžeme číst/přiřadit hodnotu bez rekurze:

```
nazev_objektu.__dict__["nazev_promenne"] # čtení
nazev_objektu.__dict__["nazev_promenne"] = hodnota # zápis
```

Pokud ovšem nemáme pádný důvod, aby byl atribut vlastnosti veřejný, tak je to zbytečné. Navíc je zde jedno úskalí. Takto lze měnit hodnoty i z vnějšku, aniž by nám prošly kontrolou. Ošetřit to můžeme pomocí *__slots__,* což by se ale nemělo vůbec použít. Podle dokumentace Pythonu bychom měli použít *__slots__* maximálně pokud vytváříme velké množství instancí nějaké třídy a chceme za každou cenu ušetřit paměť.

Přesto se dá k soukromým atributům proniknout za pomoci komolení jmen - k atributu se dostaneme za pomoci následující syntaxe:

```
objekt.__NazevTridy_nazevatributu
```

MAGICKÉ METODY

Magické metody pro vytváření objektů

Magické metody objektů jsou takové metody, které začínají a končí dvěma podtržítky.

__new__(cls, *args, **kwargs)

Metodu *__new__* voláme, když potřebujeme kontrolu nad vytvářením objektu. Zejména pokud máme vlastní třídu, která dědí od vestavěných tříd jako například *int* (číslo) nebo *str* (řetězec). Někdy je lepší pro danou situaci použít deskriptory nebo návrhový vzor Factory. Metoda __new__ vrací buď vytvořený objekt nebo nic. Pokud objekt vrátí, tak se zavolá metoda *init*, pokud ne, tak se metoda *__init__* nevolá.

```
class Test:
    def __new__(cls, fail=False):
        print("Zavolána metoda __new__")
        if not fail:
            return super().__new__(cls)
    def __init__(self):
        ("Zavolána metoda __init__")
test_1 = Test()
test_2 = Test(fail=True)
```

Metoda *__new__* bere jako první paramatert třídu daného objektu a poté další argumenty předané v konstruktoru. Třída jako parametr se do metody *__new__* přidává automaticky. Pokud tvorba objektu proběhne úspěšně, tak se i metoda *__init__* volá s parametry z konstruktoru.

V metodě __new__ můžeme dokonce už přiřazovat atributy k objektu:

```
def __new__(cls, x, y):
    self = super().__new__(cls)
    self.x = x
    self.y = y
    return self
```

__init__(self, *args, **kwargs)

Metoda *__init__* se volá při inicializaci objektů. Jako první parametr bere objekt (*self*), který je předán automaticky. Metoda *__init__* by měla vracet pouze *None*, který Python sám vrací, pokud metoda nemá specifikovaný návratový typ. Pokud metoda *__init__* vrací něco jiného než *None*, tak se vyvolá *TypeError.*

__del__(self)

Metoda *__del__* se nazývá destruktor objektu a volá se při zničení objektu, ovšem její chování záleží na konkrétní implemetaci Pythonu. V CPythonu se volá, pokud počet referencí na objekt klesne na nulu. Příkaz *del* nezpůsobí přímé zavolání metody *__del__,* pouze sníží počet referencí o jednu. Navíc není žádná záruka, že se metoda *__del__* zavolá při ukončení programu. Pro uvolňování zdrojů je lepší použít blok *try-finally* nebo *správce kontextu.*

Magické metody pro reprezentace objektů

__repr__(self)

Metoda by měla vracet reprezentaci zdrojového kódu objektu jako text tak, aby platilo:

```
x = eval(repr(x))
```

__str__(self)

Tato magická metoda by měla vracet lidsky čitelnou reprezentaci objektu a měla by vracet řetězec stejně jako metoda *__repr__().*

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __str__(self):
        return "x: {0.x}, y: {0.y}".format(self)
    def __repr__(self):
        return "Point({0.x}, {0.y})".format(self)
point = Point(10, 5)
print(point)
new_point = eval(repr(point))
```

__bytes__(self)

Tato metoda by měla vrátit reprezentaci objektu za pomoci bytů, což znamená, že by měla vracet objekt typu bytes.

__format__(self, format_spec)

Metoda slouží k formátování textové reprezentace objektu. Je volána metodou řetězce format (str.format()). K formátování lze použít vestavěnou funkci *format*, která je syntaktický cukr pro:

```
def format(value, format_spec):
    return value.__format__(format_spec)
```

Pokud se metoda nepřepíše, tak použití metody format vyvolá TypeError (od CPythonu 3.4)
Ukázka metody:

```
def __format__(self, format_spec):
    value = "x: {0.x}, y: {0.y}".format(self)
    return format(value, format_spec)
```

Magické metody pro porovnávání

Python dosazuje do porovnávacích metod odkazy na porovnávané objekty. Všechny porovnávací metody vrací *True* nebo *False*, popř. můžou vyvolat *výjimku* , pokud se porovnávání s druhým objektem nepodporuje.

__lt__(self, other)
Menší než (less than)

```
x < y
```

__le__(self, other)
Menší nebo roven (less or equal)

```
x <= y
```

__eq__(self, other)
Rovná se (equal)

```
x = y
```

__ne__(self, other)
Nerovná se (not equal)

```
x != y
```

__gt__(self, other)
Větší než (greater than)

```
x > y
```

__ge__(self, other)
Větší nebo roven (greater or equal)

```
x >= y
```

Příklad (class Point):

```
def __lt__(self, other):
    if isinstance(other, Point):
        return hypot(self.x, self.y) < hypot(other.x, other.y)
    raise TypeError("unordable types: {}() < {}()".format(self.__class__.__name__,
                                                          other.__class__.__name__))
def __le__(self, other):
    if isinstance(other, Point):
        return hypot(self.x, self.y) <= hypot(other.x, other.y)
    raise TypeError("unordable types: {}() <= {}()".format(self.__class__.__name__,
                                                           other.__class__.__name__))
```

Speciální metoda *__class__* uchovává odkaz na třídu objektu. V ukázce jsou uvedeny jen implementace metod *__lt__* a *__le__*, zbytek je podobný. Dva objekty typu Point porovnáme podle vzdálenosti od počátku souřadnic (tj. bod [0, 0]). Modul functools nabízí automatické vygenerování ostatních metod za pomoci jedné z metod *__lt__(), __lg__(), __gt__()* nebo *__ge__()* a třída by měla mít i metodu *__eq__().* Avšak použití dekorátoru total_ordering může mít negativní účinek na rychlost programu.

Vzhledem k tomu, že metoda __class__() obsahuje odkaz na třídu, tak za pomoci ní lze objekt "naklonovat":

```
def clone(self):
    return self.__class__(self.x, self.y)
```

Další magické metody

__hash__(self)
Metoda hash by měla být reimplementována, pokud je definována metoda __eq__(), aby bylo možné použít objekt v některých kolecích.

```
def __hash__(self):
    return hash(id(self))
```

Funkce id() vrací adresu objektu v paměti, která se pro daný objekt nemění.

__bool__(self)
Metoda vrací True nebo False, podle toho jak se daný objekt vyhodnotí. Číselné objekty dávají False pro nulové hodnoty a kontejnery (seznam, n-tice, ...) dávají False, pokud jsou prázdné.

```
def __bool__(self):
    return bool(self.x and self.y)
```

Volání objektu

__call__(self, *args, **kwargs)
Díky této metodě může volat objekt jako by to byla funkce. Získáme tak například "vylepšenou" funkci, která si může uchovávat stavové informace. Například faktoriál, který si ukládá vypočítané hodnoty:

```
class Factorial:
    def __init__(self):
        self.cache = {}
    def fact(self, number):
        if number == 0:
            return 1
        else:
            return number * self.fact(number-1)
    def __call__(self, number):
        if number in self.cache:
            return self.cache[number]
        else:
            result = self.fact(number)
            self.cache[number] = result
            return result
```

Správce kontextu
Metody *__enter__()* a *__exit__()* slouží, k vytváření vlastních správců kontextu. Správce kontextu před vstupem do bloku *with* se zavolá metoda *__enter__()* a při výstupu se zavolá metoda *__exit__()*.

__enter__(self)
Metoda se zavolá při vstupu do kontextu správcem kontextu. Pokud metoda vrací nějakou hodnotu, tak se uloží do proměnné následující za výrazem as:

```
with smt_ctx() as value:
    do_sth() # zde provádíme příkazy
```

__exit__(self, type, value, traceback)
Tato metoda se volá při opouštění bloku *with*. Proměnná type obsahuje výjimku, pokud v bloku *with* nějaká nastala, pokud ne, tak obsahuje *None*.

Magické metody – Matematické

Obyčejné operátory
Tyto operátory se volají standardně na objektu *a*, pokud je použijeme takto:

```
c = a + b
```

Objekt *a* je předán jako parametr self a objekt *b* jako parametr other.

__add__(self, other)
Metoda se zavolá na prvním objektu při použití operátoru +:

```
c = a + b
```

__sub__(self, other)
Metoda se zavolá při použití operátoru odečítání -:

```
c = a – b
```

__mul__(self, other)
Metoda se zavolá při použití operátoru násobení *:

```
c = a * b
```

__truediv__(self, other)
Metoda se zavolá při použití operátoru dělení /:

```
c = a / b
```

__floordiv__(self, other)
Metoda se zavolá při použití operátoru celočíselné dělení //:

```
c = a // b
```

__mod__(self, other)
Metoda se zavolá při použití operátoru zbytek po dělení - modulo %:

```
c = a % b
```

__divmod__(self, other)
Vrací dvojici (a // b, a % b) pro celá čísla:

```
c = divmod(a, b)
```

__pow__(self, other, modulo)
Metoda se spustí když použijeme operátor mocniny **. Metoda by měla být schopná brát i třetí, nepovinný argument (modulo):

```
c = a ** b
```

__lshift__(self, other)
Spuštěno při použití operátoru pro bitový posun vlevo:

```
c = a << b
```

__rshift__(self, other)
Spuštěno při použití operátoru pro bitový posun vpravo:

```
c = a >> b
```

__and__(self, other)
Spuštěno při použití bitového operátoru AND &:

```
c = a & b
```

__xor__(self, other)
Spuštěno při použití bitového operátoru XOR ^ (non-ekvivalence):

```
c = a ^ b
```

__or__(self, other)
Spuštěno při použití bitového operátoru OR |:

```
c = a | b
```

Prohozené operátory
Prohozené (reversed) operátory se zavolají na druhém objektu, pokud není poskytnuta jejich implementace na prvním objektu. Např.:

```
sth = 1 + my_object
```

Pokud int nepodporuje magickou metodu *__add__(),* což pravděpodobně ne, je zavolána metoda *__radd__()* na *my_object*. Tyto operátory se volají standardně na objektu *b*, přičemž objekt *b* je parametr *self* a objekt *a* je parametr *other*:

__radd__(self, other) – Sčítání

__rsub__(self, other) - Odečítání

__rmul__(self, other) - Násobení

__rtruediv__(self, other) - Pravé dělení

__rfloordiv__(self, other) - Celočíselné dělení

__rmod__(self, other) - Zbytek po dělení - modulo

__rdivmod__(self, other) - Vrací dvojici (a // b, a % b) pro celá čísla.

__rpow__(self, other, modulo) - Mocnina. Metoda by měla být schopná brát i třetí, nepovinný argument (modulo).

__rlshift__(self, other) - Bitový posun vlevo

__rrshift__(self, other) - Bitový posun vpravo

__rand__(self, other) - Logická funkce AND

__rxor__(self, other) - Logická funkce XOR (non-ekvivalence)

__ror__(self, other) - Logická funkce OR

Operátory in place

Tyto operátory umožňují zkrácenou notaci (na místě). Parametry metod jsou self a other, ale vrací modifikovaný self. Pokud některá metoda neexistuje, Python se ji pokusí emulovat s využitím definovaných metod. Příklad:

```
my_object += 1
```

Python zavolá metodu __iadd__(). V případě neúspěchu zavolá metodu __add__() tímto způsobem:

```
temp = my_object + 1 # zavolá __add__()
my_object = temp
```

__iadd__(self, other) - Sčítání

__isub__(self, other) - Odečítání

__imul__(self, other) - Násobení

__itruediv__(self, other) - Pravé dělení

__ifloordiv__(self, other) - Celočíselné dělení

__imod__(self, other) - Zbytek po dělení - modulo

__ipow__(self, other, modulo) - Mocnina. Metoda by měla být schopná brát i třetí, nepovinný argument (modulo).

__ilshift__(self, other) - Bitový posun vlevo

__irshift__(self, other) - Bitový posun vpravo

__iand__(self, other) - Logická funkce AND

__ixor__(self, other) - Logická funkce XOR (non-ekvivalence)

__ior__(self, other) - Logická funkce OR

Další magické metody

__neg__(self)
Unární mínus

```
-a
```

__pos__(self)
Unární plus

```
+a
```

__abs__(self)
Absolutní hodnota, implementuje chování pro funkci abs()

```
abs(a)
```

__invert__(self)
Unární inverze

```
~a
```

__complex__(self)
Implementace chování pro funkci complex()

```
complex(a)
```

__int__(self)
Implementace chování pro funkci int()

```
int(a)
```

__float__(self)
Implementace chování pro funkci float()

```
float(a)
```

__round__(self, n)
Implementace chování pro funkci round()

```
round(a)
```

__index__(self)
Python tuto metodu používá při konverzi numerických typů na int, například při ořezávání nebo pro použití vestavěných funkcí bin(), hex() a oct(). Tato metoda by měla vracet stejný výsledek jako magická metoda __int__(). A navíc by měla vracet celé číslo (int).

Magické metody v Pythonu - Kolekce a deskriptory

__len__(self)
Vrací počet položek v dané kolekci.

__getitem__(self, key)
Vrací položku z kolekce podle zadaného klíče, syntaxe platí pro třídy podobné slovníkům. Při neúspěchu má vyvolat *KeyError*.

__missing__(self, key)
Metoda je volána metodou *__getitem__(),* pokud daný klíč v podtřídě slovníku chybí.

__getitem__(self, index)
Vrací položku z kolekce pro danou pozici. Při neúspěchu má vyvolat *IndexError* (ve for cyklu označí IndexError konec posloupnosti/řady).

__setitem__(self, key, value)
Nastaví hodnotu položky v kolekci určenou klíčem, syntaxe platí pro třídy podobné slovníkům. Při neúspěchu má vyvolat *KeyError*.

__setitem__(self, index, value)
Nastaví hodnotu položky v kolekci určenou pozicí. Při neúspěchu má vyvolat *IndexError*.

__delitem__(self, key)
Smaže z kolekce položku s daným klíčem, syntaxe platí pro třídy podobné slovníkům. Při neúspěchu má vracet *KeyError*.

__delitem__(self, index)
Smaže z kolekce položku s danou pozicí. Při neúspěchu má vracet *IndexError*

__iter__(self)
Vrací iterátor pro kolekci. *Iterátor* lze vytovřit za pomoci funkce *iter()* z objektu, který má magickou metodu *__iter__()* nebo *__getitem__().*

__reversed__(self)
Vrací obrácený iterátor pro kolekci, mělo by se reimplementovat, pokud existuje rychlejší implementace než:

```
for i in reversed(range(len(self))):
    yield self[i]
```

ABSTRAKTNÍ BÁZOVÉ TŘÍDY

Abstraktní bázové třídy nám poskytují seznam rozhraní, které bychom měli implementovat. Rozhraní jsou všechny veřejné metody, včetně metod magických. Pokud implementujeme potřebné minimum, další metody se z nich odvodí.

Třída	dědí od	abstraktní metody	odvozené metody
Container		__contains__	
Hashable		__hash__	
Iterable		__iter__	
Iterator	Iterable	__next__	__iter__
Sized		__len__	
Callable		__call__	
Sequence	Sized, Iterable, Container	__getitem__, __len__	__contains__, __iter__, __reversed__, index, a count
MutableSequence	Sequence	__getitem__, __setitem__, __delitem__, __len__, insert	metody Sequence + append, reverse, extend, pop, remove a __iadd__
Set	Sized, Iterable, Container	__contains__, __iter__, __len__	__le__, __lt__, __eq__, __ne__, __gt__, __ge__, __and__, __or__, __sub__, __xor__ a isdisjoint
MutableSet	Set	__contains__, __iter__, __len__, add, discard	metoda Set + clear, pop, remove, __ior__, __iand__, __ixor__, a __isub__
Mapping	Sized, Iterable, Container	__getitem__, __iter__, __len__	__contains__, keys, items, values, get, __eq__ a __ne__
MutableMapping	Mapping	__getitem__, __setitem__, __delitem__, __iter__, __len__	metody Mapping + pop, popitem, clear, update a setdefault
MappingView	Sized		__len__
ItemsView	MappingView, Set		__contains__, __iter__
KeysView	MappingView, Set		__contains__, __iter__
ValueView	MappingView		__contains__, __iter__

Řízení atributů

__getattr__(self, name)
Tato metoda se volá, pokud selže hledání atributu přes obvyklé metody. Metoda by měla vrátit hodnotu atributu nebo vyvolat *AttributeError.*

__setattr__(self, name, value)
Volá se při nastavování atributů namísto přiřazování proměnných do *__dict__*. Pokud chce třída atribut uložit, měla by zavolat metodu *__setattr__()* nadtřídy.

__delattr__(self, name)
Mělo by se implementovat pouze v případě, když:

```
del obj.name
```

má nějaký smysluplný význam pro třídu. Ukázka:

```
class Foo:
    def __setattr__(self, name, value):
        print("Setting")
        super().__setattr__(name, value)
    def __getattr__(self, name):
        print("Getting")
        return 1
foo = Foo()
print(foo.x)
print(foo.__dict__)
foo.x = 2
print(foo.x)
print(foo.__dict__)
```

Deskriptory

Deskriptory umožňují přepsání přístupu k atributům za pomoci jiné třídy. Můžeme se tak vyhnout zbytečnému opakování property. Třída musí definovat minimálně jednu z metod *__get__(), __set__()* nebo *__delete__().* Díky deskriptoru můžeme vylepšit validování hodnot atributů. Vytvoříme si ukázkový deskriptor Typed, který bude zajišťovat, aby byl atribut určitého typu:

```
from weakref import WeakKeyDictionary
class Typed:
    dictionary = WeakKeyDictionary()
    def __init__(self, ty=int):
        self.ty = ty
    def __get__(self, instance, owner=None):
        print("Getting instance")
        return self.dictionary[instance]
    def __set__(self, instance, value):
        print("Setting instance", instance, "to", value)
        if not isinstance(value, self.ty):
            raise TypeError("Value must be type {}".format(self.ty))
        self.dictionary[instance] = value
```

__get__(self, instance, owner)
Metoda get přijímá jako parametr instanci a třídu. Třídu instance můžeme získat takto:

```
owner = type(instance)
```

V kódu vrátíme hodnotu vlastnosti uloženou v deskriptoru.

__set__(self, instance, value)
Metoda set přijímá jako parametr instanci a hodnotu, na kterou má být atribut nastavený. Při ukládání kontrolujeme, jestli je atribut daného typu. Běžně je nastavený na *int* (celé číslo). Pokud není, vyvoláme výjimku *TypeError.*

Slovník slabých referencí
Pro ukládání atributů používáme slovník slabých referencí (*WeakKeyDictionary*) z modulu weakref. Ten funguje jako normální slovník, avšak liší se v jednom detailu. Pokud pro daný objekt (klíč) ve slovníku neexistuje více referencí než ta ve slovníku, tak se objekt ze slovníku smaže. Díky tomu zamezíme memory leakům v paměti. Obyčejný slovník by zbytečně bránil odstranění objektu z paměti, i když by se již nikde jinde nepoužíval, protože právě slovník by jej používal. Pokud používáme třídu se *__slots__*, musíme do *__slots__* přidat "*weakref*", aby třída podporovala slabé odkazy.

ZÁKLADNÍ TYPY V PYTHONU A JEJICH POUŽITÍ

Základní typy v Pythonu jsou:

- **int** pro celé číslo
- **float** pro reálné číslo
- **bool** pro hodnoty *True*/*False*
- **str** pro řetězce
- **None** pro *None*

Pokud chceme explicitně deklarovat typ proměnné, použijeme k tomu dvojtečkový operátor ::

```
text: str = "Ahoj světe!"
```

Nyní je jasně řečeno, že text je typu str. Zde by to bylo stejně poznat podle hodnoty "Ahoj světe!", takže je tu takové označení trochu zbytečné. V programu je ale spoustu míst, kde to již tak jasné není. Např. můžeme typovat i parametry funkcí a jejich návratové hodnoty:

```
def generuj_ahoj(jmeno: str) -> str:
    return "Ahoj, " + jmeno + "!"
```

Těmito nástroji je pak možné zkontrolovat, zda nám vše sedí, případně vygenerovat k funkcím dokumentaci včetně typů.

Další typy v Pythonu

Samozřejmě toto nejsou všechny typy, které se mohou v Pythonu objevit, např. proměnná typu list můžeme označit pomocí:

```
muj_list: list = [1, 2, 3]
```

Toto by nám ale nebylo moc užitečné. Sice bychom věděli, že daná proměnná je list, ale nevěděli bychom jakého typu jsou prvky, které se v tomto listu nacházejí a přesně s tímto nám pomůže *knihovna typing*.

Knihovna typing

```
from typing import List, Set, Dict, Optional, Callable
```

1) Typování seznamů a množin

První dvě importované položky nám přijdou jistě intuitivní, typ prvků v nich uložených se určuje pomocí hranatých závorek:

```
muj_list: List[int] = [1, 2, 3]
muj_set: Set[int] = set(muj_list)
```

2) Slovníky

Pokud chceme typovat slovník, s jedním takovým parametrem si nevystačíme. Budeme tedy potřebovat parametry dva - první pro typ klíče a druhý pro typ hodnot:

```
muj_slovnik: Dict[int, str] = {1: 'jedna', 2: 'dva', 3: 'tri'}
```

Samozřejmě můžeme jako parametry používat i složitější typy, než jen a pouze ty primitivní:

```
muj_slovnik2: Dict[int, List[str]] = {1: ['jedna', 'one'], 2: ['dva', 'two'], 3: ['tri', 'three']}
```

3) Optional

Kdy se nám ale může hodit typ *Optional*? *Optional* ve zkratce značí to, že daná proměnná může nabývat hodnot typu prvního parametru nebo *None*. To se nám hodí například, pokud bychom chtěli vytvořit funkci na dělení a chtěli zabránit dělení nulou:

```
def deleni(a: float, b: float) -> Optional[float]:
    if b == 0:
        return None  # nesmíme dělit nulou
    return a / b
```

4) Callable

Poslední typ, který jsme si importovali, se jmenuje *Callable*. Jak již název napovídá, jedná se o typ něčeho, co se může volat, tedy funkcí:

```
def pridej_jedna(a: int) -> int:
    return a + 1
operace: Callable = pridej_jedna
print(operace(5))  # vypíše 6
```

Vytváření vlastních typů

Typy v Pythonu můžeme vytvářet třemi způsoby:

- typové aliasy
- třídy
- pojmenovaný slovník

Typové aliasy

Pokud nějaký komplikovaný typ používáme často, můžeme si vytvořit typový alias, který se vytvoří jednoduchým přiřazením původního typu do proměnné. Například se nám to může hodit při definici matic:

```
matice: List[List[int]] = [[1, 2], [3, 4]]
```

můžeme díky vytvoření typovému aliasu napsat jako:

```
Matrix = List[List[int]]
matice: Matrix = [[1, 2], [3, 4]]
```

Třídy

Každá třída se bere také jako vlastní typ, ať už ji importujeme, nebo sami vytvoříme:

```
class MojeTrida:
    pass
moje_promenna: MojeTrida = MojeTrida()
from queue import Queue
fronta: Queue = Queue()
```

Problémem, na který bychom mohli narazit, by bylo, že bychom chtěli využít typ, který je deklarován až dále v souboru (popř. typ dané třídy samotné). V době čtení daného řádku jej tedy parser nezná.

Tomu můžeme předejít tak, že místo přímého odkazu na danou třídu uzavřeme její jméno do uvozovek jako textový řetězec. Např.:

```
from typing import Optional, Any
class LinkedList:
    def __init__(self):
        self.value: Optional[Any] = Noneprvky
        self.next: Optional["LinkedList"] = None
```

Typovaný slovník

Posledním typem vytváření typů v Pythonu, který dnes probereme, je vytvoření typovaného slovníku (*TypedDict*), který se poprvé objevil v Pythonu 3.8. Jedná se o jakousi abstrakci nad klasickým slovníkem, kdy ale explicitně uvedeme, které klíče s jakým typem hodnot se mohou ve slovníku nacházet:

```
from typing import TypedDict
class AdresaDomu(TypedDict):
    mesto: str
    ulice: str
    cislo_domu: int
AdresaDomu = TypedDict('AdresaDomu', mesto=str, ulice=str, cislo_domu=int)
AdresaDomu = TypedDict('AdresaDomu', {'mesto': str, 'ulice': str, 'cislo_domu': int})
moje_adresa: AdresaDomu = {'mesto': 'Stare Mesto', 'ulice': 'Nova ulice', 'cislo_domu': 52}
```

Alternativní systém typování

Tento systém, který jsme si ukázali, je v Pythonu dostupný od verze 3.5, takže pokud používáte Python 3.4 a starší, nemáme dostupné nic z knihovny *typing*, ani dvojtečkový operátor. Přesto ale můžeme typovat - a to pomocí komentářů. Takže například řetězcovou proměnnou bychom deklarovali následovně:

```
muj_text = "Ahoj"  # type: str
```

Tento systém je možné využívat i v Pythonu 3.5 a novějším, přičemž v některých konkrétních situacích je to stále jediný možný způsob typování bez toho, aby nám interpreter hlásil část kódu, kterou neumí zparsovat.

Kolekce v Pythonu

Pojem kolekce označuje v Pythonu **datové typy**, které umožňují ukládat více hodnot v jednom objektu. Mezi **nejběžnější kolekce** v Pythonu patří **seznamy, tuple, slovníky a množiny**. Kolekcí nicméně existuje více, a ačkoli se zvenku mnohdy tváří podobně, uvnitř fungují velmi odlišně. Vybíráme si je tedy podle konkrétního účelu.

Hash table a kolekce

Seznamy, sady a slovníky jsou kolekce založeny na principu **hash table**, což je **datová struktura**, která umožňuje **rychlé vyhledávání, přidávání** a **odebírání prvků**. Hash table funguje tak, že pro každý prvek kolekce se vytvoří hash (číslo) na základě jeho hodnoty. Tato hodnota se poté použije jako **index**, který označuje, kde se prvek uloží. Když se v budoucnu potřebujeme k prvku vrátit, vytvoříme znovu hash a použijeme ho jako **index** pro jeho vyhledání.

Genericita v Pythonu

V Pythonu jsou generické typy implementovány pomocí modulu `typing`, který byl představen v Pythonu 3.5. Modul `typing` nám umožňuje specifikovat typy prvků v kolekci. Generické kolekce v Pythonu jsou **specifické pro daný typ,** což znamená, že obsahují pouze prvky jednoho typu. Cílem generik v Pythonu, stejně jako v jiných programovacích jazycích, je zlepšit reusability (opakovatelnost) kódu a snížit počet duplicitních částí.

Generické třídy a metody umožňují kódu pracovat s různými typy dat bez nutnosti explicitního určení typu. Jedna třída nebo metoda tedy může být použita pro různé typy dat bez nutnosti vytvoření nové verze pro každý konkrétní typ. To umožňuje kódu být flexibilnější. Je ale třeba mít na paměti, že když definujeme proměnnou nebo atribut třídy jako generický typ (např. `List[int]),` Python nezkontroluje v runtime, jestli proměnná nebo atribut skutečně obsahují prvky tohoto typu. To ale může vést k chybám, pokud omylem proměnné přiřadíme hodnotu špatného typu. Python se tím liší od staticky typovaných jazyků, jako jsou Java nebo C#, kde kompilátor zkontroluje v runtime, že proměnná nebo atribut prvky daného typu skutečně obsahuje.

V praxi tedy můžeme říci, že staticky typované jazyky mají větší ochranu proti typovým chybám. Avšak zároveň jsou ale více omezující při definování proměnných a atributů třídy, protože je nutné specifikovat jejich typy explicitně. V Pythonu máme tedy větší flexibilitu, ale současně to znamená, že musíme být opatrnější.

Využití kolekcí v Pythonu:

• ***práce s databází:*** slovníky a seznamy jsou často používány pro ukládání a práci s daty z databáze,
• ***textové procesy:*** seznamy a tuple jsou často používány pro práci s textem například pro rozdělení textu na slova nebo procházení textu po řádcích,
• ***matematické operace:*** množiny jsou často používány pro matematické operace, jako je například práce s množinou unikátních hodnot nebo sjednocení nebo rozdíl množin,
• ***webové aplikace: slovníky*** a seznamy jsou často používány pro práci s daty z webových aplikací, jako jsou například JSON nebo XML soubory,
• ***algoritmy:*** Seznamy a tuple jsou často používány pro implementaci různých algoritmů, jako jsou například prohledávání nebo řazení.

Type hints

slouží pro vytvoření kolekce instancí nějaké třídy:

```
from dataclasses import dataclass
@dataclass
class Postava:
        jmeno: str
        level: int = 1
hrac1 = Postava("Sven Kladivo")
hrac2 = Postava("Rudá Sonja", 2)
host = "Není postava"
postavy: list[Postava] = [hrac1, hrac2]
print(postavy) >>> [Postava(jmeno='Sven Kladivo', level=1), Postava(jmeno='Rudá Sonja', level=2)]
postavy.append(host)
print(postavy) >>> [Postava(jmeno='Sven Kladivo', level=1), Postava(jmeno='Rudá Sonja', level=2), 'Není postava']
```

Uživatelem definované generické typy

Genericita nám umožňuje vytvořit třídu, kterou lze použít s více typy. Současně díky nástrojům jako je `mypy` nedovolí, aby byly do metod instance odesílány jiné argumenty než ty zadaného typu:

```
from typing import Dict, Generic, TypeVar
T = TypeVar("T")
class Rodina(Generic[T]):
        def __init__(self) -> None:
                self._uloz: Dict[str, T] = {}
        def nastav_polozku(self, k: str, v: T) -> None:
                self._uloz[k] = v
        def nacti_polozku(self, k: str) -> T:
                return self._uloz[k]
if __name__ == "__main__":
        jmeno_pribuzneho = Rodina[str]()
        vek_pribuzneho = Rodina[int]()
        jmeno_pribuzneho.nastav_polozku("manželka", "Dana")
        jmeno_pribuzneho.nastav_polozku("dědeček", "Tomáš")
        vek_pribuzneho.nastav_polozku("Tomáš", 70)
```

Typ obsahu třídy je obecný a uvedeme ho ve chvíli, kdy vytváříme instanci této třídy. Po vytvoření instance bude tato instance na rozdíl od předchozího příkladu přijímat pouze argumenty tohoto typu.

Tuples v Pythonu

Tuples (někdy též **uspořádané *n*-tice**) se velmi podobají seznamům. Jedná se o sekvence, ve kterých ale položky nelze dále modifikovat. Jednotlivé položky se oddělují čárkou. Deklarujeme je pomocí kulatých závorek:

```
polozky= (1, 2, 3, 7)
```

V případě, že chceme deklarovat `tuple` s jedinou položkou, abychom ji odlišili od běžné hodnoty proměnné, musíme za položkou napsat čárku (`,`):

```
polozky= (1, 2, 3, 7)
```

Kolekci `tuple` používáme, když potřebujeme někde předat sekvenci *a* **chceme se ujistit, že se náhodou nezmění**. `Tuples` jsou **read-only**, takže pokud potřebujeme `tuple` z nějakého důvodu modifikovat, musíme vytvořit novou s takovými položkami, které zrovna potřebujeme. Toho dosáhneme například **převodem** `tuple` **na seznam a zase zpět**:

```
polozky = (1, 2, 3, 7)
polozky_seznam = list(polozky)
polozky_seznam[1] = 4
polozky = tuple(polozky_seznam)
```

V kódu je důležitá metoda `tuple()` která na `tuple` převádí jiné sekvence. *N*-tice mohou být také sloučeny do sebe prostřednictvím operátoru +. Ke zjištění, kolik položek naše `tuple` obsahuje, použijeme globální funkci `len()`. Můžeme použít i funkce `min()` a `max()` pro určení nejnižší a nejvyšší hodnoty (prostě jako u každé sekvence v Pythonu). Můžeme také použít operátor `in`, cyklus `for`, operátor `[]` pro indexy atd.

Množiny v Pythonu

Množina neboli `set` je druh sekvence podobné seznamu s tím rozdílem, že může obsahovat jen unikátní položky (každá položka může být v množině **pouze** jednou). Tyto **položky nejsou setříděné**, což znamená že pořadí položek není udržováno a může se nepředvídatelně změnit. Pro množiny není žádná zvláštní syntaxe jako v případě seznamů či *n*-tic, vytváříme je jednoduše použitím globální funkce `set()`, které do parametru vložíme sekvenci (i již existující):

```
planety = set(("Země", "Mars", "Jupiter", "Saturn", "Uran", "Neptun"))
```

V ukázce výše jsme vytvořili množinu šesti jmen planet. Dvojité závorky na řádku s funkcí `set()` znamenají, že jsme předali názvy planet formou *n*-tice jako **parametr této funkci**. Pořadí položek není seřazeno podle abecedy, a nezmění se ani po přidání nové položky. To ale není žádná chyba, neboť položky jsou vnitřně udržovány v pořadí, což pomáhá množině efektivně určit jedinečnost každé položky.

Metody množin

add()

Pomocí metody `add()` přidáváme položky do množiny. Z předchozího příkladu je zřejmé, že pokud se pokusíme přidat do množiny již jednou obsažený prvek, nevyskočí žádná chyba, ale **položka prostě není přidána.**

`difference()` a `difference_update()`

Množina poskytuje všechny množinové operace, které známe z matematických tříd. Můžeme se třeba zeptat na rozdíl mezi dvěma množinami. Metoda `difference()` vrací tento rozdíl dvou množin jako novou množinu. Metoda `difference_update()` naproti tomu upravuje stávající množinu a odstraní všechny položky z druhé množiny.

`remove()`, `discard()`, a `pop()`

Všechny z těchto tří metod **odstraní vybranou položku** z množiny. Metoda `remove()` vyhodí chybu, pokud se hledaná položka v množině nevyskytuje. Metoda `discard()` se chová zcela totožně, jen při absenci položky zamýšlené k odstranění chybu nevyvolá. Metoda `pop()` pak vyjme náhodnou hodnotu z množiny a tu potom vrátí.

isdisjoint()

Určuje, zda dvě množiny **nemají žádné společné položky** (průnik).

`issubset()` a `issuperset()`

Můžeme se zeptat, zda je množina podmnožinou (všechny její položky jsou přítomny v druhé množině) nebo nadmnožinou (jsou v ní přítomny všechny položky druhé množiny) pomocí metod `issubset()` pro podmnožinu a `issuperset()` nadmnožinu.

clear()

Tato metoda odstraní všechny položky z množiny (vyčistí ji):

```
mnozina.clear()
```

Slovníky

Slovník (Dictionary) funguje také podobně jako seznam, až na to, že k němu nepřistupujeme **jen** pomocí indexů, ale i na základě klíčových hodnot různých avšak neměnných datových typů. **Index slovníků nazýváme klíč**. Na pořadí jednotlivých položek ve slovníku nezáleží.

Data v sekvenci jsou uložena speciálním způsobem, tzv. hashováním. To nám umožňuje přistupovat k prvkům pomocí klíče mnohem rychleji, než kdybychom je podle této vlastnosti hledali např. v obyčejném seznamu. Zatímco u seznamu je potřeba všechny prvky projít a kontrolovat, zda náhodou jejich vlastnost neodpovídá tomu, co hledáme, slovník dokáže pro prvek sáhnout mnohem rychleji díky výpočtu tzv. hashe (otisku). Finta spočívá v tom, že jsme schopni z klíče zjistit index prvku pomocí hashovací funkce. Pokud budeme mít ve slovníku uložené zaměstnance a klíčem bude jejich jméno, hashovací funkce nám z "Jan Novák" vrátí např. 114. Sáhneme na 114. prvek a hle, je tam Jan Novák. Nemusíme slovník nijak iterovat.

Slovník deklarujeme stejně jako seznam. Hlavní rozdíl je v tom, že používáme složené závorky a musíme k položkám definovat i jejich klíče. K tomu používáme operátor dvojtečka `:`. Slovníky jsou tedy použity k **uložení hodnot v párech *klíč:hodnota*** *(key:value*):

```
oblibeneVeci= {
        'homer': 'koblihy',
```

```
    'marge': 'trouba',
    'bart': 'prak',
    'lisa': 'saxofon',
    'maggie': 'dudlik'}
```

(Zápis položek je rozdělen kvůli přehlednosti do více řádků, ale šlo by to i v jednom.)

Namísto zápisu `obliberneVeci[0]` jsme použili klíč typu `string`. Velkou výhodou použití slovníku je lepší **čitelnost**, kdy přímo vidíme, jakou hodnotu ze slovníku dostáváme. Zatímco u číselných indexů možná ani nevíme, o jakou hodnotu jde. Každý **klíč musí být unikátní**, ale hodnoty takové být nemusí. Páry *klíč: hodnota* mohou být **jakékoliv neměnné datové typy.** Pokud definujeme stejný klíč ve stejném slovníku vícekrát a s různými hodnotami, bude klíči přiřazena poslední hodnota.

Přidávání položek: Do slovníku můžeme jednoduše přidávat další položky přiřazením nových klíčů:

```
oblibeneVeci['maggie'] = 'dudlik'
```

Stejným způsobem můžeme modifikovat již uložené hodnoty.

Zjištění počtu položek: Pro zjištění počtu položek ve slovníku použijeme globální funkci **`len()`** známou již ze seznamů:

```
len(oblibeneVeci)
```

Zjištění, zda položka je v seznamu: Pomocí operátoru **`in`** se zeptáme, zda slovník obsahuje určitý klíč. (V Pythonu 2.x k tomu byla určena metoda `has_key()`, která je ale nyní zastaralá):

```
if simpson in oblibeneVeci:
```

Metody slovníků:

get()

Metoda `get()` nabízí další způsob pro získání položky ze slovníku. Hlavní výhodou této metody je, že nevyhodí žádnou výjimku v případě, že hledaný klíč ve slovníku není. Místo toho vrátí hodnotu `None` nebo některou výchozí hodnotu, specifikovatelnou ve druhém parametru:

```
print(oblibeneVeci.get('krusty'))
print(oblibeneVeci.get('krusty', 'nikdo'))
```

`values()`, `keys()`, a `items()`

Pomocí těchto metod můžeme převést slovník na seznam. Můžeme tak vyčlenit hodnoty, klíče nebo dokonce vytvořit seznamy *n*-tic párů klíč-hodnota:

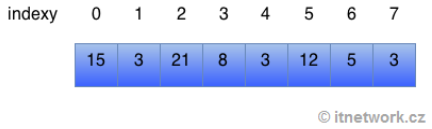
```
print(oblibeneVeci.values())
print(oblibeneVeci.keys())
print(oblibeneVeci.items())
```

clear()

Jak samotný název naznačuje, tato metoda "vyčistí" všechny položky ze slovníku.

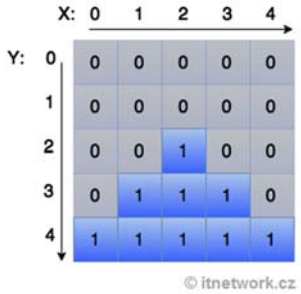
Vícerozměrné seznamy v Pythonu

jednorozměrným seznamem (ve většině ostatních programovacích jazyků se tato struktura označuje **pole**), který si můžeme představit jako řádku přihrádek v paměti počítače:



Dvourozměrný seznam

Dvourozměrný seznam si můžeme v paměti představit jako tabulku a mohli bychom si ho graficky znázornit např. takto:



Python ve skutečnosti neposkytuje žádnou dodatečnou podporu pro vícerozměrné seznamy, ale můžeme si je jednoduše deklarovat jako seznam seznamů:

```
seznam2d= []
for j in range(5):
    sloupec= []
    for i in range(5):
        sloupec.append(0)
    seznam2d.append(sloupec)
```

První číslo představuje číslo sloupce, **druhé číslo** číslo řádku, ale mohli bychom to samozřejmě udělat i obráceně - například matice v matematice mají číslo řádku jako první.

Naplnění daty

Protože budeme jako správní programátoři líní, využijeme k vytvoření řádku jedniček for cykly Pro přístup k prvku 2D seznamu musíme samozřejmě zadat dvě souřadnice:

```
seznam2d[2][2] = 1 # střed
for i in range(1, 4): # čtvrtá řada
    seznam2d[i][3] = 1
for i in range(5): # poslední řada
    seznam2d[i][4] = 1
```

Výpis

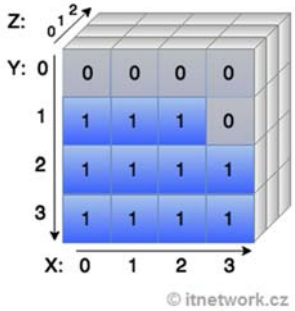
Výpis seznamu opět provedeme pomocí cyklu. Na 2D seznam budeme potřebovat cykly dva (jeden nám proiteruje sloupce a druhý řádky). Funkci `len()` takže můžeme jednoduše zjistit, kolik sloupců

je ve vnějším seznamu a kolik položek v tom vnitřním. Musíme myslet i na to, že vnější seznam může být prázdný. Cykly vnoříme tak, aby vnější cyklus iteroval přes řádky a vnitřní přes sloupce aktuálního řádku. Po vytištění řádku musíme řádek přerušit. **Oba cykly musí mít jinou řídicí proměnnou**:

```
if len(seznam2d):
        radky = len(len(seznam2d)[0])
for j in range(radky):
        for i in range(sloupce):
                print(seznam2d[i][j], end ="")
```

N-rozměrné seznamy

Někdy může být příhodné vytvořit si seznam o ještě více dimenzích. Všichni si jistě dokážeme představit minimálně 3D seznam. Vizualizace pak vypadá třeba takto:



3D seznam vytvoříme tím samým způsobem, jako 2D seznam:

```
seznam3d = []
for i in range(5):
        seznam2d = []
        for j in range(5):
                temp = []
                for k in range(5):
                        temp.append(0)
                seznam2d.append(temp)
        seznam3d.append(seznam2d)
```

Kód výše vytvoří 3D seznamu jako na obrázku. Přistupovat k němu budeme opět přes indexery (hranaté závorky) jako předtím, jen již musíme zadat tři souřadnice:

```
kinosaly[3][2][1] = 1 # Druhý kinosál, třetí řada, čtvrtý sloupec
```

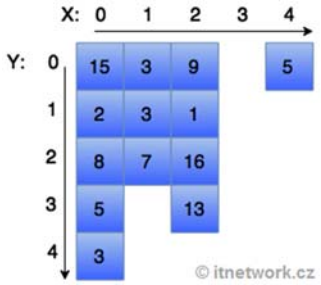
Zkrácená inicializace

vícerozměrné seznamy je možné rovnou inicializovat hodnotami:

```
kinosal = [[0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0],
        [0, 0, 1, 0, 0],
        [0, 1, 1, 1, 0],
        [1, 1, 1, 1, 1]]
```

Zubaté seznamy

V některých případech nemusíme "plýtvat" pamětí na celou tabulku a můžeme seznam vytvořit "zubatý" (anglicky jagged):



Nevýhodou tohoto přístupu je, že musíme seznam nepříjemně inicializovat sami. Původní řádek s pěti buňkami sice existuje, ale jednotlivé sloupečky si do něj musíme vložit sami:

```
zubate = [[15, 2, 8, 5, 3],
        [3, 3, 7],
        [9, 1, 16, 13],
        [],
        [5]]
```

K jeho vypsání opět využijeme vnořený cyklus:

```
for seznam in zubate:
        for prvek in seznam:
                print(prvek, end=" ")
        print(end="\n")
```

Slovo na závěr:

Někteří lidé, kteří neumí správně používat objekty, využívají 2D seznamů k ukládání více údajů o jediné entitě. Např. budeme chtít uložit výšku, šířku a délku pěti mobilních telefonů. Ačkoli se nyní může zdát, že se jedná o úlohu na 3D seznam, ve skutečnosti se jedná o úlohu na obyčejný 1D seznam objektů typu `Telefon`.

Iterátory v Pythonu

Iterovatelný objekt

Pomocí iterace můžeme **postupně získat prvky uložené v nějaké kolekci** (například aplikací `for` cyklu na seznam). Iterovatelný objekt je potom takový **objekt, na kterém lze provést iteraci.** Prvky získáme buď v pevně stanoveném pořadí (seznam) nebo v pořadí náhodném (množina).

Iterátor

Iterátor je ***zodpovědný za iteraci na iterovatelném objektu.*** Pamatuje si, které prvky již poskytl (neposkytne jeden prvek dvakrát). Ve chvíli, kdy už není další prvek k dispozici, nám to oznámí. V tento okamžik jeho úloha v programu končí, iterátor je tzv. **vyčerpaný** (*exhausted*). Pokud chceme znovu iterovat, musíme vytvořit nový iterátor.

Třída **iterovatelného objektu** musí implementovat speciální metodu `__iter__()`, která po zavolání vytvoří a vrátí novou instanci třídy `Iterator` (vytvoří nový „nevyčerpaný" iterátor). Metodu můžeme volat skrze vestavěnou funkci `iter()`. Třída **iterátoru** pak musí implementovat speciální metodu `__iter__()`, která ale vrací odkaz na svoji instanci – `self` (nevytváří novou instanci). Zároveň musí mít implementovanou metodu `__next__()`, která po zavolání vrátí další prvek z kolekce. Pokud již není další prvek k dispozici, vyvolá `StopIteration` výjimku. Metodu lze volat vestavěnou funkcí `next()`.

Je tedy velký rozdíl, jestli iterujeme na iterovatelném objektu (např. seznamu), který zavoláním funkce `iter()` vrací pokaždé nový iterátor, nebo přímo **na iterátoru, který vrací pouze sám sebe**. Technicky je sice iterátor zároveň iterovatelný objekt (oba implementují metodu `__iter__()`). My je ale budeme v této lekci rozlišovat a pokud budeme hovořit o iterovatelném objektu, budeme tím myslet iterovatelný objekt, který není zároveň iterátorem. [note].

Cyklus `for` pod pokličkou

cyklus `for`, je z hlediska iterace klíčový. Python ve skutečnosti aplikuje cyklus `while` za pomoci následujícího mechanismu:

```
seznam = [1, 2, 3, 4, 5]
iterator = iter(seznam)
try:
        while True:
                prvek = next(iterator)
except StopIteration:
        pass
```

Na začátku zavolá funkci `iter()` a obdrží iterátor. Poté na něm opakovaně volá funkci `next()` a získává jednotlivé prvky dokud nenarazí na `StopIteration` výjimku.

Vestavěné iterovatelné objekty a iterátory

Pokud iterujeme seznam (list), je zřejmé, že má definovanou metodu `__iter__()` ale nikoliv metodu `__next__()`. Zavoláním funkce `iter()` na náš seznam získáme jeho iterátor. Jelikož seznam je iterovatelný objekt, měl by vrátit **novou instanci iterátoru**, tedy nikoliv pouze odkaz na sebe.

Na iterátor můžeme volat funkci `next()`, popřípadě metodu `__next__()` a získat postupně jeho prvky:

```
horory = ["Vetřelec", "Frankenstein", "Věc"]
iterator_hororu = iter(horory)
print(next(iterator_hororu)) >>> Vetřelec
print(iterator_hororu.__next__()) >>> Frankenstein
```

V tomtéž kódu můžeme dále použít i cyklus `for`. Jen musíme pamatovat na to, že **iterátor se postupně vyčerpává**:

```
for horor in iterator_hororu:
        print(horor) >>> Věc
```

Jelikož jsme první dva prvky získali funkcí `next()`, `for` cyklus nám vrátil pouze poslední prvek. **Nyní je iterátor vyčerpaný** a pokud bychom chtěli znovu iterovat, museli bychom buď iterovat na původním seznamu (který si příslušný iterátor vytvoří sám automaticky) a nebo si iterátor znovu explicitně vytvořit sami zavoláním funkce `iter()`.

range()

Funkce `range()` vrací objekt `range`, což je **iterovatelný objekt.** Objekt `range` má implementovanou pouze metodu `__iter__()`, která před každou iterací vytvoří novou instanci iterátoru. Iterovat na objektu `range` můžeme tím pádem bez omezení.

enumerate()

Funkce `enumerate()` vrací objekt `enumerate`, což je **iterátor**. Tento objekt má implementovanou jak metodu `__iter__()`, tak metodu `__next__()`. Funkce `iter()` vrací ten samý objekt. Iterovat na objektu `enumerate` můžeme **maximálně jednou**. Funkce `enumerate()` vytvoří dvojice, kde první položkou je index a druhou položkou je příslušný prvek zadaného iterovatelného objektu.

Tabulku nejčastěji používaných funkcí, které vrací iterovatelné objekty nebo iterátory:

Funkce vracející iterovatelný objekt	Funkce vracející iterátor
list()	enumerate()
tuple()	zip()
set()	map()
dict()	filter()
dict.keys()	open()
dict.values()	
dict.items()	
range()	

Iterátory podruhé: Generátory v Pythonu

Implementace vlastního iterátoru

Když si vytvoříme vlastní *iterátor* (třídu) a *iterovatelnný objekt* (instanci), který bude využívat služeb iterátoru. Můžeme získat všechny jeho prvky pomocí *for cyklu*, nebo můžeme procházet jeho prvky pomocí metody `__next__()` která volá další prvek. Pokud nejprve procházíme prvky pomocí metody `__next__()` a po té zpustíme `for cyklus`, bude nám nabízet už jen zbylé prvky sekvence (`for cyklus` je metoda `__next__()` v cyklu. A pokud bychom zavolali metodu `__next__()`, po vyčerpání všech prvků, obdržíme výjimku ***`StopIteration`***. To proto, že **iterátor je vyčerpán**. Pokud chceme znovu iterovat, musíme vytvořit jeho **novou instanci**.

Sofistikovanější přístup je, že do našeho kódu **zakomponujeme iterovatelný objekt** v podobě třídy. Vytvoříme-li pak instanci třídy, můžeme na ní neomezeně iterovat. O vytvoření nového iterátoru v rámci každé iterace se třída postará sama. Potřebujeme-li pracovat přímo s iterátorem a volat metodu `__next__()`, nic nám nebrání si příslušný iterátor vytvořit. Jen **musíme pamatovat na jeho omezení v podobě vyčerpatelnosti.**

Generátory

Generátory jsou **speciální iterátory.** Jejich velkou výhodou je daleko snazší a přímočařejší implementace. Vytvořit generátor lze pomocí **generátorové funkce** nebo **generátorového výrazu**.

Generátorová funkce

Generátorová funkce vrací objekt typu `generator`. Je to každá funkce, která obsahuje alespoň jeden příkaz `yield`:

```
def generatorova_funkce():
    yield "Já"
    yield "mám"
    yield "hlad"
muj_generator = generatorova_funkce()
```

Samotný generátor vytvoříme tak, že tuto funkci zavoláme. Na rozdíl od běžné funkce se kód umístěný v těle generátorové funkce neprovede, pouze se vrátí objekt generátoru. Nyní máme k dispozici generátor, který se chová stejně jako nám již dobře známý *iterátor*. Pokud budeme volat funkci `next()`, získáme jednotlivé prvky:

```
print(next(muj_generator))
```

Po zavolání funkce `next()` se vykonají příkazy v těle generátorové funkce až po první `yield`, tam se zastaví a příslušná hodnota se vrátí. Při dalším zavolání se pokračuje od tohoto místa, dokud nenarazí na další `yield` atd. Jakmile narazí na příkaz `return`, anebo již nejsou další příkazy, ukončí se.

Generátorové výrazy

Jednoduché generátory mohou být vytvořeny elegantním způsobem v podobě generátorového výrazu. Syntaxe je téměř totožná jako při tvorbě seznamové komprehence. Rozdíl spočívá jen v tom, že **místo hranatých závorek se použijí kulaté**. Použitím generátorového výrazu se nám kód ještě lehce zredukuje a **zvýší se jeho čitelnost**:

```
class KolekceNahodnychCisel:
    def __init__(self, pocet_cisel, spodni_hranice, horni_hranice):
        self.pocet_cisel = pocet_cisel
        self.spodni_hranice = spodni_hranice
        self.horni_hranice = horni_hranice
    def __iter__(self):
        return (randint(self.spodni_hranice, self.horni_hranice) for _ in range(self.pocet_cisel))
```

Využití iterátorů v praxi

Slabou stránkou iterátorů je jejich jednorázové použití. Iterátory nicméně nabízejí velmi **efektivní práci se zdroji**. Pomocí iterátoru lze realizovat tzv. **odložené vyhodnocování** (lazy evaluation). Díky této strategii lze vyhodnotit určitý výraz až v okamžiku, kdy je jeho hodnota skutečně potřeba.

Představme si, že potřebujeme zpracovat nějaký velký soubor a získat z něj určité informace. Nejjednodušším řešením by bylo načíst obsah celého souboru do seznamu pomocí metody `readlines()` a následně ho zpracovat. Nicméně zdrojový soubor by v závislosti na zvoleném časovém období mohl dosahovat i extrémních velikostí. Výše zvoleným způsobem bychom celý soubor umístili do paměti počítače. My ale nepotřebujeme pracovat s kompletním obsahem souboru naráz. Jelikož funkce `open()` vrací objekt `IOWrapper`, což je iterátor můžeme obsah načíta iterovatelně. Tzn. že načítáme obsah souboru postupně řádek po řádku, tudíž v jeden okamžik využíváme **pouze tolik paměti** počítače, kolik zabírají data **jedné transakce**.
Výhodou tohoto přístupu je možnost zpracování souboru libovolné velikosti.

Seznamové komprehence

Seznamová komprehence umožňuje **stručný zápis tvorby seznamu** z iterovatelného objektu za pomoci transformace, iterace a případně filtru.

Příklad syntaxe je následující:

```
novy_seznam = [prvek * 2 for prvek in jiny_seznam if prvek < 5]
```

Jednotlivé fáze tedy jsou:

- **transformace** - výraz, který definuje změnu původního prvku (zde `prvek * 2`)
- **iterace** - transformace se postupně provádí na jednotlivé prvky iterovatelného objektu (zde `for prvek in jiny_seznam`)
- **filtr** (nepovinný) - pokud prvek z původního iterovatelného objektu splňuje danou podmínku, přidá se do nového seznamu, v opačném případě se ignoruje (zde `if prvek < 5`)

Vytvořit seznam tímto způsobem se doporučuje **pouze v jednoduchých případech**. Čím složitější seznamovou komprehenci vytvoříme, tím méně čitelná bude a může to být kontraproduktivní.

Množinové komprehence

Seznamovou komprehenci lze aplikovat i na množiny. Postup je téměř identický, jediný rozdíl je v použitých závorkách. **U množin se místo hranatých používají složené**:

```
zdrojova_mnozina = (1, 2, 3, 4, 5)
nova_mnozina_generovana = (prvek * 2 for prvek in zdrojova_mnozina if prvek < 5)
nova_mnozina = set(nova_mnozina_generovana)
print(nova_mnozina)
```

Slovníkové komprehence

Komprehence lze použít i na tvorbu slovníků. Závorky se používají stejné jako u množin. Rozdíl je ten, že pro každý prvek se místo jednoho výrazu zapisují dva výrazy oddělené dvojtečkou. První představuje ***klíč***, druhý ***hodnotu***. Následující kód ukazuje elegantní způsob, jak vytvořit kopii slovníku, kde dojde k prohození klíčů a hodnot. Nutno dodat, že hodnoty zdrojového slovníku musí být neměnného typu:

```
slovnik = {"Homer": "Simpson", "Barney": "Gumble"}
inverzni_slovnik = {prijmeni: jmeno for jmeno, prijmeni in slovnik.items()}
print(inverzni_slovnik)
```

N-ticové komprehence v Pythonu neexistují

Při hlubším zamyšlení je zřejmé, že zde chybí možnost n-ticových komprehencí. Pro ně by se nabízel zápis pomocí kulatých závorek. Toto omezení se však dá obejít vytvořením seznamové komprehence a použitím vestavěné funkce `tuple()`. Kód pro n-tici sudých čísel do 20 pak vypadá takto:

```
ntice = tuple([x for x in range(2, 21, 2)])
```

Lambda výrazy

Lambda výraz **vrací funkční objekt** (je to tedy jiný způsob tvorby funkce). Vytvoříme ho klíčovým slovem `lambda`. Následují nepovinné parametry oddělené čárkou a poté dvojtečka, za kterou se píše příslušný výraz. Ten můžeme přirovnat k tělu klasické funkce.

Oproti klasickým funkcím se liší následovně:

- *Výsledná funkce nemá jméno (proto se někdy nazývá anonymní).*
- *K definování takovéto anonymní funkce nám musí stačit jeden řádek (přesněji jeden výraz).*

Druhého bodu se dá využít např. u funkcí vyššího řádu, které přijímají jako argument jinou funkci.

Vestavěné funkce pro práci s iterovatelnými objekty:

enumerate()

Funkce vytvoří dvojice, kde **první položkou je index a druhou položkou je příslušný prvek** zadaného iterovatelného objektu:

```
rocni_obdobi = ["jaro", "leto", "podzim", "zima"]
renum_rocni_obdobi = enumerate(rocni_obdobi, start=1)
for index, obdobi in enum_rocni_obdobi:
    print(index, obdobi)
```

zip()

Funkce vytvoří n-tice ze zadaných iterovatelných objektů tak, že každá n-tice obsahuje po jednom prvku z každého iterovatelného objektu:

```
filmy = ['Terminator', 'Rambo', 'Poslední skaut', 'Titanic']
herci = ['Schwarzenegger', 'Stallone', 'Willis', 'Di Caprio']
roky = [1984, 1982, 1991]
fhr = zip(filmy, herci, roky)
for i in fhr:
    print(i)
```

map()

Funkce přijímá jako svůj argument jinou funkci, kterou aplikuje na iterovatelné objekty tak, že za parametry této funkce dosadí paralelně jednotlivé prvky příslušných iterovatelných objektů. To znamená, že počet parametrů funkce se musí rovnat počtu iterovatelných objektů.

Příklad jednoparametrické funkce se seznamem:

```
ceny_v_kc = [25, 1500, 10_000, 500_000]
def format_cen(cena):
    return str(cena) + ' Kc'
ceny = map(format_cen, ceny_v_kc)
for c in ceny:
    print(c)
```

Příklad dvouparametrické anonymní funkce se dvěma n-ticemi různé délky:

```
for soucet in map(lambda x, y: x + y, (1, 2, 3, 4), (100, 200, 300)):
    print(soucet)
```

filter()

Funkce přijímá jako svůj argument jinou funkci, kterou aplikuje na iterovatelný objekt tak, že za parametr funkce postupně dosazuje jednotlivé prvky tohoto objektu. Pokud funkce s daným argumentem vrátí `False`, prvek je zahozen:

```
cisla = [2, 5, 13, 16, 50, 55]
licha_cisla = filter(lambda x: x % 2, cisla)
print(list(licha_cisla))
```

Kombinace funkcí `map()` a `filter()`

V praxi se často setkáme s obraty, kde se kombinují výhody obou zmíněných funkcí:

```
ceny_v_kc = [25, 1500, 10_000, 500_000]
ceny_do_10_000 = map(lambda cena: str(cena) + "Kc", filter(lambda cena: cena < 10_000, ceny_v_kc))
print(list(ceny_do_10_000))
```

Stejného účinku nicméně docílíme pomocí **seznamové komprehence** následovně:

```
print([str(cena) + "Kc" for cena in ceny_v_kc if cena < 10_000])
```

V případě funkce `map()`, která pracuje s více iterovatelnými objekty, si musíme ještě vypomoci funkcí `zip()`:

```
print(list(map(lambda x, y: x + y, (1, 2, 3, 4), (100, 200, 300))))
print([x + y for x, y in zip((1, 2, 3, 4), (100, 200, 300))])
```

Použití seznamové komprehence místo kombinování funkcí se obecně považuje za čistější řešení. Kód je přímočařejší a lépe čitelný.

ChainMap, NamedTuple a DeQue v Pythonu

jsou 3 zástupci kolekcí v Pythonu, pro práci s iterovatelnými objekty.

ChainMap

`ChainMap` je datový typ, který nám **umožňuje sdružovat více slovníků do jednoho logického celku.** Jinými slovy nám `ChainMap` umožňuje jednoduše a efektivně **pracovat s více slovníky jako s jedním.**

`ChainMap` se často používá v případě, když chceme hledat hodnotu pro klíč ve **více slovnících.** Velkou výhodou `ChainMap` je, že nám umožňuje udržovat všechny slovníky v **jednom objektu.** Hledání ve sdružených slovnících bude probíhat postupně v pořadí, v jakém byly uvedeny při vytvoření instance, dokud se pro klíč nenajde odpovídající hodnota.

Příklad použití `ChainMap`

Použití `ChainMap` v Pythonu je velmi jednoduché. Prvním krokem je importovat modul `collections`. Dále si nadefinujeme slovníky `slovnik1` a `slovnik2` a poté vytvoříme proměnnou `mapa`, která bude obsahovat instanci třídy `ChainMap`:

```
from collections import ChainMap
slovnik1 = {'a':1, 'b':2}
slovnik2 = {'c':3, 'b':4}
mapa = ChainMap(slovnik1, slovnik2)
```

Proměnná `mapa` se chová jako slovník, ve kterém jsou uloženy hodnoty ze `slovnik1` a `slovnik2`. Následný výpis `print(mapa['b'])` vrací hodnotu klíče `b` z prvního slovníku, ve kterém je k dispozici, tj. ze `slovnik1`.

Pokud přidáme do proměnné `mapa` nový klíč `c` s hodnotou `5`. Tyto změny se projeví v proměnné `mapa`, **ale ne v původních slovnících.**

Když ale modifikujeme jeden ze slovníků a opět si vypíšeme obsah proměnné, vidíme, že hodnota klíče se v proměnné `mapa` se změnila, protože se změnila i v podkladovém slovníku.

Metody pro práci s `ChainMap`:

Třída `ChainMap` obsahuje mimo jiné následující metody:

maps - vrátí novou instance ChainMap, která přidá zadaný slovník (nebo slovníky) na konec seznamu slovníků,

- *new_child() - přidá nový slovník na první pozici v seznamu,*
- *get(key[, default]) - vrátí hodnotu pro daný klíč. Pokud hodnota není nalezena, vrátí hodnotu default (pokud je zadána), jinak vyvolá výjimku KeyError,*
- *keys() - vrátí seznam všech klíčů,*
- *values() - vrátí seznam všech hodnot,*
- *items() - vrátí seznam všech položek ve formátu (klíč, hodnota).*

Výše uvedené si tedy shrneme. Platí, že `ChainMap` je **aktualizovatelný** pohled nad více slovníky. Hodnoty v příslušné **instanci** se mění v závislosti na změnách v jednotlivých podkladových slovnících.

NamedTuple

Třída `NamedTuple` je speciální případ neseřazeného datového typu, který kombinuje výhody tuple a slovníku. Stejně jako tuple má také pevnou délku a její instance nelze modifikovat. Zásadní rozdíl ale spočívá v tom, že jednotlivé položky mohou být přístupné pomocí jména (stejně jako v případě slovníku). Výhodou `NamedTuple` tedy je, že umožňuje jasnější a čitelnější kód, protože jména položek jsou přímo součástí kódu. Díky pevné délce je také efektivnější než použití slovníku.

Příklad použití `NamedTuple`

`NamedTuple` se vytváří pomocí funkce `namedtuple()`. Nejprve si třídu `NamedTuple` naimportujeme z modulu `collections`. Poté pomocí funkce `namedtuple()` vytvoříme třídu `Osoba`, která rozšiřuje `NamedTuple` o položky `jmeno`, `prijmeni` a `vek`. Následně vytvoříme proměnnou `student`, v níž bude instance třídy `Osoba`:

```
from collections import namedtuple
Osoba = namedtuple("Osoba", ["jmeno", "prijmeni", "vek"])
student = Osoba("Jan", "Novák", 26)
```

K položkám `NamedTuple` pak budeme přistupovat pomocí jména položky jako atributu instance, jako by šlo o obyčejné atributy třídy:

```
print("Jméno a příjmení studenta: "+ student.jmeno + " " + student.prijmeni)
print("Věk studenta:", student.vek)
```

Následujícím způsobem pak vypíšeme celou instanci `NamedTuple` z proměnné `turista`:

```
from collections import namedtuple
Turista = namedtuple('Turista', ['jmeno', 'vek', 'zeme'])
turista = Turista('Jan', 26, 'Norsko')
print(turista)
```

Metody pro práci s `NamedTuple`

Třída `NamedTuple` obsahuje mimo jiné následující metody:

- ***make()*** *- umožňuje vytvořit instanci NamedTuple z iterovatelných objektů jako jsou seznamy nebo sekvence,*
- ***asdict()*** *- slouží k převodu na slovník,*
- ***replace()*** *- vytvoří novou instanci NamedTuple se stejným názvem, ale s novými hodnotami. Původní instance NamedTuple zůstává nezměněná,*
- ***fields*** *- slouží k získání seznamu názvů všech položek v instanci NamedTuple.*

V praxi se `NamedTuple` často používá jako výkonný a jednoduchý způsob, jak organizovat a ukládat data, která spojuje nějaký kontext, ale nejsou k nim třeba žádné další metody nebo funkce.

DeQue

Třída `Deque` (zkratka pro "Double-Ended Queue") představuje specifický typ fronty, který je implementovaný jako dvoustranně otevřený seznam. To umožňuje efektivně přidávat a odebírat prvky z obou stran. V praxi se `DeQue` využívá například při řešení úloh s BFS (Breadth-First Search) a DFS (Depth-First Search), což jsou dva typy algoritmů pro hledání nejkratší cesty, hledání všech cest, hledání komponent grafu apod.

Příklad použití DeQue

Použití `DeQue` v Pythonu je velmi jednoduché. Prvním naším krokem bude importovat modul `collections`. Poté vytvoříme instanci třídy `DeQue` a vypíšeme si ji:

```
from collections import deque
deq = deque([1,2,3,4])
print(deq)
```

Vyzkoušejme si teď, jak funguje naše tvrzení o otevřených koncích seznamu. Využijeme k tomu metody `append()`, `appendleft()`, `pop()` a `popleft()`:

```
deq.append(5)
print(deq)
deq.appendleft(0)
print(deq)
deq.pop()
print(deq)
deq.popleft()
print(deq)
```

V konzoli vidíme výsledek:

```
deque([1, 2, 3, 4])
deque([1, 2, 3, 4, 5])
deque([0, 1, 2, 3, 4, 5])
deque([0, 1, 2, 3, 4])
deque([1, 2, 3, 4])
```

Podívejme se teď blíže na to, co kód vykonal:

- *nejprve jsme vytvořili frontu deq s prvky 1, 2, 3 a 4,*
- *pomocí metody append() jsme přidali prvek 5 na konec fronty,*
- *pomocí metody appendleft() jsme přidali prvek 0 na začátek fronty,*
- *pomocí metody pop() jsme odebrali poslední prvek z fronty,*
- *pomocí metody popleft() jsme odebrali první prvek z fronty.*

Metody pro práci s DeQue

Třída `DeQue` obsahuje kromě už uvedených také následující často využívané metody:

- ***clear()*** *- smaže všechny prvky,*
- ***count()*** *- vrátí počet výskytů určitého prvku,*
- ***extend()*** *- přidá více prvků na konec,*
- ***extendleft()*** *- přidá více prvků na začátek,*
- ***remove()*** *- odstraní prvek s určitou hodnotou,*
- ***reverse()*** *- otočí pořadí prvků.*

Výhodou `DeQue` oproti prostému seznamu je rychlost, protože přidávání a odebírání prvků z obou stran je v ní implementováno efektivně. Často se používá jako zásobník nebo fronta, do které lze prvky na jednu stranu přidávat a z druhé strany je odebírat.

Counter, OrderedDict a defaultdict v Pythonu

třídy `Counter`, `OrderedDict` a `defaultdict`, jsou potomky vestavěného slovníku (třídy `dict`). Jejich techniky lze plnohodnotně realizovat klasickým slovníkem. Nicméně, pokud si chceme ulehčit práci a zvýšit čitelnost kódu, tyto specializované třídy nám určitě pomohou.

Counter

Třída `Counter` je speciální slovník, kde **jednotlivé prvky jsou uloženy jako klíče a četnost těchto prvků jako příslušné hodnoty**. Jinými slovy klíč reprezentuje daný prvek a hodnota reprezentuje počet výskytů tohoto prvku. Použitím třídy `Counter` lze tedy snadno a efektivně získat statistiku četnosti prvků v seznamu, řetězci nebo jiném iterovatelném objektu.
(V jiných programovacích jazycích se často používá pojem `Multiset`. `Counter` je pythonovská implementace tohoto datového typu.)

Příklady použití `Counter`

Ukažme si tři nejběžnější způsoby, jak vytvořit instanci třídy `Counter`:

```
from collections import Counter
c1 = Counter("abbccc")
c2 = Counter(['a', 'b', 'b', 'c', 'c', 'c'])
c3 = Counter(a=1, b=2, c=3)
print(c1, c2, c3)
```

Ve výstupu vidíme, že všechny tři způsoby dávají stejný výsledek:

```
Counter({'c': 3, 'b': 2, 'a': 1}) Counter({'c': 3, 'b': 2, 'a': 1}) Counter({'c': 3, 'b': 2, 'a': 1})
```

Instance třídy `Counter` **po dotazu na chybějící klíč vracejí nulu**. Na rozdíl od instancí třídy `dict`, kde je vyvolána `KeyError` výjimka.

Metody pro práci s `Counter`:

update()

Metoda umožňuje aktualizovat počet prvků v instanci třídy `Counter` pomocí jiného objektu. Tento objekt může být buď typu `Counter`, nebo iterovatelný objekt (například seznam, tuple, řetězec nebo slovník). Na rozdíl od stejnojmenné metody předka (`dict`) se **ve chvíli updatu již existujícího klíče jeho hodnota nepřepíše, nýbrž přičte**:

```
c1 = Counter("abbccc")
c1.update("abeceda")
print(c1)
```

Výstupem v konzoli bude:

```
Counter({'c': 4, 'a': 3, 'b': 3, 'e': 2, 'd': 1})
```

most_common(N)

Metoda vrátí **N prvků s nejvyšší četností**. Pokud parametr N není uveden, vrátí seznam všech prvků:

```
print(c1.most_common(3))
```

Výstupem v konzoli bude:

```
[('c', 4), ('a', 3), ('b', 3)]
```

elements()

Tato metoda vrátí iterátor jednotlivých prvků, kde **každý prvek se opakuje přesně tolikrát, kolikrát je jeho hodnota** (četnost). Abychom na výstupu viděli nějaký smysluplný výsledek, převedeme si ho na seznam:

```
print(list(c1.elements()))
```

Výstupem v konzoli bude:

```
['a', 'a', 'a', 'b', 'b', 'b', 'c', 'c', 'c', 'c', 'e', 'e', 'd']
```

total()

Metoda `total()` vrátí **součet četností všech prvků** dohromady:

```
print(c1.total())
```

Výstupem v konzoli bude:

```
13
```

OrderedDict

Jak už název napovídá, klíčovou vlastností třídy `OrderedDict` je to, že **prvky jsou uloženy v takovém pořadí, v jakém byly postupně do slovníku přidávány.** S příchodem Pythonu 3.6 bylo řazení prvků implementováno i do klasického slovníku a mohlo by se zdát, že použití `OrderedDict` již nedává smysl. Zde jsou argumenty, proč `OrderedDict` stále svůj smysl má:

- ***čitelnost kódu**: použitím třídy `OrderedDict` dáváme jasně najevo, že pořadí prvků je důležité. To se hodí ve chvíli, kdy náš kód čte někdo jiný a nebo pokud se k němu vracíme my sami po delší době,*
- ***zpětná kompatibilita**: pokud chceme garantovat přenositelnost mezi libovolnými verzemi Pythonu, musíme sáhnout po `OrderedDict`. Pak máme jistotu, že náš program se bude chovat stejně i na systémech s verzemi nižšími jak Python 3.6,*
- ***význam operátoru ==**: porovnání dvou instancí třídy `OrderedDict` pomocí operátoru == vrací `True` když obě instance obsahují nejen shodné prvky, ale zároveň i **ve stejném pořadí**! U klasického slovníku se pořadí ignoruje.*

Metody pro práci s `OrderedDict`:

Vytvoření instance třídy `OrderedDict`:

```
from collections import OrderedDict
usporadany_slovnik = OrderedDict(Sly="Rambo", Arnie="Terminator", Leo="Titanic")
```

dir()

Vestavěná funkce `dir()` vrací seznam atributů objektu. V kombinaci s množinovým rozdílem můžeme zjistit, které z nich jsou jedinečné pro `OrderedDict`.

```
atributy_dict = dir(dict())
atributy_OrderedDict = dir(OrderedDict())
print(set(atributy_OrderedDict) - set(atributy_dict))
```

Výstupem v konzoli bude:

```
{'move_to_end', '__dict__'}
```

move_to_end(last=True)

Metoda `move_to_end()` přemístí existující prvek na konec slovníku (pokud `last=True` a nebo argument není zadán). Pokud `last=False`, existující prvek je přemístěn na **začátek slovníku**. V případě, že daný prvek neexistuje, je vyvolána `KeyError` výjimka:

```
usporadany_slovnik.move_to_end("Sly")
print(usporadany_slovnik)
usporadany_slovnik.move_to_end("Leo", last=False)
print(usporadany_slovnik)
```

Výstupem v konzoli bude:

```
OrderedDict([('Arnie', 'Terminator'), ('Leo', 'Titanic'), ('Sly', 'Rambo')])
OrderedDict([('Leo', 'Titanic'), ('Arnie', 'Terminator'), ('Sly', 'Rambo')])
```

__dict__

Díky tomuto atributu můžeme **dynamicky přidávat vlastní metody.** Příklad použití vypadá takto:

- *nejprve si pomocí `lambda` výrazu vytvoříme vlastní funkci, která vrátí seřazený slovník dle klíče,*
- *následně ji přidáme objektu `usporadany_slovnik` jako metodu s názvem `serad_dle_klice()`.*

Obojí lze uskutečnit pomocí jednořádkového zápisu:

```
usporadany_slovnik.serad_dle_klice = lambda: sorted(usporadany_slovnik.keys())
```

Nyní můžeme pomocí této nové metody **vypsat prvky slovníku abecedně dle jména herce** (nikoliv dle pořadí, v jakém byly do slovníku postupně přidávány):

```
for herec in usporadany_slovnik.serad_dle_klice():
    print(herec + " --> " + usporadany_slovnik[herec], end=", ")
```

Výstupem v konzoli bude:

```
Arnie --> Terminator, Leo --> Titanic, Sly --> Rambo
```

defaultdict

Poslední třídou, se kterou se v této lekci seznámíme, je `defaultdict`. Třída pracuje zajímavým způsobem. Poskytuje slovník s výchozí hodnotou pro klíče, které ještě neexistují v daném slovníku. Jakmile je tedy přijat požadavek na hodnotu chybějícího prvku (neexistujícího klíče), na místo výjimky `KeyError` která by vznikla v případě klasického slovníku, se provedou následující dva kroky:

1) Chybějící klíč se přidá do slovníku.
*2) Zavolá se tzv. **tovární metoda** a její vrácený objekt se nastaví jako hodnota tohoto klíče.*

Tovární metodou může být libovolný volatelný objekt včetně funkcí a tříd. Jejím účelem je vygenerování výchozí hodnoty pro chybějící klíč. (Tovární metoda musí být bezparametrická.)

Často se **za tovární metody dosazují některé vestavěné funkce**. Například `int()` po zavolání vrací nulu, `str()` prázdný řetězec a `list()` prázdný seznam. Pojďme si to ukázat na příkladu. K dispozici máme následující seznam zaměstnanců:

```
from collections import defaultdict
zamestnanci = [
    ("Jan Novák", "Výroba"),
    ("Michaela Modrá", "Obchod"),
    ("Pavlína Peterková", "Prodej"),
    ("Karel Nový", "Výroba"),
    ("Petr Blažek", "Obchod")]
```

Dále si vytvoříme instanci třídy `defaultdict`. Jako typ pro tovární metodu využijeme seznam a dosadíme tedy `list` (bez závorek, zatím ji nevoláme):

```
oddeleni = defaultdict(list)
```

Nyní můžeme iterovat na původním seznamu a vytvořit slovník jednotlivých oddělení. Metoda `append()` nevyhodí výjimku ani ve chvíli dotazu na neexistující klíč. To proto, že se nejprve automaticky vytvoří prázdný seznam:

```
for zamestnanec, pozice in zamestnanci:
        oddeleni[pozice].append(zamestnanec)
print(oddeleni)
```

Ve výstupu vidíme:

```
defaultdict(class 'list', {'Výroba': ['Jan Novák', 'Karel Nový'], 'Obchod': ['Michaela Modrá', 'Petr Blažek'], 'Prodej': ['Pavlína Peterková']})
```