Python pro matematickou (pato)fyziologii

Michal Šitina

22. února 2022

Ústav patologické fyziologie
Lékařská fakulta
Masarykova univerzita Brno

Obsah

1 Úvod

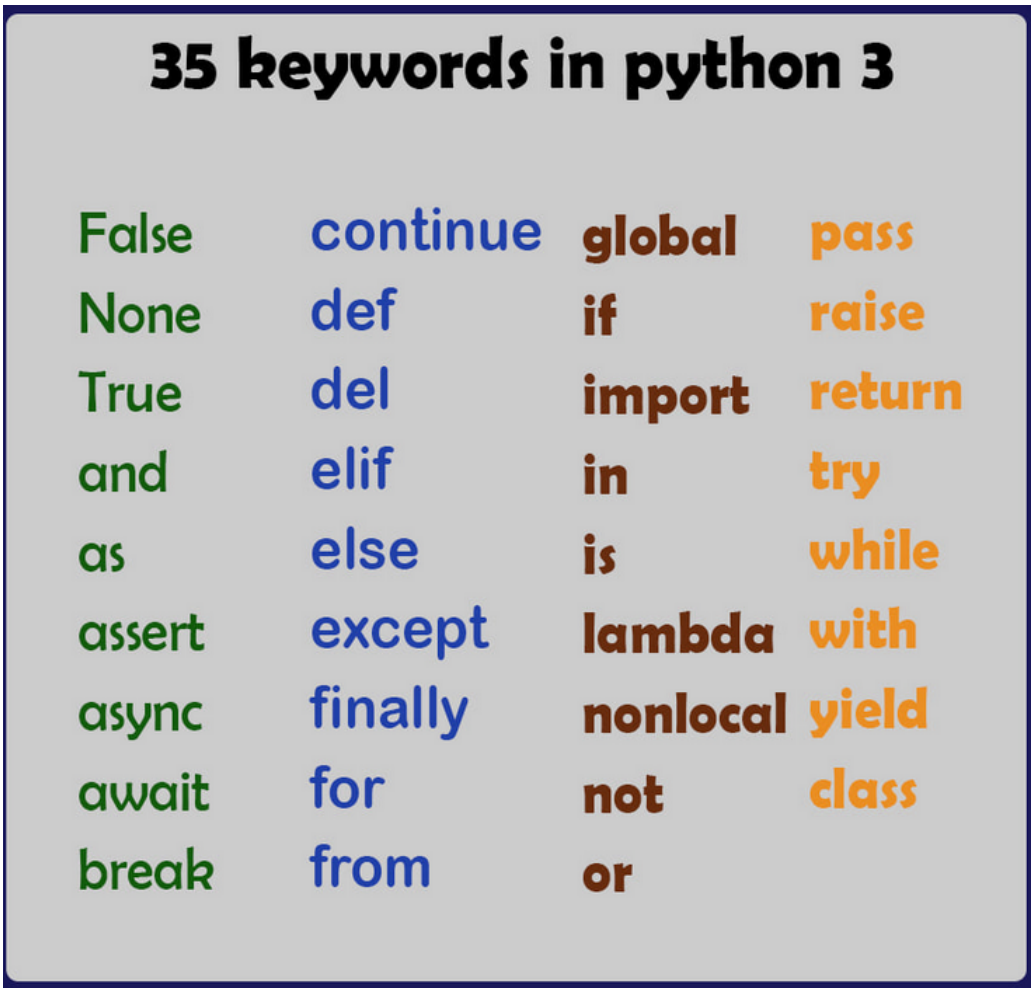
Python je v současnosti jedním z nejpopulárnějších programovacích jazyků. Je široce použitelný v mnoha oblastech - vědecké výpočty, analýza dat, machine learning, programování aplikací, práce s databázemi, webové aplikace a další. Jedná se o tzv. ***objektově orientovaný*** jazyk, ale lze v něm programovat i „jednoduše“ neobjektově, procedurálně. Python je ***interpretovaný*** (synonymum skriptovací) jazyk. Programátor píše obyčejný textový soubor, většinou s příponou .py, obvykle označovaný jako ***skript***. Tento soubou se předloží programu s názvem `python`. Program `python` je tzv. interpret, tedy program, který rozumí textu ve skriptu, čte jej řádek po řádku a převádí jednotlivé příkazy do strojových příkazů (instrukcí) pro procesor počítače, který je vykonává. Pythonský skript `soubor.py` bychom tedy spustili např. z příkazové řádky Windows pomocí příkazu `C:\python soubor.py`. Obvykle však používáme nějaký program usnadňující psaní skriptu, obecně označovaný jako IDE (Integrated Development Environment). IDE pro Python je např. Jupyter Notebook nebo Spyder, které jsou součástí distribuce [1] Anaconda, nebo PyCharm či IDLE. Pak lze většinou kliknout na ikonu „Run“ a program přímo spustit.

Jistá slova, tzv. ***klíčová slova***, např. print, for, in, range... mají v Pythonu vlastní speciální význam a funkci a nelze je použít např. jako název proměnné. S těmito asi 35 klíčovými slovy lze naprogramovat vpodstatě cokoli. Zvládnout programování v základním Pythonu tedy vlastně znamená naučit se syntaxi (jak přesně se příkazy zapisují, aby jim interpret rozuměl) a sémantiku (co přesně znamenají) těchto klíčových slov. Krom toho ale existuje obrovské balíků, též označovaných jako ***moduly*** nebo ***packages***, které usnadňují a rozšiřují schopnosti Pythonu. Byly však naprogramovány v základním Pythonu. Protože Python patří mezi tzv. ***objektově orientované jazyky***, často se hovoří o objektech.

Zápis programu v Pythonu je jednoduchý, místo např. středníků používaných v jiných profgramovacích jazycích se pro účely seskupování textu programu („co k sobě patří“) používá odsazení (tzv. indentace). Indentaci nelze vynechat či pozměnit, změnila by se logika programu. Např. v následujícím programu je první print() součástí for cyklu (vytiskne se při každém běhu cyklu), druhá print() již není součástí (vytiskne se pouze jednou).

```
>>> for i in range(4): # komentáře následují za #
                       # slouží k lepší orientaci programátora
                       # a čtenáře, interpret je přeskočí
>>>     print('je součástí')
>>> print('není součástí')

je součástí
je součástí
je součástí
je součástí
není součástí
```

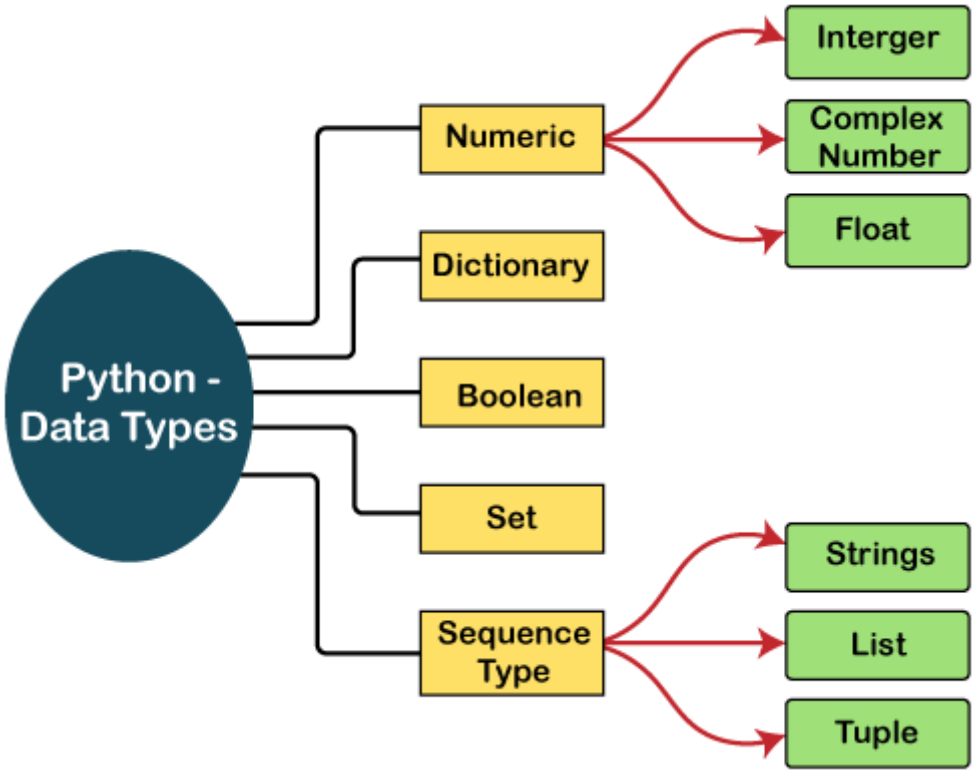
[1] Python je programovací jazyk. Pro programování v Pythonu vlastně ani nepotřebujete počítač, program můžete psát třeba na papír. Rozhodující je interpret, tedy program, který textu rozumí. Interpret lze stáhnout a nainstalovat, text programu napsat v textovém editoru, najít a stáhnout si všechny potřebné doplňkové moduly apod. Je to postup možný, šetří paměť počítače, ale je poněkud pracný. Jednodušší je nainstalavat Python společně s řadou doplňkových modulů, IDE apod., jako celek, v podobě takzvané distribuce Pythonu. Jednotlivé distribuce se liší právě těmito doplňkovými produkty, vlastní pythonský interpret ve všech distribucích stejný (snad až na drobné odlišnosti). Nejznámější distribucí je Anaconda, případně její stručnější verze Miniconda.



Obrázek 1.1: Klíčová slova Pythonu

2 Datové typy

Každá proměnná v Pythonu je určitého datového typu, například číslo nebo text. Proměnným určitého typu odpovídají jisté operace. Lze například sčítat dvě čísla, ale nelze sčítat číslo s písmenem, jinak Python hlasí chybu. Narozdíl od např. Javy nebo C++ není třeba v Pythonu proměnné deklarovat, t.j. předem říci, jaký je datový typ proměnné. Pokud napíšeme `x=3`, Python sám pozná, že proměnná x je typu int (celé číslo), proměnnou x vytvoří a přiřadí ji hodnotu 3.



Obrázek 2.1: Datové typy Pythonu

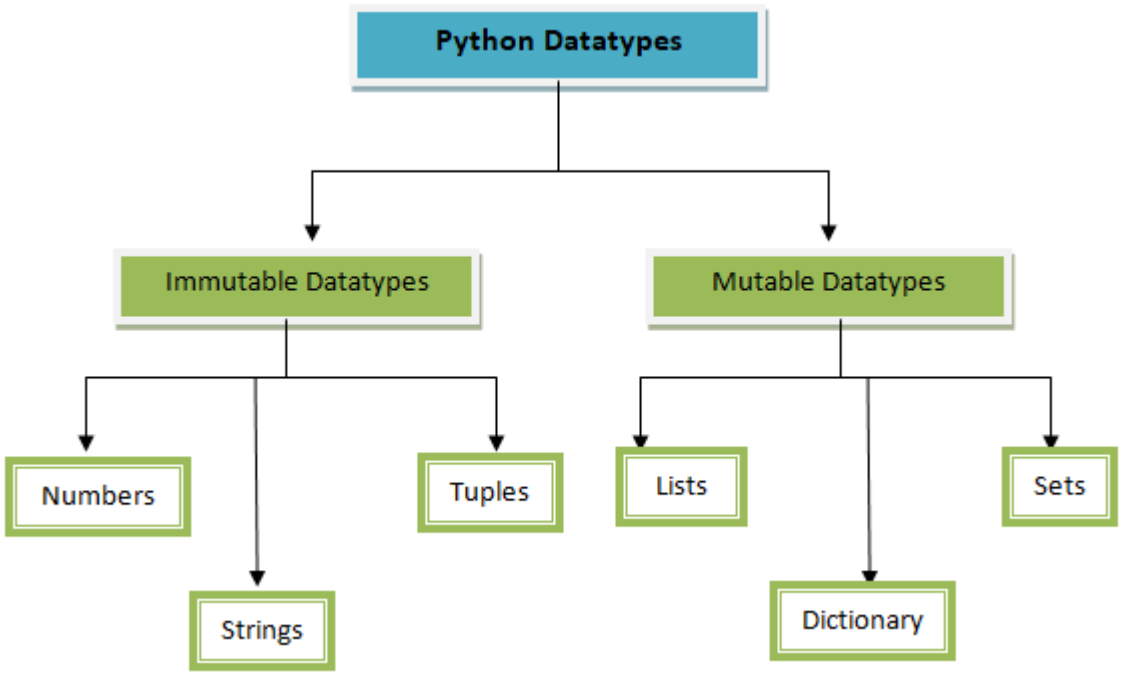
2.1 Čísla

Čísla lze zadávat v různých soustavách.

0o26	osmičková soustava
19	desítková soustava
0x1A	šestnáctková soustava
0b1101001	dvojková soustava

Mohou být datového typu ***int*** (celá čísla), ***float*** (desetinná čísla) a ***complex*** (komplexní čísla).

12	int
19.15	float
4+7j	complex



Obrázek 2.2: Měnitelnost proměnných dle datových typů

Pomocí `type` lze zjistit datový typ.

```
>>>type(19.15)
float
```

Operace s čísly:

```
3/2 == 3.0/2 == 3/2.0   # výsledek dělení je vždy float
3 // 2                  # celočíselné dělení
3 mod 2                 # zbytek po dělení (operace zvaná modulo)
2**3                    # 2 na 3.
```

2.2 Sekvence

Sekvence je seskupení hodnot za sebou vnímaných a označených jako celek. Podobné struktury jsou v různých jazycích označovány různě, např. vektor, pole, array apod. Jejich konkrétní vlastnosti se mohou lišit. Jako sekvence vystupují v Pythonu dva datové typy - ***list*** a ***tuple***, česky přeloženo jako seznam a n-tice, případně ještě textový řetezec, ***string***.

Prvky sekvence mají pořadí (na rozdíl od datových typů ***set*** a ***dictionary***). Číslování začíná od 0, proto má např. 7. prvek index 6. Jednotlivé prvky sekvence mohou být jakéhokoli datového typu, i každý prvek jiného typu. Rozdíl mezi seznamem a n-ticí je, že

list	je měnitelný (mutable), po vytvoření lze měnit jednotlivé hodnoty
tuple	je neměnitelná (immutable), po vytvoření již nelze měnit hodnoty

```
>>> x = [1,5.2, abc, [1,3,4]]  # vytvoří list x
>>> y = (1,5.2, abc, [1,3,4])  # vytvoří tuple y
>>> print(type(x))
>>> print(type(y))
```

```
<class 'list'>
<class 'tuple'>
```

Speciální sekvencí je řetězen znaků - ***string***. Tvoří se pomocí "..."nebo '...'. Podrobněji je problematika rozebrána v kapitole 6.

```
>>> x = abc
>>> type(x)  # zjistí datový typ
str
```

Základni operace se sekvencemi

x in s	Je prvek x v sekvenci s?
x not in s	Není prvek x v sekvenci s?
s + t	spojení sekvencí s a t za sebou, tzv. konkatenace
s * n	konkatenace n sekvencí s za sebou
s[i]	i-tý prvek sekvence s
s[i:j]	prvky sekvence s indexy i až j
len(s)	délka sekvence
min(s)	minimum sekvence
max(s)	maximum sekvence
s.count(x)	kolikrát je x v sekvenci s
s.index(x)	index prvního výskytu x v sekvenci s

```
>>> y.count(1)
1
```

```
>>>     l = [[1,2,3], 4.5, dfdfd]
>>>     l[-1]  # posledni element listu l
'dfdfd'
```

```
>>>     [5] + [7] + [2,3,9] # konkatenace
[5, 7, 2, 3, 9]
```

Speciální operace s listy

Plynou z měnitelnosti listu.

```
>>> s = [2,4,6,8]
>>> s.append(10) # přidá prvek 10
[2, 4, 6, 8, 10]
```

```
>>> s.extend([10,12]) # přidá sekvenci
[2, 4, 6, 8, 10, 12]
```

```
>>> del s[1] # smaže 2. element (s indexem 1)
[2, 6, 8, 10, 12]
```

```
>>> s.insert(1, 7) # vloží 7ku na 2. místo (s indexem 1)
[2, 7, 4, 6, 8]
```

```
>>> posledni = s.pop() # poslední prvek vrátí a odstraní
>>> posledni
12
```

```
>>> s
[2, 7, 4, 6]
```

```
>>> s.remove(7) # odstraní 7ku
[2, 4, 6]
```

```
>>> s.sort() # seřadí s podle velikosti
```

2.3 Množiny

Datový typ ***set*** odpovídá představě množiny známé z matematiky. Prvky jsou každý pouze jednou, nemají pořadí (nelze je indexovat), nejsou uspořádané. Jednotlivé prvky však mohou být různých datových typů. Jako v matematice se množiny tvoří pomocí .

```
a = {1,2,3,'gk'}    # definice množiny
b = set([1,2,3])    # vytvoří množinu ze sekvence
```

Operace s množinami

s1.union(s2)	sjednocení
s1.intersection(s2)	průnik
s1.difference(s2)	rozdíl
s1.symmetric_difference(s2)	symetrický rozdíl
s1.issubset(s2)	Je podmnožina?
s1.isupperset(s2)	Je nadmnožina?
x in s	Obsahuje množina s prvek x?
x not in s	Neobsahuje množina s prvek x?

2.4 Dictionary

Datový typ ***dictionary*** odpovídá představě slovníku. Jedná se o množinu uspořádaných dvojic klíč:hodnota (key:value páry). Jako v množině se každý klíč vyskytuje pouze jednou, páry nemají pořadí (nelze je indexovat), nejsou uspořádané. Klíče i hodnoty mohou být různých datových typů.

```
# definice dictionary
>>> d = {jedna: 1,dve: 2,tri: 3,seznam: ['a',2,3],Karel: ctvrty}
>>> d[seznam]
['a', 2, 3]
```

```
>> d[ctyri] = 7 # přidá key:value pár
>>> print(d)
{'jedna': 1, 'dve': 2, 'tri': 3, 'seznam': ['a', 2, 3], 'Karel':
        'ctvrty', 'ctyri': 7}
```

Operace s dictionary

d[k]	value s key == k
d[k] = x	do key == k vloží hodnotu x
d.clear()	vše odstraní, zůstane prazdné dictionary
del d[k]	smaže value s key = k
d.get(k, x)	vrátí d[k], pokud k je v d, jinak vrátí x
k in d	Je key k v d?
k not in	Není key k v d?
d.keys()	vráti list s keys
d.values()	vrátí list s values

d1.update(d2) doplní dictionary d1 o hodnoty d2

```
>>> del d[tri]
>>> d
{'jedna': 1, 'dve': 2, 'seznam': ['a', 2, 3], 'Karel': 'ctvrty',
        'ctyri': 7}
```

```
>> d.get(deset, 'není')
'není'
```

```
>> d.keys()
dict_keys(['jedna', 'dve', 'seznam', 'Karel', 'ctyri'])
```

2.5 Ostatní datové typy

BOOL	logická proměnná, nabývá hodnot ***True*** nebo ***False***
NONE	lze použít při vytvoření proměnné, kdy ještě nevíme, jakého bude typu, a později do ní uložit cokoli

2.6 Změna datového typu

Změna datového typu, běžně označovaná jako ***přetypování***, mění datový typ proměnné, ale (většinou) zachovává jeho hodnotu. Řadu přetypování Python provede sám (tzv. implicitně), aniž bychom ho k tomu vyzvali. Pro srozumitelnost kódu i zabránění nečekaných chyb je však lépe přetypovávat explicitně pomocí příkazů `int()`, `float()`, `complex()`,`bool()`, `set()`, `tuple()`, `list()`

```
>>> 5 + int(14)
19
```

```
>>> str(5) + '14' # konkatenace = připojení jednoho stringu za druhý
'514;
```

```
>>> int(1.7) # přetypování se ztrátou hodnoty
1
```

```
>>> float(3.1415)
3.1415
```

```
float(12)
12.0
```

```
>>> float(1E5)
1e-05
```

```
bool(None)
False
```

```
bool(1)
True
```

```
>>> str(3.1415)
'3.1415'
```

```
>>> type('12')
str
```

```
>>> a = input('Prosím zadejte hodnotu a: ') # načte hodnotu zadanou
                                    # na klávesnici a uloží jako string
>>> b = input('Prosím zadejte hodnotu b: ')
>>> soucet = float(a) + float(b)          # přetypuje string na float
>>> print('Součet  '+ a + 'a  '+ b + 'je  '+ str(soucet))
Prosím zadejte hodnotu a: 5.1
Prosím zadejte hodnotu b: 4.7
Součet 5.1 a 4.7 je 9.8
```

2.7 Porovnání

> < >= <=	větší, menší, větší nebo rovno, menší nebo rovno
==	má stejnou hodnotu
!=	má odlišnou hodnotu
is	je identický, t.j. má stejné ID = stejnou adresu v paměti
is not	je odlišný, t.j. má odlišné ID = odlišnou adresu v paměti

Každý objekt v Pythonu má své ***ID***, které značí umístění objektu v paměti.

```
>>> x = 149
>>> id(x)  # adresa v paměti
140730856847776
```

Stejný string je v paměti uložen pouze jednou, je to tedy identický objekt, na který může „ukazovat“ více různých proměnných (t.j. názvů).

```
>>> s1 = 'stejny'
>>> s2 = 'stejny'
>>> s1 is s2
True
```

3 Programové konstrukty

3.1 Podmínka

Nutným konstruktem každého programovacího jazyka je ***podmínka***, která umožní odlišné chování programu za různých situací. Základní součást podmínky je logický výraz, který je vyhodnocen jako pravdivý (True) nebo nepravdivý (False). Logický vyraz jsou vyhodnocován vždy zleva doprava, až je situace jasna, dál už ne. Porovnání ma přednost pred logickým výrazem.

Komponenty podmínky:

logické operátory	== != < >	
logické spojky	and	oba operandy musí být pravdivé
	or	alespoň 1 operand musí být pravdivý
	not	negace
	kombinace	(not) <operand 1> and/or (not) <operand 2> ...
	příklad	not 1 != 1 and not 2 < 3

```
>>> a = 150
>>> if a > 5 and a <= 100:    # za if následuje logický výraz
>>>   print('ano')           # provede, pokud je výraz pravdivý
>>> else:
>>>   print('ne')            # provede, pokud je výraz nepravdivý
ne
```

```
>>> a = 150
>>> b = 50
>>> if a > 200:
>>>    print(a je větší než 200)  # provede, pokud je  výraz pravdivý
>>> elif b > 60:                 # pokud byl výraz nepravdivý,
                                 # řeší se další logický výraz
>>>    print(a je menší než 200 a b je větší než 60)
>>> elif b > 40:
>>>    print(a je menší než 200, b je menší než 60, ale větší než 40)
>>> else:
       # provede, pokud byly všechny předchozí výrazy nepravdivé
>>>    print(a je menší než 200 a b je menší nebo rovno 40)
a je menší než 200, b je menší než 60, ale větší než 40
```

3.1.1 Podmíněný výraz

Podmíněný výraz je kompaktní verze podmínky.

```
>>> A = False
>>> text = 'platí A' if a == True else 'A neplatí'
>>> print(text)
A neplatí
```

3.2 Cykly

Dalším základním konstruktem každého programovacího jazyka jsou ***cykly***, tedy úseky programu, které jsou opakovány tak dlouho, dokud platí nějaká podmínka. Obvykle jsou k dispozici 2 cykly - ***while*** a ***for***. While se provádí, dokud platí podmínka. For cyklus v Pythonu prochází zadanou sekvenci, např. hodnoty 1 až 5. For cyklus lze nahradit while cyklem.

```
>>> a = 1
>>> while a = 100: # Nejprve se vyhodnotí podmínka. Pokud platí,
                   # proběhne 1 cyklus a znovu se hodnotí podmínka
>>>     a = a+3
>>> print(a)
103
```

```
>>> for i in [1,2,3,4]: # proměnná i v každém kroku nabývá další
                        # hodnoty z posloupnosti
>>>     print('opakování číslo: ' + str(i))
opakování číslo: 1
opakování číslo: 2
opakování číslo: 3
opakování číslo: 4
```

```
>>> staty = (Česko, Německo, Rakousko)
>>> for stat in staty:
>>>     print(stat)
Česko
Německo
Rakousko
```

```
>>> for i in [1, 2, 3, 4, 5]:
>>>     print(i*i, end= ' ~~ ')
1' ~~ '4' ~~ '9' ~~ '16' ~~ '25
```

```
>>> for i in [1, 2, 3]:
>>>     print('i se nemusí použít', end= ' ')
i se nemusí použít i se nemusí použít i se nemusí použít
```

```
>>> for i in range(5):   range(n) vytvoří sekvenci 0 až n1
>>> print(i, end=' ')
0 1 2 3 4
```

```
>>> for i in range(2, 5):  range(a,b) vytvoří sekvenci a až b1
>>>     print(i, end=' ')
2 3 4
```

```
>>> for i in range(1, 10, 2): # range(a,b,c) vytvoří sekvenci a až b
                              #   s krokem c
>>>     print(i, end=' ')
1 3 5 7 9
```

Příkaz ***pass*** umožní volný průběh podmínkou. Nestane se nic a cyklus či podmínka pokračuje dál. Hodí se k testování chyb - pokud chyba nenastala, nic se neděje a program běží dál.

```
# program zjišťuje počet souhlásek ve slově

>>> slovo = input(zadejte slovo: )
>>> pocet = 0
>>> for pismeno in slovo:
>>>     if pismeno in 'aeiouAEIOU': # malá nebo velká samohláska
>>>        pass                     # hned pokračuje další písmeno
>>>     else:
>>>       pocet = pocet + 1
>>> print('Slovo obsahuje'  + str(pocet) +  'souhlásek.')
zadejte slovo: abrakadabra
Slovo obsahuje 6 souhlásek.
```

3.2.1 break a continue

break a ***continue*** umožňují předčasné ukončení cyklu. Break ukončí cyklus úplně, continue ukončí aktuální běh a jde na další vyhodnocení podmínky či hodnotu for cyklu. Pokud za break následuje else, obsah else se neprovede.

```
# program testuje, zda je zadané číslo prvočíslo

>>> n = int(input('zadejte číslo: '))
>>> for i in range(2, n):
>>>   if n mod i == 0: # zbytek po dělení je 0, n je tedy dělitelné i
>>>     print(str(n) +  'je dělitelné'  + str(i))
>>>     break     # situace je jasná, cyklus nemusí pokračovat
>>>   if i == n:  # n není dělitelné číslem 2 až n, je tedy prvočíslo
>>>     print(str(n) + je prvočíslo)
zadejte číslo: 21
21 je dělitelné 3
```

```
# program zjišťuje počet samohlásek ve slově

>>> slovo = input('zadejte slovo: ')
>>> pocet = 0
>>> for pismeno in slovo:
>>>   if pismeno not in 'aeiouAEIOU': # malá nebo velká samohláska
>>>      continue                     # proměnná počet se nezmění
                                       # a pokračuje další písmeno
>>>   pocet = pocet + 1
>>> print('Slovo obsahuje'  + str(pocet) +  'samohlásky.')
zadejte slovo: slovotvorba
Slovo obsahuje 4 samohlásky.
```

3.3 Comprehensions

Comprehensions představují způsob, jak jednoduše vytvořit složité sekvence splňující nějaké podmínky. Zapisují se podobně jako množiny v matematice: „množina všech (funkcí) x, kde pro x cosi platí (x např. prochází členy nějaké sekvence či splňuje nějakou podmínku)“.

```
list = [výraz for prvek in sekvence]
dictionary = {key:value for prvek in sekvence}
```

```
>>> print(  [i**2 for i in [0, 1, 2, 3, 4]]  )
>>> print(  [i for i in range(50) if i6 == 0]  )
>>> print(  str(i):i**3 for i in range(4)  )
>>> print(  x*2:[sudý] if int(x)2 == 0 else [lichý] for x in [1,2,3])
[0, 1, 4, 9, 16]
[0, 6, 12, 18, 24, 30, 36, 42, 48]
{'0': 0, '1': 1, '2': 8, '3': 27}
{'11': ['lichý'], '22': ['sudý'], '33': ['lichý']}
```

4 Funkce

Dalším typickým konstruktem většiny programovacích jazyků jsou ***funkce***. Funkce je úsek textu programu, jemuž je přiřazen vlastní název a může být v programu opakovaně volán. Text funkce by se dal opakovaně psát přímo do hlavního textu programu, kdykoli je volána funkce (v takovém případě by volána nebyla), ale s pomocí funkcí se program mnohem lépe strukturuje i rychlejsi probíhá. Chovají se podobně jako v matematické - nějaké proměnné, označované jako parametry, do funkce vstupují, jiné proměnné jsou výsledkem funkce. Říkáme, že funkce vrací určité hodnoty (čísla, písmena, logickou hodnotu apod.). Funkce je v Python vlastním objektem, má tedy své jméno, adresu v paměti apod.

```
# definice funkce
>>> def jmenoFunkce(parametry):
>>>     prikazovy blok # odsazeni říká, co vše patří do funkce
>>>     return(...)     # říká, co má funkce vracet
```

```
>>> def pocetSamohlasek(slovo):
>>>      pocet = 0
>>>      for pismeno in slovo:
>>>          if pismeno in aeiouAEIOU:
>>>              pocet = pocet + 1
>>>      return(pocet)
```

```
>>> type(pocetSamohlasek)
function
```

```
>>> text = input('zadejte slovo: ')
>>> pocetSsamohlasek(text)
zadejte slovo: pokus
2
```

```
# i výrazy mohou být argumenty funkce
>>> len('len'  + 'zjišťuje'  + 'délku'  + 'textu')
24
```

```
# funkce může mít více výstupů
# funkce nemusí mít žádné parametry

>>> def viceVystupu():
>>>      a = 4
>>>      return(9,[1,2,3],pokus, a)

>>> cislo, vektor, text, promenna = viceVvystupu()
>>> print(cislo)
>>> print(vektor)
>>> print(text)
>>> print(promenna)
9
```

```
[1, 2, 3]
pokus
4
```

Užitečná je funkce ***map***, která aplikuje jinou funkci na každý prvek sekvence.

```
>>> list(map(len, [Praha, Brno, Ostrava])) # výsledek map přetypován
                                           #  na list
[5, 4, 7]
```

4.1 Předání parametru funkci

V programovacím jazyce C se používávé koncept ukazatele. ***Ukazatel*** v jazyce C je proměnná, která neobsahuje přímo číslo, písmo apod., ale obsahuje adresu v paměti, kde je uložen nějaký objekt, např. číslo, vektor, řetězec znaků. Může nastat situace, kdy 2 různé ukazatele s různými jmény obsahují stejnou hodnotu, tedy ukazují na stejnou oblast v paměti. O tom je důležité vědět, jinak se může stát, že změníme hodnotu proměnné, na níž ukazuje jeden ukazatel, a překvapí nás, že se změnila i hodnota, na níž ukazuje jiný ukazatel. Podobný princip se vyskytuje i jiných programovacích jazycích včetně Pythonu, např. při volání funkce a předávání parametru. V Pythonu se obvykle používá pojem reference namísto ukazatel.

Jsou 2 způsoby, jak se funkci předá parametr, vlastní předání probíhá automaticky:

call by-value Předává se skutečná kopie parametru, která se během činnosti funkce mění, původní proměnná zadávaná jako parametr zůstává nedotčena. Způsob je paměťově náročnější, protože se musí vytvořit kopie a uložit do paměti. Typické pro neměnitelné datové typy (číslo, string, tupel)

call-by-reference Předává se reference na proměnnou. Funkce přes referenci přistupuje přímo na proměnnou a tu mění. Mění se tedy původní proměnná, nevytváří se kopie. Typické pro měnitelné datové typy (set, dictionary, list).

```
>>> def coSeMeni(a, b): # jména proměnných uvnitř funkcí mohou být
                        #  libovolná
>>>     a[2] = 8        # a ukazuje na stejný list jako posloupnost
>>>     b = 8           # b je vlastní lokální proměnná funkce, již
                        #  nemá nic společného s cislo

>>> posloupnost = [1,2,3,4]  # call by reference
>>> cislo = 2                # call by value
>>> coSeMeni(posloupnost, cislo)
>>> print(posloupnost)
>>> print(cislo)
[1, 2, 8, 4]                 # list posloupnost se změnil
2                            # cislo se nezměnilo
```

4.1.1 Defaultní parametry

Abychom nemusili funkci vždy předat všechny parametry, lze použít tzv. ***defaultních parametrů***. Jejich hodnoty se zadají v definici funkce. Pokud se parametr nepředá, automaticky se použije defaultní hodnota.

```
>>> def mocnina(zaklad = 2, exponent = 2):
>>>     return(zaklad**exponent)

>>> mocnina()
4
```

```
>>> mocnina(3)
9
```

```
>>> mocnina(4, 3)
64
```

```
>>> mocnina(exponent = 3)
8
```

4.1.2 Libovolný počet parametrů

Někdy předem nevíme, kolik parametrů bude funkce zpracovávat. Např. pokud jsou parametrem slova načtená ze souboru. Pak lze použít * před parametrem. Funkce automaticky uloží všechny parametry do tuple, jíž lze procházet pomocí for cyklu.

```
>>> def soucet(*cisla):
>>>  print(type(cisla))
>>>  soucet = 0
>>>  for cislo in cisla:
>>>     soucet += cislo # zkrácený zápis pro soucet = soucet + cislo
>>>  return(soucet)

>>> soucet(1, 2, 3, 4, 5)
<class 'tuple'>
15
```

4.2 Jmenné prostory

Testování shody se v Pythonu provádí pomocí operátoru ==. Operátor = znamená přiřazení. Při zadání `x = 2` program vyhradí místo v paměti, uloží do něj číslo 2 a krom toho vytvoří jmenovku x, která obsahuje adresu, kde se hodnota 2 v paměti nachází. Obecně říkáme, že program v paměti vytvořil objekt určitého datového typu, který referencuje pomocí proměnné. Pomocí id(x) můžeme zjistit, jaká hodnota je v proměnné x zapsána, tedy na kterou buňku v paměti odkazuje. Pokud vytvoříme ***povrchovou kopii*** proměnné, nevytvoří se nový objekt v paměti, pouze se vytvoří nová jmenovka ukazující na tentýž objekt. Naproti tomu ***hluboká kopie*** vytvoří nový objekt jinde v paměti, který je shodný, ale ne identický s objektem původním. Krom toho vytvoří jmenovku, která ukazuje na nový objekt. Hluboká kopie se vytváří např. při předání parametru call-by-value.

```
>>> x=2 # v paměti uložil číslo 2 na adrese 140721937983296
>>> id(x)
140721937983296
```

```
>>> x=2
>>> y=x # vytvoří jen novou jmenovku ukazující na stejnou dvojku v
        # paměti. y je povrchová kopie x
```

```
>>> id(x)==id(y)
True
```

```
>>> x = [1,2,3]
>>> y = [1,2,3] # y je hluboká kopie x
>>> id(x)==id(y)
False
```

Z pohledu funkce extistují dva tzv. ***jmenné prostory***, globální a lokální. Proměnné, které byly definovány vně funkce, před jejím voláním, patří do ***globálního*** prostoru. Proměnné definované uvnitř funkce jsou v ***lokálním*** prostoru. Funkce nejprve hledá proměnné v lokálním prostoru a pracuje s nimi. Pokud proměnnou v lokálním prostoru nenajde, hledá v globálním prostoru. Pokud mají globální i lokální proměnná stejné jméno, pracuje funkce primárně s proměnnou lokální. Navíc ještě existuje vestavěný (build-in) jmenný prostor - funkce dosažitelné odkudkoli jako print, len, min, max, které jsou přímo součástí pythonu. Pomocí globals() a locals() lze vypsat obě skupiny proměnných.

```
>>> def funkceNaNic():
>>>     x = 20
>>>     # print(globals())
>>>     print(locals())

>>> x = 2
>>> funkceNaNic}()
>>> #print(globals())
>>> #print(locals())
{'x': 20}
```

Klíčové slovo ***global*** funkci informuje, ze proměnná není lokalní.

```
>>> def nasobeni(x): # x je zde nová lokální proměnná, které byla
                     #  předána hodnota 5 (call by value)
>>>     x = x * 3    # manipuluje s lokální proměnnou x, kterou
                     #  ztrojnásobí, ale globální x nemění
>>>     print(x)     # vytiskne lokální x
>>> x = 5           # globální x
>>> nasobeni(x)
>>> print(x)        # vytiskne globální x
15
5
```

```
>>> def nasobeni():
>>>     global x  # proměnná x použitá dále je globální
>>>     x = x * 3 # ztrojnásobí globální proměnnou x
>>>     print(x)  # vytiskne globální x
>>> x = 5        # globální x
>>> nasobeni()
>>> print(x)     # vytiskne globální x
15
15
```

4.3 Rekurze

Rekurze je zvláštní programový konstrukt, kdy funkce za určitých podmínek volá sama sebe. V matematice se podobně využívají rekurzivní definice, typickým příkladem je definice faktoriálu.

```
>>> def faktorial(n):
>>>     if n = 1:
>>>         return(1) # jakmile je parametr = 1, funkce vrátí 1 a
                      #  dále už sama sebe nevolá, čímž ukončí rekurzi
>>>     else:
>>>         return(n*faktorial(n1)) # funkce volá sama sebe, ale s
                                    #  parametrem o 1 nižším
>>> print(faktorial(10))
3628800
```

Rekurze je elegantní stručný zápis, nicméně při psaní programu je radno se jí vyhnout. Jednak je někdy komplikovaná na představu, co se v programu vlastně děje, jednak může rychle zpomalit program s nárůstem velikosti zadání, jednak je paměťově náročná. Při každém volání funkce v rekurzi je v paměti vytvořen nový objekt s lokálním jmenným prostorem, tzv. ***Execution Frame***. Obsahuje aktuální příkaz, který se má právě provést, lokální jmenný prostor a odkaz na nadřazený volající execution frame. Běh programu v daném execution frame je při volání další funkce v rekurzi pozastaven, dokud tato nevrátí hodnotu. Hloubka rekurze je omezena velikostí paměti.

4.4 Lambda funkce

Lambda funkce, též označovaná jako anonymní funkce, je „dočasná“ funkce, která nemá vlastní jméno, ani není uložena jako objekt v paměti. Je to vlastně speciální výraz „lambda argumenty: vyraz“. Funkce vrací hodnotu výrazu. Může zrychlit běh programu ve srovnání s použitím „obvyklé“ funkce definované pomocí `def`. Často se lamda funkce používá uvnitř následujících funkcí:

map(funkce, sekvence)	map na každý prvek sekvence aplikuje funkci
filter(funkce, sekvence)	filter vybere ze sekvence hodnoty definované ve funkci
list.sort(klic)	sort seřadí sekvenci podle klíče zadaného funkcí
sorted(list, klic)	sorted seřadí sekvenci podle klíče zadaného funkcí

```
>>> vstup = [1,2,3,4,5]
>>> vystup = map(lambda x: 3*x+5, vstup) # výstupem map je kolekce,
                                         # ale ne list
>>> print(list(vystup))                 # přetypování na list
[8, 11, 14, 17, 20]
```

Pokud se lambda funkce pojmenuje, stává se z ní „normální“ funkce

```
>>> soucet = lambda x, y: x + y # do funkce vstupují 2 parametry
>>> print(type(soucet))
>>> print(soucet(2, 11))
<class 'function'>
13
```

```
>>> fibo =[0, 1, 1, 2, 3, 5, 8, 13, 21, 34] # Fibbonaciho posloupnost
>>> licheFibo =filter(lambda x: x  2, fibo) # pokud je výsledkem 0,
                                            #  číslo se nevybere
```

```
>>> print(list(licheFibo))
[1, 1, 3, 5, 13, 21]
```

```
>>> seznam = [(2, C),(3, A),(1, B)]
>>> seznam.sort() # seřazení podle první hodnoty
>>> print(seznam)

>>> seznam.sort(key = lambda x: x[1]) # seřazení podle druhé hodnoty
>>> print(seznam)

>>> seznam2 = [(9,10,130),(3,331,1),(95,0,7,0,1),(35,1),(42,1)]
>>> print(sorted(seznam2)) # seřazení podle první hodnoty
>>> print(sorted(seznam2, key = lambda x: sum(x))) # podle součtu
[(1, 'B'), (2, 'C'), (3, 'A')]
[(3, 'A'), (1, 'B'), (2, 'C')]
[(3, 331, 1), (9, 10, 130), (35, 1), (42, 1), (95, 0, 7, 0, 1)]
[(35, 1), (42, 1), (95, 0, 7, 0, 1), (9, 10, 130), (3, 331, 1)]
```

5 Práce se soubory

Čtení z a zápis do souborů probíhá přes ***File objekt***. Při zápisu do souboru se data nejprve uloží do file objektu v paměti a teprve z něj jsou zapsána na disk. Opačný je průběh při čtení souboru z disku. File-objekt se tvoří pomocí `open(jmeno souboru, modus [, kodovani])`, metaforicky řečeno se „otevře soubor“. Modus znamená, v jaké podobě se soubor otevře, čili co umožní. Možnosti jsou:

r	read; otevřen pouze pro čtení
w	write; otevřen pro čtení a přepis souboru. První uložený znak celý soubor smaže a přepíše!!
a	append; otevřen pro čtení a přidání na konec souboru. První uložený znak soubor nesmaže, ale přidá se na konec souboru.
rb	read binary; otevřen pouze pro čtení v binárním modu, t.j. bez interpretace
wb	write binary; otevřen pro čtení a přepis souboru v binárním modu

5.1 Funkce file objektu

Funkce file objektu jsou následující:

mode	modus r, w, a, rw, rb
closed	True = soubor otevřený, False = soubor zavřený
close()	uložit a zavřít soubor
flush()	uložit, ale nezavírat soubor
read(bytes)	načte a vrátí obsah souboru (přesněji file objektu) jako string (po úsecích o maximální velikosti bytes)
readline()	načte a vrátí další řádek jako string
readlines()	načte a vrátí celý soubor po řádcích a uloží jako list stringů
seek(pos)	kurzor bude umístěn na polohu pos
tell()	vrací aktuální polohu kurzoru
write(str)	zapíše string str do souboru (přesněji do file objektu)

```
>>> data = open('c:\\Users\\Public\\soubor.txt', r, encoding=utf8)
           # otevře soubor, tedy vytvoří file objekt
           # r je defaultní modus
           # encoding určí způsob interpretace bytů čtených z disku
>>> print(data)
>>> print(data.closed)
>>> data.close() # uloží případné změny do souboru (v r modu nelze)
                 #  a uzavře file objekt
>>> print(data.closed)
<_io.TextIOWrapper name='c:\\Users\\Public\\soubor.txt' mode='r'
 encoding='utf-8'>
False
True
```

5.2 Zápis do souboru

```
>>> output = open('c:\\Users\\Public\\soubor.txt', w)
```

V tuto chvíli už je soubor.txt ***smazaný***, ale otevřený a připravený k zápisu!!!!

```
>>> print(output.closed)
False
```

```
>>> text = input('Prosím zadejte nějaký text: ')
>>> output.write(text + '\n')
Prosím zadejte nějaký text: zatím v souboru není nic zapsáno,
jen ve File objektu
```

```
>>> output.read() # soubor je otevřen jen pro zápis, proto nelze číst
```

```
-----------------------------------------
UnsupportedOperation Traceback (most recent call last)
<ipython-input-135-fb91d3264105> in <module>
--> 1 output.read ()
UnsupportedOperation: not readable
```

```
>>> print(output)
<_io.TextIOWrapper name='c:\\Users\\Public\\soubor.txt' mode='w'
 encoding='cp1250'>
```

```
>>> output.close() # až nyní se zapsaly změny
>>> output.read()  # nefunguje, soubor uz je zavřený
```

```
-----------------------------------------
ValueError Traceback (most recent call last)
<ipython-input-138-3ab1c9ca5a00> in <module>
--> 1 output.read () # nefunguje, soubor nebyl otevren ke cteni
ValueError: I/O operation on closed file.
```

```
>>> output = open('c:\\Users\\Public\\soubor.txt', r)
>>> print(output.read())
>>> output.close()
zatím v souboru není nic zapsáno, jen ve File objektu
```

```
>>> output = open('c:\\Users\\Publics\\soubor.txt', a)
                 # modus a = přidávání na konec souboru
>>> output.write('Tento text se přidá na konec souboru, ')
>>> output.flush() # nyní se zapíše na disk, ale nezavře
>>> output.closed  # False .... není zavřený
>>> output.write('do File objektu lze dále připisovat')
>>> output.close() # nyní se zapíše do souboru a File objekt se zavře
>>> output.closed  # True .... nyní je File objekt zavřený
>>> print(output.mode)
a
```

5.3 Načtení ze souboru

Při otevření souboru v modu čtení (r) řádky souboru se automaticky uloží jako speciální sekvence zvaná ***iterátor***, přes níž lze procházet (iterovat) pomocí for cyklu

```
>>> infile = open('c:\Users\Public\soubor.txt') # infile je iterátor
>>> print(type(infile))  # class _io.TextIOWrapper
>>> for line in infile:  # iterace přes iterátor
>>>     print(line)
>>> infile.close()
<class '_io.TextIOWrapper'>
zatím v souboru není nic zapsáno, jen ve File objektu
```

```
>>> infile = open('c:\\Users\\Public\\soubor.txt')
>>> print(infile.readline()) # alternativní načítání po řádcích
zatím v souboru není nic zapsáno, jen ve File objektu
Tento text se přidá na konec souboru, do File objektu lze dále
 připisovat
```

```
>>> print(infile.readline())
Tento text se přidá na konec souboru, do File objektu lze dále
 připisovat
```

```
>>> print(infile.readline()) # načtení řádku, který neexistuje
                             #  ...vrátí prázdný řádek
```

```
# načtení souboru jako listu řádků
>>> infile = open('c:\\Users\\Public\\soubor.txt').readlines()
>>> print(type(infile))
>>> print(infile)
<class 'list'>
['zatím v souboru není nic zapsáno, jen ve File objektu\n', 'Tento
text se přidá na konec souboru, do File objektu lze dále
připisovat \n']
```

```
# načtení souboru jako dlouhého stringu
>>> infile = open('c:\\Users\\Public\\soubor.txt').read()
>>> print(type(infile))
>>> print(infile)
<class 'str'>
zatím v souboru není nic zapsáno, jen ve File objektu
Tento text se přidá na konec souboru, do File objektu lze dále
připisovat
```

5.4 with..as

Alternativně lze soubor otevřít pomocí with..as. Výhodou je, že jakmile skončí `with` blok, soubor se automaticky uzavře.

```
>>> with open('c:\\Users\\Public\\dulezity_soubor.dat', w) as output:
>>>     output.write('Tento soubor se zavře i bez použití close().')
```

```
>>> with open('c:\\Users\\Public\\dulezity_soubor.dat', r) as output:
>>>     print(output.read())
Tento soubor se zavře i bez použití close().
```

5.5 Nastavení polohy kurzoru

Příkaz `seek` nastavuje polohu kurzoru. Naopak `tell()` sdělí aktualní polohu kurzoru. Obě funkce jsou užitečné např. při „automatických“ úpravách textu.

seek(x, 0)	nastaví kurzor x pozic za začátek souboru
seek(x, 1)	nastaví kurzor x pozic za aktuální pozici
seek(0, 2)	nastaví kurzor na konec

```
>>> output = open('c:\\Users\\Public\\soubor.txt', rb)
              # některé fungují pouze v binárním modu
>>> print(output.tell())
>>> output.seek(50)
>>> print(output.tell())
>>> output.seek(5, 1) # 5 pozic za aktuální pozici
>>> print(output.tell())
>>> output.seek(0, 2) # kurzor na konec
>>> print(output.tell())
0
50
55
164
```

5.6 Odchytávání chyb

Pokud program narazí na chybu, t.j. příkaz, který není možné provést, např. dělení nulou, nebo otevření souboru, který neexistuje, jeho běh se ukončí. Přitom však jsou některé „chyby“ normální, očekávané, např. právě situace, kdy některý soubor neexistuje. Program by tedy v takovém případě měl chybu rozeznat, uživatele na ni upozornit a pokračovat v běhu. Takovým „polochybám“ říkáme ***výjimky***. ***Odchytávání chyb*** je programový konstrukt, který umí řešit právě takové výjimečné situace. Typicky se uplaňuje při práci se soubory, kde výjimky mohou očekávaně nastat. V Pythonu existují příkazy `try`, `except`, `else` a `finally`.

`try`	určí oblast programu, kde se odchytává chyba
`except`	určí, co se stane, pokud chyba nastane
`else`	určí, co se stane, pokud chyba nenastane
`finaly`	provede se, ať už chyba nastane nebo ne, resp. odchytí se nebo ne

```
>>> print('tento text se napíše, protože program zatím běží')
>>> e = open('/cesta/k_souboru/neexistuje/soubor_take_ne.txt')
        # bez ošetření chyby se program ukončí. FileNotFoundError
>>> print('tento text se už nenapíše, protože se program skončil')
tento text se napíše, protože program zatím běží
```

```
-----------------------------------------
FileNotFoundError Traceback (most recent call last)
<ipython-input-138-3ab1c9ca5a00> in <module>
--> 1 output.read () "tento text se už nenapíše, protože se program skončil"
```

```
FileNotFoundError: [Errno 2] No such file or directory:
'/cesta/k_souboru/neexistuje/soubor_take_ne.txt'
```

```
>>> try: # v následujícím řádku bude odchycena chyba, pokud nastane
>>>     e = open('/cesta/k_souboru/neexistuje/soubor_take_ne.txt')
>>> except:
>>>     print('Chyba byla zachycena.')
>>>     print('Program pokračuje dál.')
Chyba byla zachycena.
Program pokračuje dál.
```

```
>>> data = 'velmi důležitá data'
>>> try:
>>>     e = open('/cesta/k_souboru/neexistuje/soubor_take_ne.txt'},w)
              # FileNotFoundError
>>>     e.write(data) # data se nemohla uložit
>>>     e.close()
    # chybí except, chyba se neodchytí a program skončí
>>> finally: # chyba se sice neodchytí, ale data se před ukončením
              #  programu ještě uloží
>>>     f = open('c:\\Users\\Public\\zaloha.txt', w)
>>>     f.write(daten)
>>>     f.close()
```

```
---------------------------------------
FileNotFoundError Traceback (most recent call last)
<ipython-input-181-e024678d284e> in <module>
1 daten = 'velmi důležitá data'
2 try:
--> 3 e = open('/cesta/k_souboru/neexistuje/soubor_take_ne.txt', 'w')
# FileNotFoundError
4 e.write(daten) # data se nemohla uložit
5 e.close()
FileNotFoundError: [Errno 2] No such file or directory:
'/cesta/k_souboru/neexistuje/soubor_take_ne.txt'
```

6 String Objekty

Textové řetězce jsou v Pythonu reprezentovány datovým typem string. Oproti některým jiným programovacím jazykům neexistuje v Pythonu datový typ `char`, reprezentující jednotlivé znaky. I jednotlivé znaky jsou v Pythonu datového typu string. Vytvořením řetězce pomocí "..."nebo '...' vznikne string objekt v paměti. Je neměnitelný, t.j. při jakékoli změně se vytvoří nový string objekt s jinym id(). String objekty mají řadu in-built funkcí, umožňujících modifikaceů řetězců.

6.1 Standardní string metody

```
>>> print('uprostřed'.center(20,'+'))  # + kolem uprostřed, délka 20
>>> print('doprava'.rjust(10))        # posune string doprava
>>> print('doleva'.ljust(10, '!'))    # posune string vlevo,zbytek !
>>> print('Bedřich Smetana'.lower()) # vše malými písmeny
>>> print('Antonín Dvořák'.upper())  # vše velkými písmeny
>>> print('věta má začínat velkým písmenem'.capitalize())
                                     # první písmeno velké
+++++uprostřed++++++
doprava
doleva!!!!
bedřich smetana
ANTONÍN DVOŘÁK
Věta má začínat velkým písmenem
```

```
>>> print('Končí text otazníkem?'.endswith('?')) # shoda na konci
>>> print('Například pes, kočka, kůň...'.isalpha())
    # Jde o alfanumerický výraz? !tečky ani čárky nejsou alfan. výraz
>>> print(''.isalnum())   # prázdný řetězec není alfanumerický string
>>> print('19'.isdigit()) # Jde o číslo?
>>> print('pondělí, úterý, středa'.islower()) # Jen malá písmena?
>>> print('ČTVRTEK, PÁTEK'.isupper())         # Jen velká písmena?
True
False
False
True
True
True
```

```
>>> print('      Zleva    '.lstrip() + '!' )
      # odstraní mezery na začátku řetězce
>>> print('      Zprava    '.rstrip() )
     # odstraní mezery na konci řetězce
>>> print('      obouSTRAanně    '.strip(' oR') )
     # z řetězce odstraní mezery nebo o nebo R, z obou stran

>>> text = 'Po této větě končí řádek.\n Pokračuje další řádek!
```

```
            Končí však teprve zde.\n Třetí řádek bude poslední.'
>>> print( text.splitlines() ) # rozštípe text na list řádků (konec
                               #  řádku je vyznačen \n)
>>> print( text.split() )      # rozštípe text na list stringů podle
                               #  mezery (slova)
>>> print(text.split('řádek')) # rozštípe text na list stringů podle
                               #  slova řádek
Zleva    !
Zprava
bouSTRAanně
['Po této větě končí řádek.', 'Pokračuje další řádek! Končí však
 teprve zde.', 'Třetí řádek bude poslední.']
['Po', 'této','větě', 'končí', 'řádek.','Pokračuje','další','řádek!',
'Končí', 'však','teprve', 'zde.', 'Třetí','řádek','bude','poslední.']
['Po této větě končí ', '.\nPokračuje další ', '! Končí však teprve
zde.\nTřetí', ' bude poslední.']
```

```
>>> print( text.count('de') )         # počet výskytů v textu
>>> print( text.find('de') )          # poloha prvního výskytu v textu
                                      # pokud v textu nenajde, vrátí 1
>>> print(text.replace('de', 'DE'))   # nahradí všechny de pomocí DE
5
21
Po této větě končí řáDEk.
Pokračuje další řáDEk! Končí však teprve zDE.
Třetí řáDEk buDE poslední.
```

6.2 Aplikace string metod na soubor

```
# rozštěpení textu souboru na slova
>>> with open('c:\Users\Public\dulezity_soubor.dat') as inputFile:
>>>     for line in inputFile:
>>>         print(line.split( ))
['Tento', 'soubor', 'se', 'zavře', 'i', 'bez', 'použití', 'close().']
```

```
>>> with open(c:UsersPublicdulezity\PYZus{soubor.dat}) as inputFile:
>>>     fileContent = inputFile.readlines()

>>> print( fileContent[0].split( ) )
>>> print( fileContent[0].split( )[1].split(s) )
     # zřetězení funkcí. Poslední řádek rozštěpí podle písmene s
['Tento', 'soubor', 'se', 'zavře', 'i', 'bez', 'použití', 'close().']
['clo', 'e().']
```

6.3 Automatické doplňování a formátování textu

Používání se k doplnění hodnot proměnných do textu, přičemž jejich hodnoty jsou známy až v běhu programu, nikoli při jeho psaní. Možnostmi jsou konkatenace, %-notace, format() notace a nejnověji f notace.

6.3.1 Konkatenace

```
>>> cislo = 19
>>> text = '5 ' + str(cislo) + ' ' + str(36)
>>> print(text)
5 19 36
```

6.3.2 % notace

Doplňovaný text stojí před %, doplněné hodnoty ve stejném pořadí za %. %x vyznačuje místo doplnění.

%s	doplněný text je typu string
%d	doplněný text je přirozené číslo
%.2f	doplněný text je typu float, zaokrouhleno na 2 desetinná místa

```
>>> cislo = 5
>>> print(%s %d %a %d je .2f % ('Podíl z', 19, cislo, 3.849))
Podíl z 19 a 5 je 3.85
```

6.3.3 format() notace

Doplňovaný text stojí v {}, doplněné hodnoty uvnitř format().

```
>>> x = 2
>>> 'Za kostelem {}, pak {} ulice rovně.'.format('doprava', x, 1, 4)
                                          # ignoruje nadbytečná data
'Za kostelem doprava, pak 2 ulice rovně.'
```

```
# lze i měnit pořadí nebo označovat proměnné jmény
>>> napoj = čaje
>>> 'Krabice {a} měří {1} x {0} x {2} cm.'.format(10, 25, 8, a=napoj)
'Krabice čaje měří 25 x 10 x 8 cm.'
```

6.3.4 f - notace

Nejnovější a nejsnazší. Syntaxe: `f'..\{..\}..\{..\}'`.

```
>>> a = Karl
>>> b = Popper
>>> print(f'Rakouský filozof {a} {b}.')
>>> print(f'Výsledek: {5.193669119149:>10.2f}')
     # desetinné číslo, zaokrouhlené na 2 desetinná místa,
     #  minimální šířka 10 znaků
Rakouský filozof Karl Popper.
Výsledek:       5.19
```

Specifikace f notace: `proměnná : zarovnání minimální_šířka .přesnost typ_čísla`

zarovnání	<	zarovnání doleva
	>	zarovnání doprava
	^	centrovaný
minimální šířka		minimální počet rezervovaných míst

přesnost	počet desetinných míst	
typ čísla	d	int
	f	float
	x	hexadecimální
	b	binární

```
>>> tabulka = ''
>>> for i in range(1,6):
>>>     tabulka += f'{i:>10d}{i**2:>8.2f}{i**3:>6x}\n'
>>> print(tabulka)
     1    1.00     1
     2    4.00     8
     3    9.00    1b
     4   16.00    40
     5   25.00    7d
```

6.4 Datový formát CSV

CSV soubory mají specifikovaný formát, z nějž lze zrekonstruovat tabulku. Jednotlivé buňky jsou odděleny čímkoli (např. , nebo tab nebo mezerou) a řádky pomocí `\n`. Soubor nemusí mít příponu .csv, podstatné je, zde je uvnitř dodržen csv formát. V následujícím příkladu načítáme soubor.txt, který je napsám v csv formátu:

```
# obsah souboru soubor.txt
                            1 2 3 4 \n
                            4 5 6 7 \n
                            7 8 9 10
```

Snadněji a v návaznosti na další běžné činnosti s tabukami lze csv soubor načíst pomocí funkce `pandas.read_csv()` z balíku ***pandas*** (viz kapitola ...).

```
>>> import csv
# načtení csv souboru
>>> reader = csv.reader(open(c:UserPublicsoubor.txt))
             # automaticky interpretuje csv formát a uloží jako
             #  iterovatelný objekt reader
>>> tabulka = [radek for radek in reader]
             # převedení readeru na list řádků, s pomocí
             #  comprehensions (viz kapitola 2.5)
>>> print(tabulka)
[['1 2 3 4'], ['4 5 6 7 '], ['7 8 9 10']]
```

```
>>> tabulka_jako_list = ['1\t2', 'Isaak\tNewton' ,
                                        'Christian\tDoppler']
>>> reader = csv.reader(tabulka_jako_list, delimiter='\t')
             # delimiter = symbol oddělující buňky tabulky
>>> tabulka = list(reader)
>>> print(tabulka) # chápe tabulator jako oddělovač buněk

>>> reader = csv.reader(tabulka_jako_list, dialect=excel)
            # používá oddělovače stejně jako excel
>>> print(list(reader)) # nechápe \t jako oddělovač buněk
[['1', '2'], ['Isaak', 'Newton '], ['Christian', 'Doppler']]
```

```
[['1\t2'], ['Isaak\tNewton '], ['Christian\tDoppler']]
```

```
# zápis do csv souboru
>>> tabulka = [['jméno', 'příjmení', 'věk', 'povolání'], ['Martha',
                                          'Argerich', 77, 'klavíristka']]
>>> with open('c:\\Users\\Public\\tabulka.txt', w, newline='') as f:
>>>      writer = csv.writer(f)
>>>      writer.writerows(tabulka)

>>> tabulka2 = list(csv.reader(open('c:\\Users\\Pu..\\tabulka.txt')))
>>> print(tabulka2)
[['jméno', 'příjmení', 'věk', 'povolání'], ['Martha', 'Argerich',
'77', 'klavíristka']]
```

7 Systémové programování

Systémovým programováním se rozumí ovlivňování či volání funkcí operačního systému z Pythonu.

K dispozici jsou (nejméně) 3 moduly:

sys	zajišťuje přístup k a umožnuje ovlivnění interpretu Pythonu
os	umožní přímo volat funkce operačního systému (např. správu dat a procesů)
time	umožní interakce se systémovými hodinami

7.1 sys

Vybrané funkce a atributy modulu sys

argv	list argumentů příkazové řádky
platform	označení aktuální platformy operačního systému (linux, win32 ...)
exc_info()	vrací trojici hodnot s informací o výjimce, která se právě řeší
exit()	ukončí provádění python skriptu
modules	dictionary všech standardních modulů v Pythonu
path	list stringů vyhledávacích cest, kde se hledají moduly
stdin	file-objekt pro vstup, lze jen číst
stdout	file-objekt pro výstup, lze jen zapisovat
stderr	file-objekt pro chyby, lze jen zapisovat
version	string s aktuální verzí interpretu Pythonu

```
# list všech atributů a funkcí modulu sys

>>> import sys
>>> print(dir(sys))
['__breakpointhook__', '__displayhook__', ... # zkráceno
... 'thread_info', 'version', 'version_info', 'warnoptions']
```

```
# příklad zjištění operačního systému a interpretu Pythonu
# a chování programu podle jeho typu

>>> import sys
>>> if sys.platform == 'linux':
>>>     path = '/mnt/home/Documents/file.txt'
>>> elif sys.platform in ['win32', 'win64']:
>>>     path = 'C:\\Users\\Public\\soubor.txt'
>>> else:
>>>     sys.exit('Váš operační systém není podporován.')

>>> print(sys.version_info)   # aktuální verze interpretu Pythonu
>>> print(sys.version)        # aktuální verze interpretu Pythonu
>>> if sys.version[0] == '2':   # Python verze 2
>>>     print('Používáte Python verzi 2.')
```

```
>>> elif sys.version[0] == '3': # Python verze 3
>>>     print('Používáte Python verzi 3.')
sys.version_info(major=3, minor=8, micro=5, releaselevel='final', serial=0)
3.8.5 (default, Sep  3 2020, 21:29:08) [MSC v.1916 64 bit (AMD64)]
Používáte Python verzi 3.
```

Jak již bylo uvedeno, pythonský skript lze spustit přímo z příkazové řádky (***command line***), přitom lze předat argumenty, jimiž se bude řídit běh programu. Uvnitř programu lze tyto argumenty zjistit pomocí příkazu sys.argv, který vrací list argumentů. Spusťme z příkazové řádky pomocí `python3 muj_python_skript.py argument1 argument2 argument3` např. skript, který obsahuje následující text:

```
import sys
print(len(sys.argv))  # počet argumentů
print(type(sys.argv)) # datový typ sys.argv
for arg in sys.argv:
    print(arg, end= ' ')
```

Výstupem je

```
4 # první argument je jméno skriptu !!!
<'class list'>
```

7.2 os

Modul `s` simuluju příkazovou řádku operačního systému, umožňuje zhruba totéž. Tedy např. vytvářet a mazat adresáře, spouštět programy, měnit parametry operačního systému apod.

path	cesta k adresáři nebo souboru, rozlišujeme absolutní a relativní cestu
absolutní cesta	začíná u hlavního kořene, např. `'c:\\abc\\def\\soubor.ghi'`
relativní cesta	začíná z aktuálního adresáře, např. pokud jím je abc, pak `'..\\def\\soubor.ghi'`

Operace se soubory a adresáři

chdir(path)	změna adresáře (change directory)
getcwd()	aktuální adresář (get current working directory)
listdir(path)	výpis obsahu aktuálního adresáře (list of directory)
path.isdir(path)	vrací True, pokud path je adresář
path.isfile(path)	vrací True, pokud path je soubor
access(path, mode)	prověří přístupová práva a existenci souboru
chmod(path, mode)	mění přístupová práva k soubour či adresáři (change mode)
mkdir(path)	vytvoří adresář daný v path
makedirs(path)	vytvoří adresář daný v path se všemi podadresáři
remove(path)	odstraní soubor daný v path
rmdir(path)	odstraní prázdný adresář daný v path
removedirs(path)	odstraní všechny prázdné adresáře dané v path
rename(old, new)	přejmenuje adresář či soubor old na new
renames(old, new)	přejmenuje adresář či soubor old na new i všechny mezi-adresáře
path.exists(path)	true, pokud path existuje

```
>>> import os
>>> print(os.getcwd())                  # vypíše aktuální adresář
```

```
>>> os.chdir( 'C:\\Users\\Public' )     # změna adresáře
>>> print(os.getcwd())
>>> print(os.listdir(os.getcwd())) # vypíše obsah aktuálního adresáře
>>> print(os.path.isdir(../anaconda3)) # Jedná se o adresář?
                                       #  ..je relativní cesta
>>> print(os.path.isfile('C:\Users\Public\soubor.txt')) # Jedná se o
                                    # soubor? použita absolutní cesta
C:\Users\sitina.michal
C:\Users\Public
['AccountPictures', 'Desktop', 'desktop.ini','Documents','Downloads',
'dulezity_soubor.dat','Foxit Software','Librar','Music','NTUSER.DAT',
'NTUSER.DAT.LOG1', ..... 'tabulka.txt', 'Thunder Network', 'Videos',
'zaloha.txt']
False
True
```

```
>>> import os
>>> os.mkdir('C:\Users\Public\new')
>>> os.makedirs('C:\Users\Public\new\newer\newest')
>>> os.mkdir('C:\Users\Public\new\newer\newest') # FileExistsError
>>> os.rename('C:\Users\Public\soubor.txt',   # přejmenování souboru
              'C:\Users\Public\soubor.dat')
>>> os.rename('C:\Users\Public\soubor.dat',      # přesunutí souboru
              'C:\Users\Public\new\soubor.dat')
>>> os.remove('C:\Users\Public\newsoubor.dat')   # odstraní soubor
>>> os.rmdir('C:\Users\Public\new') # chyba  adresář není prázdný
>>> os.removedirs('C:\Users\Public\new\newer\newest')
                                    # odstraní adresáře až po Public
```

`os.walk(path)` projde celou path a vrátí iterátor přes trojice (aktuální adresář, list podadresářů, list souborů().

```
>>> import os
>>> g = os.walk('C:\\Users\\Public')
>>> for element in g:
>>>     print(element)
('C:\\Users\\Public', ['AccountPictures', 'Desktop', .....
........('C:\\Users\\Public\\Videos', [], ['desktop.ini'])
```

```
# alternativně
>>> import os
>>> for path, dirs, files in os.walk('C:\\Users\\Public'):
>>>     print(f'{path} || {dirs} || {files}')
C:\Users\Public || ['AccountPictures', 'Desktop', 'Documents',
'Downloads', 'Foxit Software', 'Libraries', 'Music', 'Pictures',
'Thunder Network','Videos']||['desktop.ini','dulezity\_soubor.dat',
'NTUSER.DAT', 'NTUSER.DAT.LOG1','NTUSER.DAT.LOG2', '.....
.....C:\Users\Public\Videos || [] || ['desktop.ini']
```

7.2.1 Propojeni na operační systém

Pomocí `os.system(příkaz)` lze provést příkaz operačního systému, jako by byl zadán přímo do příkazové řádky. Lze tak např. uvnitř běhu pythonského programu spustit jiný pythonský skript

nebo jiný program.

```
>>> import os
>>> os.chdir('C:\\Users\\Public')
>>> os.system(dir) # Windows vypsal obsah adresáře...výsledek
                   #  se ale vytiskl "uvnitř", nevidíme jej
>>> os.system( notepad tabulka.txt) # ze skriptu lze spustit jiný
                    # soubor  zde se otevře tabulka.txt v Notepadu
0
```

`os.popen(command, mode)` funguje stejně jako `os.system`, ale výsledek vytiskne přímo na obrazovku, nikoli „uvnitř“.

```
>>> import os
>>> os.chdir(C:UsersPublic)
>>> proces = os.popen(dir ,r ) # otevře a spustí proces
>>> vysledek = proces.read()   # načtení výsledku příkazu Windows
>>> proces.close()             # ukončení procesu
>>> print(vysledek)

Volume in drive C is Windows
Volume Serial Number is 9A67-A78C

Directory of C:\\Users\\Public

08.02.2022  10:27    <DIR>          .
08.02.2022  10:27    <DIR>          ..
24.04.2021  07:53    <DIR>          Documents
24.04.2021  06:31    <DIR>          Downloads
05.02.2022  22:27                44 dulezity_soubor.dat
14.11.2019  09:16    <DIR>          Foxit Software
24.04.2021  06:31    <DIR>          Music
14.11.2019  23:54             8.192 NTUSER.DAT
24.04.2021  06:31    <DIR>          Pictures
07.02.2022  14:27                61 tabulka.txt
29.10.2020  19:44    <DIR>          Thunder Network
24.04.2021  06:31    <DIR>          Videos
05.02.2022  22:23                19 zaloha.txt
4 File(s)          8.316 bytes
9 Dir(s)  99.319.402.496 bytes free
```

Stejnou funkci nabízí modul ***subprocess***

```
>>> import subprocess
>>> proces = subprocess.Popen(dir,stdout=subprocess.PIPE,shell=True)
>>> vysledek = proces.stdout.read()
>>> print(vysledek) # pozn. zkráceno
b' Volume in drive C is Windows\r\n Volume Serial Number is
9A67-A78C\r\n\r\n Directory of C:\\Users\\Public\r\n\r\n
08.02.2022  10:27    <DIR> .\r\n08.02.2022  10:27 .......
..... 99\xff456\xff524\xff288 bytes free\r\n'
```

7.3 time

Čas lze v Pythonu zadávat ve 3 formátech:

sekundy od 1.1.1970 00:00	1295763
tuple	n-tice time.struct_time(tm_year=2020, tm_mon=2, tm_mday=19, tm_hour=13, tm_min=7, tm_sec=50, tm_wday=2, tm_yday=50, tm_isdst=0)
string z 24 znaku	např. 'Tue Apr 10 19 05 36 1990'

Vybrané funkce modulu time

asctime(tuple)	mění tuple na 24 znaků nebo udá aktuálni čas v 24-znakovém formátu
ctime(secs)	mění sekundy na 24 znaků nebo udá aktuálni čas v 24-znakovém formátu
gmtime(secs)	mění sekundy na tuple, nebo udá aktuálni čase jako tuple
localtime(secs)	udá aktuálni lokální čas jako tuple
mktime(tuple)	mění tuple na sekundy
sleep(n)	čeká n sekund
time()	aktuální čas v sekundách
wait(x)	zastaví proces na x sekund

```
>>> import time
>>> print(time.asctime())
>>> print(time.gmtime())
>>> print(time.localtime())
>>> print(time.time())
>>> t = time.time()      # uloží aktuální čas
>>> time.sleep(1)        # spí 1 sekundu
>>> print(time.time()t) # časový rozdíl
Tue Feb  8 11:17:08 2022
time.struct\_time(tm\_year=2022, tm\_mon=2, tm\_mday=8, tm\_hour=10,
tm\_min=17, tm\_sec=8, tm\_wday=1, tm\_yday=39, tm\_isdst=0)
time.struct\_time(tm\_year=2022, tm\_mon=2, tm\_mday=8, tm\_hour=11,
tm\_min=17, tm\_sec=8, tm\_wday=1, tm\_yday=39, tm\_isdst=0)
1644315428.9226818
1.0149154663085938
```

```
print(time.asctime(time.localtime())) # konverze formátu času
Tue Feb  8 11:17:14 2022
```

7.3.1 Měření trvání běhu programu

```
>>> t = time.time()      # počáteční čas
>>> soucet = 0
>>> for i in range(1000000):
>>>     soucet += i*i
>>> print(time.time()-t) # čas koncový - počáteční
>>> print(soucet)        # tisk výsledku není vhodné zahrnout do mě-
                         #  ření trvání programu, protože může trvat
                         #  déle než vlastní program
0.18821930885314941
333332833333500000
```

8 Importování modulu

Jak již bylo poznamenáno, základní Python obsahuje jen asi 35 výrazů, klíčových slov, které v zásadě stačí k naprogramování čehokoli. Na druhou stranu naprogramovat i vcelku jednoduchou funkci pomocí základních příkazů může být složité a zdlouhavé. Proto existují doplňkové moduly,k teré umožňují snadno provést řadu jinak složitých činností. V jiných programovacích jazycích se používají i další synonyma - např. baličky, packages, pakety, knihovny. Modul sám je „obyčejný“ pythonský skript, tedy soubor zakončený (obvykle) koncovkou .py, který je psán pomocí základních příkazů nebo pomocí dalších modulů. Většinou obsahuje několik funkcí. Příslušný modul je třeba ztáhnout a uložit do některého z adresáře, v nichž Python moduly hledá (viz dále). Pro použití modulu v programu je třeba modul importovat pomocí příkazu `import`. V programu pak používáme jednotlivé funkce modulu. Modul random například obsahuje funkci randint().

```
>>> import random # soubor random.py byl nalezen v adresáři
                 #  C:\\Users\\sitina.michal\\anaconda3\\lib
>>> print(random)
>>> print(random.randint(5,19)) # modul random obsahuje řadu funkcí
                                #  generujících náhodná čísla.randint
                                #  generuje náhodné číslo mezi 5 a 19
>>> print(randint(5,19)) # Python nezná přímo funkci randint, musíme
                         #  mu prozradit, že je v modulu random
<module 'random' from 'C:\Users\sitina.michal\anac...\lib\random.py'>
10
```

```
>>> from random import randint  # alternativní způsob importu, kdy se
                                #  neimportuje celý modul, ale pouze
                                #  funkce randint
>>> print(randint(0,10))        # funkci randint lze v tomto případě
                                #  volat přímo
2
```

```
>>> from random import randint as ri # místo randint lze dále psát ri
                                     # výhodné u dlouhých názvů funkcí
>>> print(ri(0,100))
87
```

8.1 Průběh importování modulu

Python hledá modul jako stejnojmenný soubor .py v následujícím pořadí:

1. v aktuálním adresáři
2. v adresářích uvedených v tzv. proměnné prostředí (environment variable) PYTHONPATH (proměnná systému Windows, nikoli proměnná Pythonu)
3. ve standardní instalační cestě Pythonu, např. /anaconda3/lib/site-packages (zde jsou uloženy standardní knihovny)

Proměnná `sys.path` obsahuje všechny cesty, kde Python interpret hledá

```
>>> import sys
>>> print(sys.path) # vypíše všechny cesty (t.j. 1, 2 i 3), kde
                    #  Python interpret hledá
['C:\Users\sitina.michal\MS\Python and Python for R\python skriptum',
'C:\\Users\\sitina.michal\\anaconda3\\python38.zip',
'C:\\Users\\sitina.michal\\anaconda3\\DLLs',
'C:\\Users\\sitina.michal\\anaconda3\\lib',
'C:\\Users\\sitina.michal\\anaconda3', '',
'C:\\Users\\sitina.michal\\anaconda3\\lib\\site-packages',
'C:\\Users\\sitina.michal\\anaconda3\\lib\\site-
packages\\locket-0.2.1-py3.8.egg',
'C:\\Users\\sitina.michal\\anaconda3\\lib\\site-packages\\win32',
'C:\\Users\\sitina.michal\\anaconda3\\lib\\site-packages\win32\\lib',
'C:\\Users\\sitina.michal\\anaconda3\\lib\\site-packages\\Pythonwin',
'C:\\Users\\sitina.michal\\.ipython']
```

8.2 Vytvoření vlastního modulu

Snadno lze vytvořit vlastní modul. Vytvoří se pythonský skript obsahující někajé funkce, s příponou .py. Vytvořme soubor mujmodul.py, v programu definujme funkci moje funkce a soubor uložme do stávajícího adresáře.

```
>>> import os
>>> print(os.getcwd())  # vypíše aktuální adresář
C:\Users\Public
```

```
# mujmodul.py....uložili jsme jej v C:\\Users\\Public
>>> def mojefunkce(): # definice funkce uvnitř modulu
>>>     print('Používáte funkci z modulu mujmodul')
```

```
>>> mujmodul.py in os.listdir(os.getcwd())
True  # potvrzuje, ze mujmodul.py je uložen v C:\\Users\\Public
```

```
>>> import mujmodul       # importování vlastního modulu mujmodul.py
>>> mujmodul.mojefunkce() # použití metody z vlastního modulu
Používáte funkci z modulu mujmodul
```

8.3 Instalace externích paketů

Nainstalovat externí paket znamená na webu najít paket a všechny nutné související pakety (tzv. dependencies), uložit je do příslušných knihoven Pythonu (např. /anaconda3/lib/site-packages) a případně doplnit cesty do proměnných prostředí. To vše, zejména pokud je potřeba více souvisejících paketů, může být komplikované. Jednodušší je použít instalátorů (tzv. package managers) pip nebo conda.

Instalátor ***pip*** je automaticky součástí instalace Pythonu. V příkazové řádce Windows stačí zadat pip install jmeno_paketu. Pip nainstaluje modul včetně všech dependencies. Pip automaticky hledá pakety v ***Python Package Index*** - https://pypi.org/. Lze však instalovat pakety i z jiné webové adresy (pip install –index-url http://nejaka.webova.stranka/test/jmenoPaketu) nebo z vlastního adresáře (pip install ..\downloads\SomePackage-1.9.0.zip).

pip list	ukáže již nainstalované pakety
pip search jmenoPaketu	hledá paket nebo jemu podobné
pip install jmenoPaketu	nainstaluje paket
pip install jmenoPaketu –upgrade	aktualizuje verzi paketu

Instalátor ***conda*** je součástí distribuce Anaconda. V příkazové řádce Windows se condal install jmeno_paketu. Též nainstaluje všechny dependence. Pokud nelze conda spustit přímo z příkazové řádky Windows, je nutno spustit Anaconda Navigator a v něm CMD.exe Prompt.

conda list	ukáže již nainstalované pakety
conda search jmenoPaketu	hledá paket nebo jemu podobné
conda install jmenoPaketu	nainstaluje paket
conda update jmenoPaketu	aktualizuje verzi paketu
conda update python	aktualizuje verzi Pythonu (t.j. Python interpretu)

9 Knihovny NumPy a SciPy

Knihovny ***NumPy***, ***SciPy***, ***pandas*** a ***Matplotlib*** jsou klíčové pro vědecké použití Pythonu a pro analýzu dat. Knihovna numpy, numeric Python, umožňuje tvorbu a efektivní manupilaci s vektory a maticemi a poskytuje metody lineární algebry. Práce s numpy má „MATLAB“ styl. Umožňuje „rozumně“ užívat běžné operace, např. „+“ pro sčítání vektorů apod. Knihovna ***scipy***, scientific Python, doplňuje řadu speciální funkcí užívaných ve vědeckých výpočtech, jako jsou problémy minimalizace, regrese nebo Fourierova transformace.

9.1 arange a linspace

Sekvence čísel „od-do“ je v numpy možné vytvořit pomocí příkazů `arange` a `linspace`. Odpovídají příkazu `range` v základním Pythonu, ale výsledek je jiného datového typu i povahy. Výsledkem `range` je objekt třídy range, sekvence, kdy lze volat jednotlivé elementy, ale nelze např. sčítat 2 range objekty. Naproti tomu `arange` a `linspace` jsou třídy ***numpy.ndarray*** a podporují řadu užitečných operací, jako např. vektorové sčítání, násobení maticemi apod.

`arange(a,b,x)` vrátí sekvenci hodnot od ***a*** včetně do méně než ***b*** („a inclusive, b exclusive“) vzdálených od sebe o x. Naproti tomu `linspace(a,b,n)` vrátí rovnoměrně rozdělenou sekvenci n hodnot od ***a*** do ***b*** včetně.

```
>>> import numpy as np

>>> print(range(5,19))
>>> print(type(range(5,19)))          # objekt třídy range
>>> print(np.arange(5, 19))           # sekvence od 5 do 18 po jedné
>>> print(type(np.arange(5,19)))      # objekt třídy numpy.ndarray
>>> print( np.arange(5, 19, 3.6))     # sekvence od 5 do 18 po 3.6
>>> print( np.linspace(5, 19, 10) )   # sekvence 10 čísel od 5 do 18
range(5, 19)
<class 'range'>
[ 5  6  7  8  9 10 11 12 13 14 15 16 17 18]
<class 'numpy.ndarray'>
[ 5.   8.6 12.2 15.8]
[ 5.  6.55555556  8.11111111  9.66666667 11.22222222 12.77777778
14.33333333 15.88888889 17.44444444 19.
```

```
>>> x = np.arange(5, 19)
>>> print(x[-1]) # výběr posledního prvku sekvence
18
```

Linspace vlastně není vlastní funkce, ale pouze zjednodušení použití funkce arange. „uvnitř“ volá funkci `arange(a,b+1,(b-a)/(n-1))`. Takovou funkci, která tvoří pouze „vnější obal“ jiné funkce, označujeme jako ***wrapper***. Výsledný datový typ je v obou případech stejný, array.

```
>>> print(np.linspace(0,10,10))
>>> print(np.arange(0,11,10/9))
array([ 0.        ,  1.11111111,  2.22222222,  3.33333333,  4.44444444,
```

```
5.55555556,  6.66666667,  7.77777778,  8.88888889, 10.        ])
array([ 0.        ,  1.11111111,  2.22222222,  3.33333333,  4.44444444,
5.55555556,  6.66666667,  7.77777778,  8.88888889, 10.        ])
```

NumPy pracuje podstatně efektivněji, než kdyby stejná úloha byla řešena pomocí základních datových struktur a příkazů Pythonu, jako jsou list nebo cyklus. Např. sečtení 2 dlouhých vektorů je v NumPy 50-krát rychlejší.

```
>>> import numpy as np
>>> import time

>>> def normal_array(): # řešení pomocí listu a cyklu
>>>     t = time.time() # měření času
>>>     x, y = range(10000000), range(10000000) # definice vektorů
>>>     z = [] # prázdný list
>>>     for i in range(len(x)):
>>>         z.append(x[i] + y[i]) # postupné přidávání součtu do listu z
>>>     print(time.time() - t)
>>>     return()

>>> def numpy_array():  # řešení pomocí numpy
>>>     t = time.time()
>>>     x, y = np.arange(10000000), np.arange(10000000)
>>>     z = x + y # součet vektorů pomocí +, t.j. z je vektor
>>>     print(time.time() - t)
>>>     return()

>>> normal_array()
>>> numpy_array()
2.722107410430908
0.04695248603820801 # 58-krát rychlejší výpočet
```

9.2 Vektory a matice

Nejdůležitějším objektem NumPy, kolem něhož je vše ostatní vystavěno, je ***ndarray***, n-rozměrné pole (n-dimensional array). Podle počtu dimenzí pak výsledek označujeme jako skalár (0 dimenzí), vektor (1 dimenze), matice (2 dimenzí), 3D pole (3 dimenzí) apod. Podobně jako u obdobných objektů v matematice musí být datový typ všech prvků ndarray stejný. Tím se liší od datové tabulky (dataframe).

```
>>> import numpy as np
>>> x = np.array(19)                    # 0 dimenzí = skalár
>>> print(type(x))
>>> y = np.array([5,19,36,69,119,149])  # 1 dimenze = vektor
>>> z = np.array([1.2, 3.4, 5.6, 7.8, 9.0])
>>> print(np.ndim(x)) # number of dimensions
>>> print(np.ndim(y))
>>> print(y.dtype)    # typ elementů - integer, float ...
>>> print(z.dtype)
<class 'numpy.ndarray'>
0
1
```

```
int64
float64
```

Objekty vyšších dimenzí se vytvoří jako „dimenze dimenzí“, tedy např. matice jako vektor vektorů.

```
>>> x = np.array([ [5.1, 9.3, 6.6], # matice 2x3 = 3 vektory o 3 prvcích
                   [9.1, 1.9, 1.4]])
>>> print(x.ndim)

>>> y = np.array([[1,2,3],          # matice 6x3
                  [4,5,6],
                  [7,8,9],
                  [0,1,2],
                  [3,4,5],
                  [6,7,8]])
>>> print(np.shape(y))              # ukáže rozměry matice
2
(6, 3)
```

```
>>> y.reshape(3,6)                  # změna tvaru matice
>>> print(np.shape(y.reshape(3,6)))
(3, 6)
```

```
>>> y.reshape(2,3,3)                # změna na 3D matici
array([[[1, 2, 3],                  # vektor 2 2D matic
        [4, 5, 6],
        [7, 8, 9]],
       [[0, 1, 2],
        [3, 4, 5],
        [6, 7, 8]]])
```

9.2.1 Přístupy na prvky pole

Elementy všech dimenzí jsou indexovány od 0.

```
>>> x = np.arange(1,13).reshape(3,4) # jak jednoduše vytvořit matici
>>> print(x)
>>> print( np.shape(x) )
>>> print(x[1,1])  # počitání elementů od 0
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
(3, 4)
6
```

„:“ znamená volbu všech řádků nebo sloupců

```
>>> x[:,1]      # všechny řádky
>>> x[2,:]      # všechy sloupce
>>> x[1:3,1:3] # sloupce i řádky 1 až 2 (nikoli 3!!)
array([ 2,  6, 10])
array([ 9, 10, 11, 12])
array([[ 6,  7],
       [10, 11]])
```

```
>>> x[[0,1,2,1],[0,1,2,3]] # výběr prvků pomocí listy indexů
                           # zde vybrány prvky [0,0], [1,1], [2,2] a [1,3]
>>> x[::3,1::2] # každý 3. řádek od 0–tého, každý 2. sloupec od prvního
array([ 1,  6, 11,  8])
array([[2, 4]])
```

Metoda ix_ vybere submatici pomocí indexu řádků a sloupců

```
>>> x[np.ix_([0,2],[0,3])] # elementy z řádku 0 a 2 a sloupců 0 a 3
array([[ 1,  4],
       [ 9, 12]])
```

Výběr elementů podmínkou

```
>>> x[x % 2 == 0]   # vrátí 1D vektor se sudými prvky
>>> x[ x[:,1]>3 ,:] # všechny sloupce, řádky ty, pro něž je hodnota v 1. sloupci
array([ 2,  4,  6,  8, 10, 12])
array([[ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

9.2.2 Povrchová a hluboká kopie matice

Výběr submatice pomocí indexů v NumPy nevytváří nový objekt, pouze odkazuje na část puvodní matice. Jedná se o tzv. **povrchovou kopii**, adresa jednotlivých pravků v paměti je stejná. Ta šetří paměť počitače, ale mění původní matici při změně matice „nové“. Naproti tomu standardní Python při výběru prvků vytváří nový objekt, tzv. **hlubokou kopii**. Adresa prvků v paměti je odlišná.

```
# hluboká kopie ve standardním Pythonu
>>> y = [4,18,35,68,118,148]
>>> y2 = y[:3] # nový objekt
>>> print( y )
>>> print( y2 )
>>> y2[1] = 19 # puvodní objekt se nemění
>>> print( y )
>>> print( y2 )

[4, 18, 35, 68, 118, 148]
[4, 18, 35]
[4, 18, 35, 68, 118, 148]
[4, 19, 35]
```

```
# povrchová kopie v NumPy
>>> x = np.array([4,18,35,68,118,148])
>>> x2 = x[:3] # reference na původní objekt
>>> print( x )
>>> print( x2 )
>>> x2[1] = 19 # původní objekt se změní též
>>> print( x )
>>> print( x2 )

[4, 18, 35, 68, 118, 148]
[4, 18, 35]
[4, 19, 35, 68, 118, 148]
```

```
[4, 19, 35]
```

NumPy metoda `element.data` ukáže adresu v paměti. `may_share_memory()` zjistí, zda se paměťové oblasti dvou objektů překrývají.

```
>>> x[0].data
>>> x2[0].data
>>> np.may_share_memory(x, x2)
<memory at 0x0000020E00018680> # stejná adresa
<memory at 0x0000020E00018680> # povrchová kopie
True
```

Funkce `copy()` vytvoří hlubokou kopii.

```
>>> x = np.array([4,18,35,68,118,148])
>>> x2 = np.ndarray.copy(x)
>>> x2[1] = 19
>>> print(x)
>>> print(x2)
>>> np.may_share_memory(x, x2)
[  4  18  35  68 118 148]
[  4  19  35  68 118 148]
False
```

9.2.3 tvorba specialnich matic (jednotkova, nahodna . . .)

```
>>> np.zeros(5)
>>> np.ones((4,5), dtype=int)

>>> np.identity(4)
>>> np.eye(4,5,0, dtype=np.int64)   1 na hlavni diagonale

>>> x = np.arange(1,13).reshape(2,2,3)
>>> np.zeros\PYZus{like}(x, dtype=np.float64)   vytvori matici stejneho formatu

>>> np.random.randint(0, 10, (5,5))
>>> np.random.poisson(10, (5,5))
array([[10, 15,  9,  8,  4],
       [10, 12,  4, 13, 13],
       [10, 10, 14, 10, 11],
       [ 9, 10, 19, 13, 12],
       [ 9, 11,  7, 10, 11]])
```

9.2.4 Operace nad arrays

```
>>> x = np.array([5.1, 9.3, 6.6, 9.1, 1.9, 1.4, 9.0])
```

```
        x  2.3
```

```
>>> array([ 2.8,  7. ,  4.3,  6.8, -0.4, -0.9,  6.7])
```

```
>>> 6*x
```

```
        array([30.6, 55.8, 39.6, 54.6, 11.4,  8.4, 54. ])

>>> x = np.arange(9).reshape(3,3)

>>> x

        array([[0, 1, 2],
        [3, 4, 5],
        [6, 7, 8]])

>>> x + np.arange(9,18).reshape(3,3)

array([[ 9, 11, 13],
        [15, 17, 19],
        [21, 23, 25]])

>>> x * 1.9

        array([[ 0. ,  1.9,  3.8],
        [ 5.7,  7.6,  9.5],
        [11.4, 13.3, 15.2]])

>>> x + np.array([1,2,3])    chybejici dimenze se automaticky doplni

        array([[ 1,  3,  5],
        [ 4,  6,  8],
        [ 7,  9, 11]])

>>> x = np.arange(9).reshape(3,3)
>>> y = np.arange(9,18).reshape(3,3)
>>> x * y        nasobeni matic komponentneweise
array([[  0,  10,  22],
       [ 36,  52,  70],
       [ 90, 112, 136]])

>>> np.dot(x,y)  radne nasobeni matic
array([[ 42,  45,  48],
       [150, 162, 174],
       [258, 279, 300]])

OPERACE PO DIMENZICH
>>> x = np.array([[2,3,1,9],[5,6,12,8],[1,2,5,0]])
>>> x
array([[ 2,  3,  1,  9],
       [ 5,  6, 12,  8],
       [ 1,  2,  5,  0]])

>>> np.max(x)
12

>>> np.max(x, 0)  maxima z 0te dimenze
array([ 5,  6, 12,  9])
```

```
>>> np.max(x, 1)  maxima z 1te dimenze
array([ 9, 12,  5])

>>> np.sum(x, (0))  sloupcove soucty
array([ 8, 11, 18, 17])

>>> np.all(x {} 19)
False

>>> np.any(x  1, 1)   neco z radku mensi nez 1?
array([False, False,  True])

>>> np.transpose(x)  == x.T
array([[ 2,  5,  1],
       [ 3,  6,  2],
       [ 1, 12,  5],
       [ 9,  8,  0]])

>>> x.T
array([[ 2,  5,  1],
       [ 3,  6,  2],
       [ 1, 12,  5],
       [ 9,  8,  0]])

# np.concatenate()  .... jako rbind() a cbind()
>>> x = np.array([[2,3,1,9],[5,6,12,8],[1,2,5,0]])
>>> y = np.arange(9,18).reshape(3,3)
>>> np.concatenate((x,y),1)
array([[ 2,  3,  1,  9,  9, 10, 11],
       [ 5,  6, 12,  8, 12, 13, 14],
       [ 1,  2,  5,  0, 15, 16, 17]])

>>> np.concatenate((x,y),0)  # jine rozmery, nelze
```

---------- ValueError Traceback (most recent call last) <ipython-input-161-0faf6e832fa3> in <module> ----> 1np.concatenate((x,y),0) # jine rozmery, nelze

<__array_function__ internals> in concatenate(*args, **kwargs)

ValueError: all the input array dimensions for the concatenation axis must match exactly, but along dimension 1, the array at index 0 has size 4 and the array at index 1 has size 3

```
>>> np.stack()  vyvori 3D array = 2 2D za sebe
>>> x = np.arange(10).reshape(2,5)
>>> y = np.arange(10).reshape(2,5)
>>> np.stack((x,y))
array([[[0, 1, 2, 3, 4],
        [5, 6, 7, 8, 9]],
       [[0, 1, 2, 3, 4],
        [5, 6, 7, 8, 9]]])

>>> np.newaxis  rozsiri array o osu
>>> x = np.array([5,19,36,69,119,149])
>>> y = x[:,np.newaxis]
```

```
>>> y
array([[  5],
       [ 19],
       [ 36],
       [ 69],
       [119],
       [149]])
```

```
>>> np.sort()  sorting po osach
>>> x = np.array([[2,3,1,9],[5,6,12,8],[1,2,5,0]])
>>> x
array([[ 2,  3,  1,  9],
       [ 5,  6, 12,  8],
       [ 1,  2,  5,  0]])
```

```
>>> np.sort(x)  v radcich
array([[ 1,  2,  3,  9],
       [ 5,  6,  8, 12],
       [ 0,  1,  2,  5]])
```

```
>>> np.sort(x, 0)  ve sloupcich
array([[ 1,  2,  1,  0],
       [ 2,  3,  5,  8],
       [ 5,  6, 12,  9]])
```

9.3 Linearni Algebra s NumPy

9.3.1 trida matrix

- wrapper okolo array
- rada lin. alg. algoritmu se da uplatnit na matrix a array

```
>>> x = np.matrix([[2,3,1],[5,6,12],[1,2,5]])
>>> x
matrix([[ 2,  3,  1],
        [ 5,  6, 12],
        [ 1,  2,  5]])
```

```
>>>     x*x  radne maticove nasobeni
matrix([[ 20,  26,  43],
        [ 52,  75, 137],
        [ 17,  25,  50]])
```

```
>>> x**3  radne maticove mocneni
matrix([[ 213,  302,  547],
        [ 616,  880, 1637],
        [ 209,  301,  567]])
```

```
>>> x = np.matrix([[2,3,1],[5,6,12],[1,2,5]])
>>> np.linalg.eig(x)
(array([11.82016663, -0.92462244,  2.10445581]),
```

```
matrix([[-0.30650748, -0.68940485, -0.88797034],
        [-0.900331  ,  0.71359219, -0.17296339],
            [-0.30896158, -0.12452769,  0.42613652]]))
```

```
>>> np.linalg.det(x)
-23.0
```

```
>>> np.linalg.inv(x)
matrix([[-0.26086957,  0.56521739, -1.30434783],
        [ 0.56521739, -0.39130435,  0.82608696],
        [-0.17391304,  0.04347826,  0.13043478]])
```

```
# soustavy linearnich rovnic

>>> variablen = np.array([[1,3],[3,4]])
>>> koeffizienten = np.array([30, 20])
>>> loesung = np.linalg.solve(variablen, koeffizienten)
>>> loesung
array([-12.,  14.])
```

9.4 SciPy

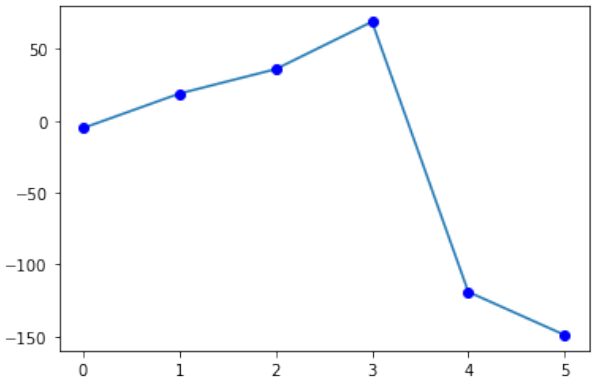
rada matematickych a statistickych funkci

```
>>> import scipy.stats as st

>>> N = st.norm(5, 2)         normalni rozdeleni s u=5 a sigma=2
>>> N.cdf(2)                  (kumulativni) distribucni funkce N
>>> N.pdf([1,5,7])            hustota pravdepodobnosti  probability density funct
>>> P = st.pareto(1, 0, 2)    Paretovo rozdeleni
>>> a = P.rvs(100)            gereruje 100 cisel z rozdeleni P

>>> parameter = st.pareto.fit(someData) \PYZsh{ nauci se parametry rozdeleni z n
```

```
>>> N.pdf([1,5,7])
array([0.02699548, 0.19947114, 0.12098536])
```

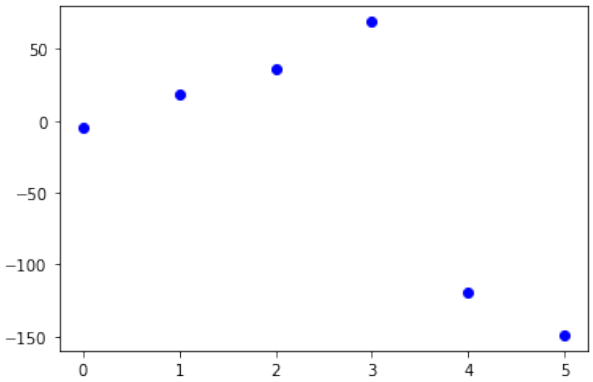
10 13. Matplotlib

```
>>> import matplotlib.pyplot as plt
>>> plt.plot([5, 19, 36, 69, 119, 149])    hodnoty interpretuje jako y
>>> plt.show()
```



Obrázek 10.1: Datové typy Pythonu

```
>>> plt.plot([5, 19, 36, 69, 119, 149], ob)  formatovani
>>> plt.show()
```



Obrázek 10.2: Datové typy Pythonu

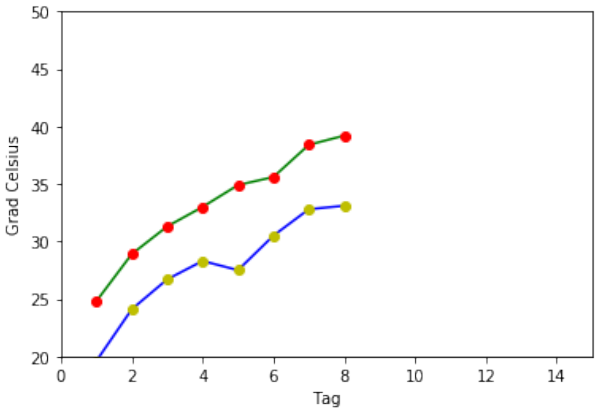
FORMATOVANI

```
-   souvisla cara
--  carkovane
-.  cerchovane
,   marker pixelu
```

```
o    marker kruhu
x    X-marker
*    marker *
D    marker kosoctverec
v    marker trojuhelnih

b    modra
g    zelena
r    cervena
k    cerna
```

10.0.1 Estetika grafu, osy, meritka ...

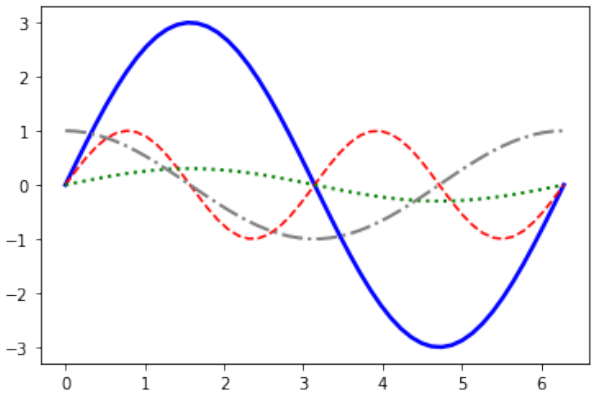
```
        x,y plot

>>> tage = list(range(1,9))
>>> plt.ylabel(Grad Celsius)
>>> celsiusMin = [19.6, 24.1, 26.7, 28.3, 27.5, 30.5, 32.8, 33.1]
>>> celsiusMax = [24.8, 28.9, 31.3, 33.0, 34.9, 35.6, 38.4, 39.2]
>>> plt.xlabel(Tag)
>>> plt.ylabel(Grad Celsius)
>>> plt.plot(tage, celsiusMin, b, tage, celsiusMin, oy, tage, celsiusMax, g, tag
>>> plt.axis()
>>> plt.axis([0, 15, 20, 50])  pokud chybi, nastavi se automaticky
>>> plt.show()
```



Obrázek 10.3: Datové typy Pythonu

```
>>> import matplotlib.pyplot as plt
>>> X = np.linspace(0, 2 * np.pi, 50, endpoint=True)
>>> F1 = 3 * np.sin(X)    pise se jako matematicka funkce
>>> F2 = np.sin(2*X)
>>> F3 = 0.3 * np.sin(X)
>>> F4 = np.cos(X)
>>> plt.plot(X, F1, color=blue, linewidth=2.5, linestyle=)
>>> plt.plot(X, F2, color=red, linewidth=1.5, linestyle=)  lze do grafu pridavat
>>> plt.plot(X, F3, color=green, linewidth=2, linestyle=:)
>>> plt.plot(X, F4, color=grey, linewidth=2, linestyle=.)
```

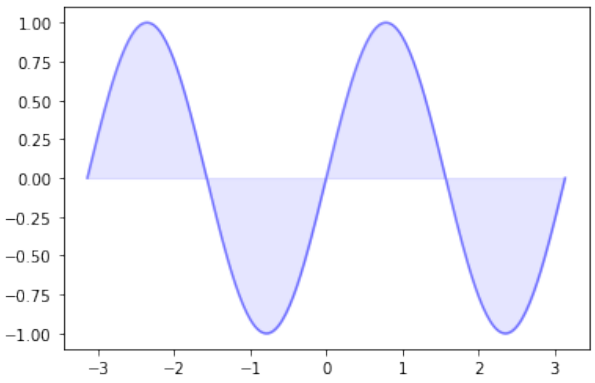
```
>>> plt.show()
```



Obrázek 10.4: Datové typy Pythonu

fill_between() ... vyplni plochu mezi grafem a osou

```
>>> n = 256
>>> X = np.linspace(np.pi,np.pi,n)
>>> Y = np.sin(2*X)
>>> plt.plot(X, Y, color=blue, alpha=0.50)    alpha nastavuje intenzitu modrosti
>>> plt.fill_between(X, 0, Y, color=blue, alpha=.1)
>>> plt.show()
```

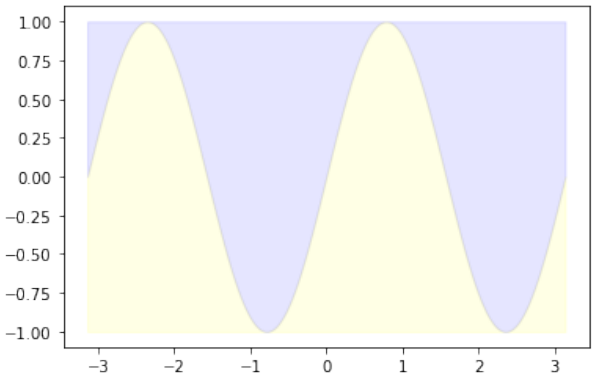


Obrázek 10.5: Datové typy Pythonu

```
>>> plt.fill_between}X, Y, 1, color=blue,alpha=.1)
>>> plt.fill_between(X, 1, Y, color=yellow,alpha=.1)
>>> plt.show()
```

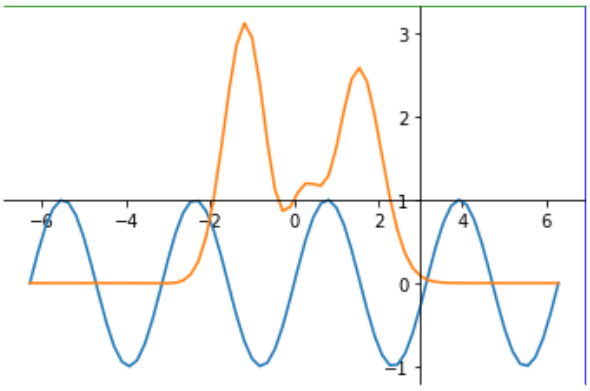
gca() - lze libovolne nastavit OSY a MERITKO

```
>>> import matplotlib.pyplot as plt

>>> X = np.linspace(2 * np.pi, 2 * np.pi, 70, endpoint=True)

>>> F1 = np.sin(2* X)
>>> F2 = (2*X**5 + 4*X**4  4.8*X**3 + 1.2*X**2 + X + 1)*np.exp(X**2)
```



Obrázek 10.6: Datové typy Pythonu

```
>>> ax = plt.gca()  vytvori referenci na objekt os

>>> ax.spines[top].set\PYZus{color}(green)      spices  vyznaceni car \PYZhy{ ok
>>> ax.spines[right].set\PYZus{color}(blue)
>>> ax.xaxis.set\PYZus{ticks\PYZus{}position}(bottom)    tick  vyznaceni markeru

>>> ax.spines[bottom].set\PYZus{position}((data,1))
>>> ax.yaxis.set\PYZus{ticks\PYZus{}position}(left)
>>> ax.spines[left].set\PYZus{position}((data,3))

>>> plt.plot(X, F1)
>>> plt.plot(X, F2)
>>> plt.show()
```

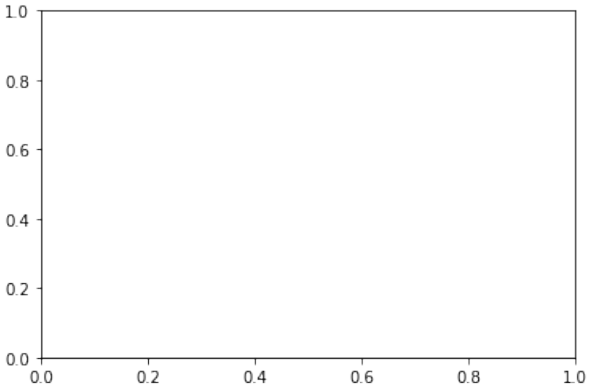


Obrázek 10.7: Datové typy Pythonu

xticks() a yticks() - nastaveni odsazeni markeru polohy

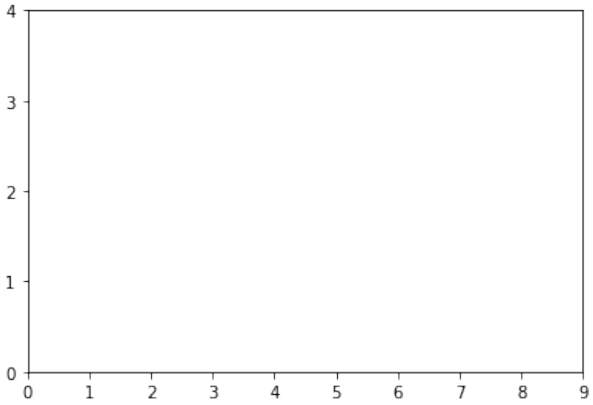
```
>>> ax = plt.gca()
>>> locs, labels = plt.xticks()
>>> print(locs, labels)
>>> plt.show()
[0.  0.2 0.4 0.6 0.8 1. ] <a list of 6 Text xticklabel objects>
```

```
>>> ax = plt.gca()
```



Obrázek 10.8: Datové typy Pythonu

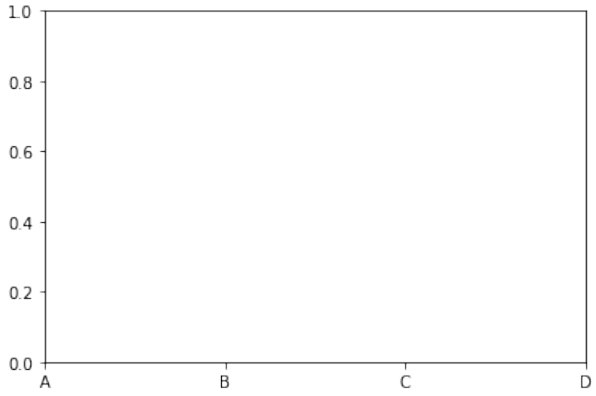
```
>>> plt.xticks(np.arange(10))
>>> plt.yticks(np.arange(5))
>>> plt.show()
```



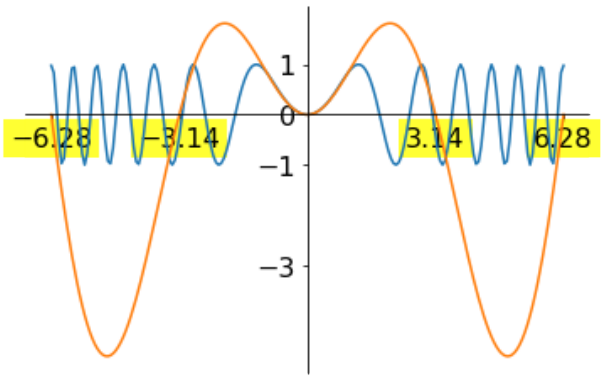
Obrázek 10.9: Datové typy Pythonu

```
>>> ax = plt.gca()
>>> plt.xticks(np.arange(4), (A,B,C,D))
>>> plt.show()
```

```
>>> import matplotlib.pyplot as plt
>>> X = np.linspace(2 * np.pi, 2 * np.pi, 200, endpoint=True)
>>> F1 = np.sin(X**2)
>>> F2 = X * np.sin(X)
>>> ax = plt.gca()
>>> ax.spines[top].set_color(none)
>>> ax.spines[right].set_color(none)
>>> ax.xaxis.set_ticks_position(bottom)
>>> ax.spines[bottom].set_position((data,0))
>>> ax.yaxis.set_ticks_position(left)
>>> ax.spines[left].set_position((data,0))

>>> plt.xticks( [6.28, 3.14, 3.14, 6.28])
>>> plt.yticks([3, 1, 0, +1, 3])
```



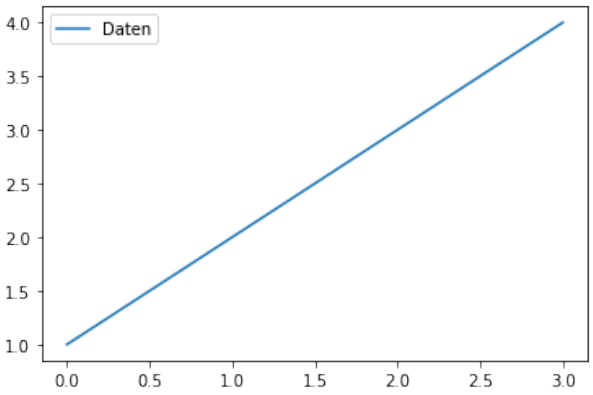

```
>>> for xtick in ax.get_xticklabels():
>>>     xtick.set_fontsize(16)
>>>     xtick.set_bbox(dict(facecolor=yellow,edgecolor=None, alpha = 0.8 ))
>>> for ytick in ax.get_yticklabels():
>>>     ytick.set_fontsize(16)
>>>     ytick.set_bbox(dict(facecolor=white,edgecolor=None, alpha = 0.8 ))

>>> plt.plot(X, F1)
>>> plt.plot(X, F2)

>>> plt.show()
```

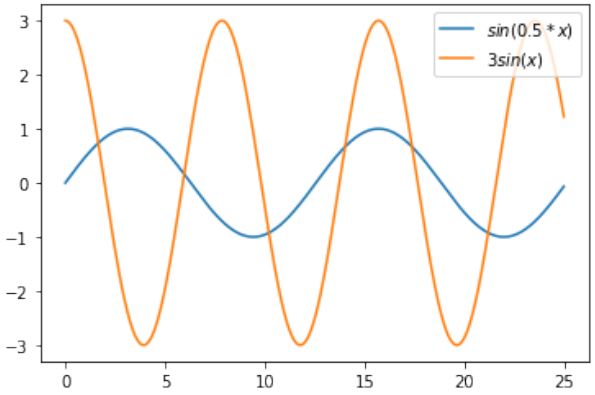


legenda

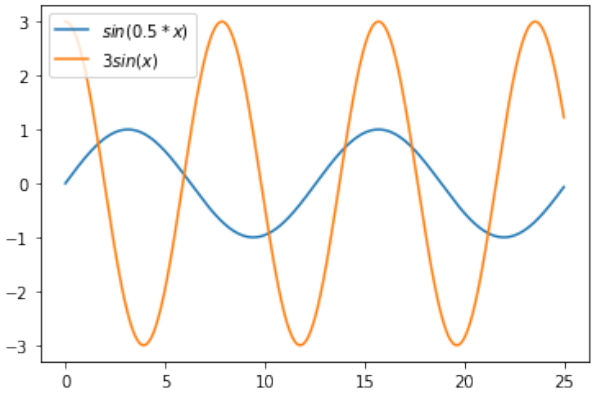
```
>>> ax = plt.gca()
>>> ax.plot([1, 2, 3, 4])
>>> ax.legend([Daten])
>>> plt.show()
```

```
>>> X = np.linspace(0, 25, 1000)
>>> F1 = np.sin(0.5 * X)
>>> F2 = 3 * np.cos(0.8*X)
>>> plt.plot(X, F1, label='$sin(0.5 * x)$')
>>> plt.plot(X, F2, label='$3 sin(x)$')
>>> plt.legend(loc=upper right)
>>> plt.show()
```





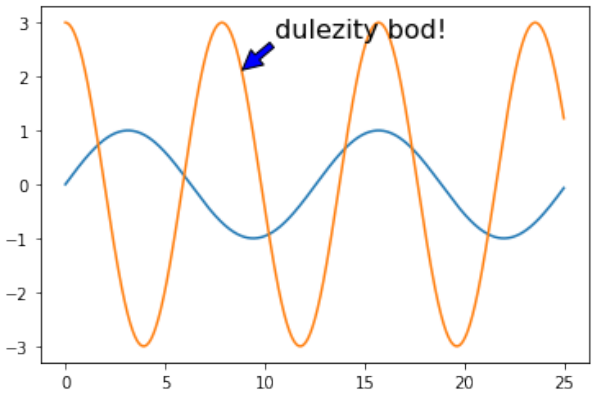

```
>>> X = np.linspace(0, 25, 1000)
>>> F1 = np.sin(0.5 * X)
>>> F2 = 3 * np.cos(0.8*X)
>>> plt.plot(X, F1, label='$sin(0.5 * x)$')
>>> plt.plot(X, F2, label='$3 sin(x)$')
>>> plt.legend(loc=best)
>>> plt.show()
```



annotate() - popisky grafu

```
>>> import matplotlib.pyplot as plt

>>> X = np.linspace(0, 25, 1000)
>>> F1 = np.sin(0.5 * X)
>>> F2 = 3 * np.cos(0.8*X)
```

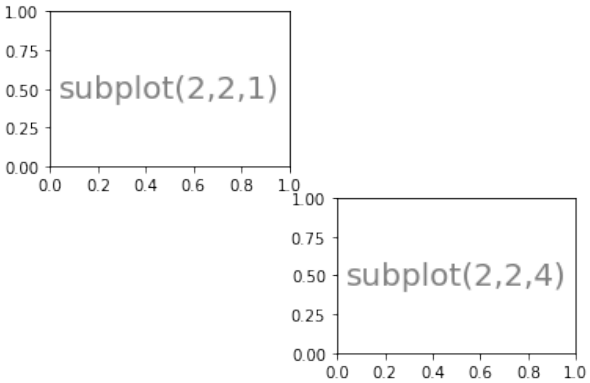
```
>>> plt.plot(X, F1)
>>> plt.plot(X, F2)

>>> plt.annotate(dulezity bod!,
>>> xy=(11.3*np.pi/4, 3*np.sin(3*np.pi/4)),
>>> xycoords=data,
>>> xytext=(+20, +20),
>>> textcoords=offset points,
>>> fontsize=16,
>>> arrowprops=dict(facecolor=blue))
>>> plt.show()
```



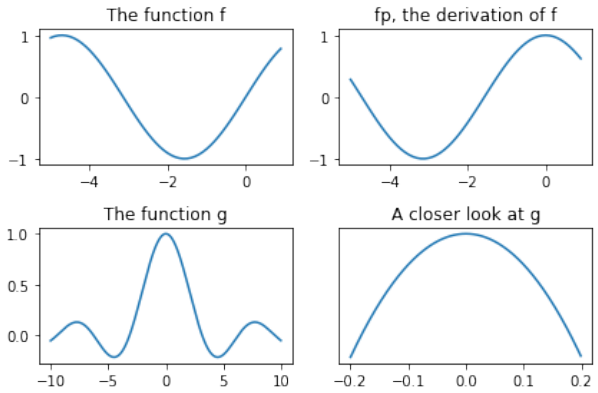
10.1 subplots

subplots(nrows, ncols, plotNumber)

```
>>> import matplotlib.pyplot as plt

>>> plt.figure(figsize=(6, 4))
>>> plt.subplot(221)
>>> plt.text(0.5, 0.5, subplot(2,2,1),horizontalalignment=center,verticalalignme
>>> plt.subplot(224)
>>> plt.text(0.5, 0.5, subplot(2,2,4),ha=center,va=center, fontsize = 20, alpha=
>>> plt.show()
```



```
>>> import matplotlib.pyplot as plt

>>> def fun1(x):
>>>     return(np.sin(x))

>>> def fun2(x):
>>>     return(np.cos(x))

>>> def fun3(x):
>>>     return(np.sin(x)/x)

>>> fig = plt.figure(figsize=(6, 4))

>>> sub1 = fig.add_subplot}(221)
>>> sub1.set_title}(The function f)
>>> t = np.arange(5.0, 1.0, 0.1)
>>> sub1.plot(t, fun1(t))

>>> sub2 = fig.add_subplot}(222) , axisbg=grey)
>>> sub2.set_title}(fp, the derivation of f)
>>> sub2.plot(t, fun2(t))

>>> sub3 = fig.add_subplot}(223)
>>> sub3.set_title}(The function g)
>>> t = np.arange(10.0, 10.0, 0.02)
>>> sub3.plot(t, fun3(t))

>>> sub4 = fig.add_subplot}(224)  , axisbg=grey)
>>> sub4.set_title}(A closer look at g)
>>> sub4.set_xticks}([0.2, 0.1, 0, 0.1, 0.2])
>>> sub4.set_yticks}([0.15, 0.1, 0, 0.1, 0.15])
>>> t = np.arange(0.2, 0.2, 0.001)
>>> sub4.plot(t, fun3(t))

>>> plt.tight_layout}()
>>> plt.show()
```



11 Pandas

= datova knihovna pro analyzu dat, ktera lze reprezentovat 2D tabulkou (co umi excel, umi i pandas)

zakladni datovy typ je DataFrame = tabulka

DataFrame je prakticky list stejne dlouhych listu

https://pandas.pydata.org/ - vcetne obsahle dokumentace

```
# conda install pandas
>>> import pandas
```

11.1 Datove typy v Pandas

11.1.1 DataFrame = tabulka

stejne jako data frames v R

```
# nacteni z csv souboru

>>> actors = pandas.read_csv}(static/actors.csv, index_col}=None)
>>> actors
name  birth  alive
0    Terry   1942  False
1  Michael   1943   True
2     Eric   1943   True
3   Graham   1941  False
4    Terry   1940   True
5     John   1939   True
```

```
# vytvoreni df z listu listu (= seznamu seznamu)

>>> items = pandas.DataFrame([
             [Book, 123],          prvni radek
             [Computer, 2185],     druhy radek
                             ])
>>> items
0     1
0      Book   123
1  Computer  2185
```

```
# vytvoreni df z seznamu slovniku (= list of dictionaries)
# kazdy sloupec ma jmeno

>>> items = pandas.DataFrame([
             {}name: Book, price: 123{},          prvni radek
             {}name: Computer, price: 2185{},     druhy radek
```

```
                                ])
>>> items

name  price
0      Book    123
1  Computer   2185
```

```
>>> actors.info()   info o tabulce
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6 entries, 0 to 5
Data columns (total 3 columns):
\#   Column  Non-Null Count  Dtype
---  ------  --------------  -----
0   name    6 non-null      object
1   birth   6 non-null      int64
2   alive   6 non-null      bool
dtypes: bool(1), int64(1), object(1)
memory usage: 230.0+ bytes
```

11.1.2 Series = sloupce

jako vector v R (c(1,2,3) == [1,2,3])

list hodnot, ale na rozdil od sloupce ma jmeno, datovy typ a index, ktery jednotlive hodnoty pojmenovava

```
# tvorba sloupce z listu
>>> type(pandas.Series([1,2,3,4,5]))
pandas.core.series.Series
```

```
>>> type([1,2,3,4,5])
list
```

```
# tvorba sloupce vybranim z tabulky DataFrame
>>> birth_years} = actors[birth]
>>> birth_years}
0    1942
1    1943
2    1943
3    1941
4    1940
5    1939
Name: birth, dtype: int64
```

```
>>> type(birth_years})
pandas.core.series.Series
```

```
>>> birth_years}.name
'birth'
```

```
>>> birth_years}.index
RangeIndex(start=0, stop=6, step=1)
```

```
>>> birth_years}.dtype
dtype('int64')
```

operace jako v R, t.j. nad kazdou hodnotou jednotlive. . . .vznikne novy sloupec

aritmetické operace (+, -, , /, //, %, **) nebo porovnávání <> ==

```
>>> ages = 2016  birth_years}
>>> ages
0    74
1    73
2    73
3    75
4    76
5    77
Name: birth, dtype: int64
```

```
>>> birth_years} {} 1940
0     True
1     True
2     True
3     True
4    False
5    False
Name: birth, dtype: bool
```

scitani dvou sloupcu jako vektorove scitani

```
>>> actors[name] + [ (1),  (2),  (3),  (4),  (5),  (6)]
0      Terry (1)
1    Michael (2)
2       Eric (3)
3     Graham (4)
4      Terry (5)
5       John (6)
Name: name, dtype: object
```

```
>>> actors[birth] + [1,20,3000,40000,500000,6000000]
0       1943
1       1963
2       4943
3      41941
4     501940
5    6001939
Name: birth, dtype: int64
```

```
# funguji vsechny retezcove operace

>>> actors[name].str.upper()
0      TERRY
1    MICHAEL
2       ERIC
3     GRAHAM
4      TERRY
5       JOHN
Name: name, dtype: object
```

operace s daty a casy (**datetime**) pod dt

```
>>> birth_years}[2]  vyber prvku jak z listu
1943
```

```
>>> birth_years}[2:2]

2    1943
3    1941
Name: birth, dtype: int64
```

```
>>> birth_years}[birth_years} {} 1940]  boolovsky vyber
0    1942
1    1943
2    1943
3    1941
Name: birth, dtype: int64
```

```
>>> birth_years}[(birth_years} {} 1940) \PYZam{} (birth_years}  1943)]
standardni boolovske operatory
0    1942
3    1941
Name: birth, dtype: int64
```

Operace se sloupci

VELKE mnozstvi preddefinovanych operaci se sloupci

```
>>> print(Součet: , birth_years}.sum())
>>> print(Průměr: , birth_years}.mean())
>>> print(Medián: , birth_years}.median())
>>> print(Počet unikátních hodnot: , birth_years}.nunique())
>>> print(Koeficient špičatosti: , birth_years}.kurtosis())
Součet:  11648
Průměr:  1941.3333333333333
Medián:  1941.5
Počet unikátních hodnot:  5
Koeficient špičatosti:  -1.4812500000001654
```

apply - aplikace jakekoli funkce na vsechny hodnoty sloupce

```
>>> actors[name].apply(lambda x: .join(reversed(x)))    jako v R
0      yrreT
1    leahciM
2       cirE
3     maharG
4      yrreT
5       nhoJ
Name: name, dtype: object
```

```
>>> actors[alive].apply({}True: alive, False: deceased{}.get)  get je funkci slo
0    deceased
1       alive
2       alive
```

```
3    deceased
4       alive
5       alive
Name: alive, dtype: object
```

Vyber prvku z tabulky

```
>>> actors[name]   jmenem sloupce
0      Terry
1    Michael
2       Eric
3     Graham
4      Terry
5       John
Name: name, dtype: object
```

```
>>> actors[1:1]   intervalem radku
name  birth  alive
1  Michael   1943   True
2     Eric   1943   True
3   Graham   1941  False
4    Terry   1940   True
```

```
>>> actors[[name, alive]]   seznamem sloupcu
name  alive
0    Terry  False
1  Michael   True
2     Eric   True
3   Graham  False
4    Terry   True
5     John   True
```

alternativne vyber prvku pomoci indexeru **loc** a **iloc**

loc

loc vybira radky podle indexu radku (cisla)

```
>>> actors.loc[2]  hranate zavorky
name     Eric
birth    1943
alive    True
Name: 2, dtype: object
```

```
>>> actors.loc[2, birth]  [sloupce, radky]
1943
```

```
>>> actors.loc[2:4, birth:alive]    obsahuje i obě koncové hodnoty, rozdil oprot
                                     jako indexy v rozsahu lze uzit i textove re
birth  alive
2   1943   True
3   1941  False
4   1940   True
```

kdyz se vybere 1 hodnota, automaticky klesa dimenzionalita (tabulka -> sloupec, sloupec -> skalar)

```
>>> type(actors.loc[2:4, name])  tabulka { sloupec}
pandas.core.series.Series
```

```
>>> actors.loc[:, alive]  : = kompletni interval
0    False
1     True
2     True
3    False
4     True
5     True
Name: alive, dtype: bool
```

```
>>> actors.loc[:, [name, alive]]  indexovani listem hodnot
name  alive
0    Terry  False
1  Michael   True
2     Eric   True
3   Graham  False
4    Terry   True
5     John   True
```

```
>>> actors.loc[[3, 2, 4, 4], :]  radky i sloupce lze preskupit, duplikovat ...
name  birth  alive
3  Graham   1941  False
2    Eric   1943   True
4   Terry   1940   True
4   Terry   1940   True
```

iloc umi totez, co loc, ale indexuje pouze cisly, jako NumPy

```
>>> actors
name  birth  alive
0    Terry   1942  False
1  Michael   1943   True
2     Eric   1943   True
3   Graham   1941  False
4    Terry   1940   True
5     John   1939   True
```

```
>>> actors.iloc[0, 0]
'Terry'
```

```
>>> actors.iloc[1, 1]
1939
```

```
>>> actors.iloc[:, 0:1]
name
0    Terry
1  Michael
2     Eric
3   Graham
```

```
4    Terry
5     John
```

```
>>> actors.iloc[[0, 1, 3], [1, 1, 0]]
alive  birth    name
0  False   1942   Terry
5   True   1939    John
3  False   1941  Graham
```

```
>>> actors.iloc[1].loc[name]  loc a iloc lze kombinovat
'John'
```

14.1.5 Indexy

nazvy radku a sloupcu: index pro radky - **index = rownames v R**, index pro sloupce - **columns = colnames v R**

```
>>> actors.index  jako rownames v R
RangeIndex(start=0, stop=6, step=1)
```

```
>>> actors.columns  jako colnames v R

Index(['name', 'birth', 'alive'], dtype='object')
```

```
>>> actors.index = actors[name]  rownames se da zmenit prirazenim jineho sloupce
>>> actors

name  birth  alive
name
Terry      Terry   1942  False
Michael  Michael   1943   True
Eric        Eric   1943   True
Graham    Graham   1941  False
Terry      Terry   1940   True
John        John   1939   True
```

```
>>> actors.index
Index(['Terry', 'Michael', 'Eric', 'Graham', 'Terry', 'John'], dtype='object',
        name='name')
```

```
>>> actors = actors.sort_index}()  serazeni podle indexu
actors
name  birth  alive
name
Eric        Eric   1943   True
Graham    Graham   1941  False
John        John   1939   True
Michael  Michael   1943   True
Terry      Terry   1942  False
Terry      Terry   1940   True
```

```
>>> actors.loc[[Eric, Graham]]
name  birth  alive
name
Eric      Eric   1943   True
Graham  Graham   1941  False
```

viceurovnovy klic - z vice sloupcu

```
>>> indexed_actors} = actors.set_index}([name, birth])  presunul 2 sloupce do in
            indexed_actors}
alive
name    birth
Eric    1943    True
Graham  1941   False
John    1939    True
Michael 1943    True
Terry   1942   False
1940    True
```

```
>>> indexed_actors}.index
MultiIndex([(   'Eric', 1943),
( 'Graham', 1941),
(   'John', 1939),
('Michael', 1943),
(  'Terry', 1942),
(  'Terry', 1940)],
names=['name', 'birth'])
```

```
>>> indexed_actors}.loc[Terry]
alive
birth
1942   False
1940    True
```

```
>>> indexed_actors}.loc[Terry].loc[1940]
alive    True
Name: 1940, dtype: bool
```

```
>>> indexed_actors}.loc[(Terry, 1940)]
alive    True
Name: (Terry, 1940), dtype: bool
```

pridani sloupce do DataFrame

```
>>> last_names} = pandas.Series([Gilliam, Jones, Cleveland],
     index=[(Terry, 1940), (Terry, 1942), (Carol, 1942)])
        last_names}
(Terry, 1940)      Gilliam
(Terry, 1942)        Jones
(Carol, 1942)    Cleveland
dtype: object
```

```
>>> indexed_actors}[last_name}] = last_names}  vytvoreni noveho sloupce. dle sho
indexed_actors}  Carol, 1942 neexistuje jak oindex
```

```
alive last\_name
name    birth
Eric    1943    True       NaN
Graham  1941   False       NaN
John    1939    True       NaN
Michael 1943    True       NaN
Terry   1942   False     Jones
1940    True   Gilliam
```

NaN = "Not a Number" = NULL v SQL nebo None v Pythonu, informace chybí, není k dispozici nebo nedává smysl

```
>>> indexed_actors}[last_name}].isnull()  isnull() je == NaN ??
name     birth
Eric     1943      True
Graham   1941      True
John     1939      True
Michael  1943      True
Terry    1942     False
1940     False
Name: last\_name, dtype: bool
```

```
>>> indexed_actors}.fillna(0)  NaN nahradi necim jinym
alive last\_name
name    birth
Eric    1943    True         0
Graham  1941   False         0
John    1939    True         0
Michael 1943    True         0
Terry   1942   False     Jones
1940    True   Gilliam
```

```
>>> indexed_actors}.dropna()  vynecha radky obsahujicici NaN
alive last\_name
name  birth
Terry 1942   False     Jones
1940    True   Gilliam
```

11.1.3 14.2 Merge vice DataFrames

DataFrame maji metodu merge(), jako JOIN v SQL. primarne spojeni podle sloupcu
roydil sql a pandas – ***pandas.pydata.org/pandas-docs/stable/comparison_with_sql.html***

```
>>> actors = pandas.read_csv}(static/actors.csv, index_col}=None)
>>> actors
```

```
name  birth  alive
0    Terry   1942  False
1  Michael   1943   True
2     Eric   1943   True
3   Graham   1941  False
4    Terry   1940   True
5     John   1939   True
```

```
>>> spouses = pandas.read_csv}(static/spouses.csv, index_col}=None)
>>> spouses
```

```
name  birth          spouse\_name
0     Graham   1941       David Sherlock
1       John   1939         Connie Booth
2       John   1939     Barbara Trentham
3       John   1939   Alyce Eichelberger
4       John   1939        Jennifer Wade
5      Terry   1940         Maggie Westo
6       Eric   1943           Lyn Ashley
7       Eric   1943       Tania Kosevich
8      Terry   1942        Alison Telfer
9      Terry   1942      Anna Soderstrom
10   Michael   1943        Helen Gibbins
```

```
>>> actors.merge(spouses)
```

```
name  birth  alive          spouse\_name
0      Terry   1942  False        Alison Telfer
#1      Terry   1942  False      Anna Soderstrom
2    Michael   1943   True        Helen Gibbins
3       Eric   1943   True           Lyn Ashley
4       Eric   1943   True       Tania Kosevich
5     Graham   1941  False       David Sherlock
6      Terry   1940   True         Maggie Westo
7       John   1939   True         Connie Booth
8       John   1939   True     Barbara Trentham
9       John   1939   True   Alyce Eichelberger
10      John   1939   True        Jennifer Wade
```

11.1.4 14.3 Preskladani tabulky (“z 2D indexovaneho sloupce do tabulky”)

pozn: **pandas.date_range** - vytvari kalendarni intervaly (cas lze zadavat jako retezec) datetime

```
>>> import pandas
>>> import itertools
>>> import random
>>> random.seed(0)

>>> months = pandas.date_range}(201501, 201612, freq=M)
>>> categories = [Electronics, Power Tools, Clothing]
>>> data = pandas.DataFrame([{}month: a, category: b, sales: random.randint(1000
>>> for a, b in itertools.product(months, categories)
        if random.randrange(20) {} 0])
vytvoreni vetsi tabulky s nahodnymi sales
```

```
>>> months
DatetimeIndex(['2015-01-31', '2015-02-28', '2015-03-31', '2015-04-30',
'2015-05-31', '2015-06-30', '2015-07-31', '2015-08-31',
'2015-09-30', '2015-10-31', '2015-11-30', '2015-12-31',
```

```
'2016-01-31', '2016-02-29', '2016-03-31', '2016-04-30',
'2016-05-31', '2016-06-30', '2016-07-31', '2016-08-31',
'2016-09-30', '2016-10-31', '2016-11-30'],
dtype='datetime64[ns]', freq='M')
```

```
>>> data.head()  head(10), prvnich 10 radku
month     category  sales
0 2015-01-31  Electronics   5890
1 2015-01-31  Power Tools   3242
2 2015-01-31     Clothing   6961
3 2015-02-28  Electronics   3969
4 2015-02-28  Power Tools   4866
```

```
>>> len(data)  pocet radku
67
```

```
>>> data[sales].describe()  zakladni statistika sloupce

count      67.000000
mean     4795.552239
std      3101.026552
min      -735.000000
25\%      2089.000000
50\%      4448.000000
75\%      7874.000000
max      9817.000000
Name: sales, dtype: float64
```

```
>>> indexed = data.set_index}([category, month])  category a month jako index
>>> indexed.head()

sales
category    month
Electronics 2015-01-31   5890
Power Tools 2015-01-31   3242
Clothing    2015-01-31   6961
Electronics 2015-02-28   3969
Power Tools 2015-02-28   4866
```

14.3.1 unstact

snizeni dimenzionality indexu - presune month (t.j. vnitrni index) do radku tabulky i s hodnotami sales

reverzni operace k stack

```
>>> unstacked = indexed.unstack(month)
>>> unstacked

sales                                                          \textbackslash{}
month       2015-01-31 2015-02-28 2015-03-31 2015-04-30 2015-05-31 2015-06-30
category
Clothing        6961.0     2578.0     9131.0      618.0     4796.0
8052.0
```

```
Electronics     5890.0     3969.0     1281.0     7725.0     4409.0
4180.0
Power Tools     3242.0     4866.0     1289.0     1407.0     8171.0
9492.0

{\ldots}             \textbackslash{}
month       2015-07-31 2015-08-31 2015-09-30 2015-10-31  {\ldots} 2016-02-29
category                                                 {\ldots}
Clothing        7989.0        NaN       31.0     7896.0  {\ldots}
4194.0
Electronics     6253.0        NaN     7086.0     8298.0  {\ldots}
6290.0
Power Tools     3267.0     5534.0     2996.0     2909.0  {\ldots}
8769.0

\textbackslash{}
month       2016-03-31 2016-04-30 2016-05-31 2016-06-30 2016-07-31 2016-08-31
category
Clothing        2059.0      471.0     5410.0     8663.0     9817.0
6969.0
Electronics     2966.0     9039.0     1450.0     3515.0     8497.0
349.0
Power Tools     2012.0     6807.0      314.0     2858.0     6382.0
9039.0


month       2016-09-30 2016-10-31 2016-11-30
category
Clothing        -735.0     4448.0     -259.0
Electronics     9324.0      919.0       18.0
Power Tools     2119.0     5095.0     1397.0

[3 rows x 23 columns]
```

```
>>> unstacked.columns

MultiIndex([('sales', '2015-01-31'),
('sales', '2015-02-28'),
('sales', '2015-03-31'),
('sales', '2015-04-30'),
('sales', '2015-05-31'),
('sales', '2015-06-30'),
('sales', '2015-07-31'),
('sales', '2015-08-31'),
('sales', '2015-09-30'),
('sales', '2015-10-31'),
('sales', '2015-11-30'),
('sales', '2015-12-31'),
('sales', '2016-01-31'),
('sales', '2016-02-29'),
('sales', '2016-03-31'),
('sales', '2016-04-30'),
```

```
('sales', '2016-05-31'),
('sales', '2016-06-30'),
('sales', '2016-07-31'),
('sales', '2016-08-31'),
('sales', '2016-09-30'),
('sales', '2016-10-31'),
('sales', '2016-11-30')],
names=[None, 'month'])
```

```
>>> unstacked.columns = unstacked.columns.droplevel()  odstrani sales z indexu p
>>> unstacked

month        2015-01-31  2015-02-28  2015-03-31  2015-04-30  2015-05-31
\textbackslash{}
category
Clothing         6961.0      2578.0      9131.0       618.0      4796.0
Electronics      5890.0      3969.0      1281.0      7725.0      4409.0
Power Tools      3242.0      4866.0      1289.0      1407.0      8171.0

month        2015-06-30  2015-07-31  2015-08-31  2015-09-30  2015-10-31
{\ldots}  \textbackslash{}
category
{\ldots}
Clothing         8052.0      7989.0         NaN        31.0      7896.0
{\ldots}
Electronics      4180.0      6253.0         NaN      7086.0      8298.0
{\ldots}
Power Tools      9492.0      3267.0      5534.0      2996.0      2909.0
{\ldots}

month        2016-02-29  2016-03-31  2016-04-30  2016-05-31  2016-06-30
\textbackslash{}
category
Clothing         4194.0      2059.0       471.0      5410.0      8663.0
Electronics      6290.0      2966.0      9039.0      1450.0      3515.0
Power Tools      8769.0      2012.0      6807.0       314.0      2858.0

month        2016-07-31  2016-08-31  2016-09-30  2016-10-31  2016-11-30
category
Clothing         9817.0      6969.0      -735.0      4448.0      -259.0
Electronics      8497.0       349.0      9324.0       919.0        18.0
Power Tools      6382.0      9039.0      2119.0      5095.0      1397.0

[3 rows x 23 columns]
```

```
>>> type(unstacked)

        pandas.core.frame.DataFrame
```

```
>>> unstacked.loc[Electronics].sum()  ted uz lze analyzovat jako dataframe

        103742.0
```

```
>>> unstacked.loc[[Electronics, Power Tools], 201603:201605]

        month        2016-03-31  2016-04-30  2016-05-31
        category
        Electronics      2966.0      9039.0      1450.0
        Power Tools      2012.0      6807.0       314.0
```

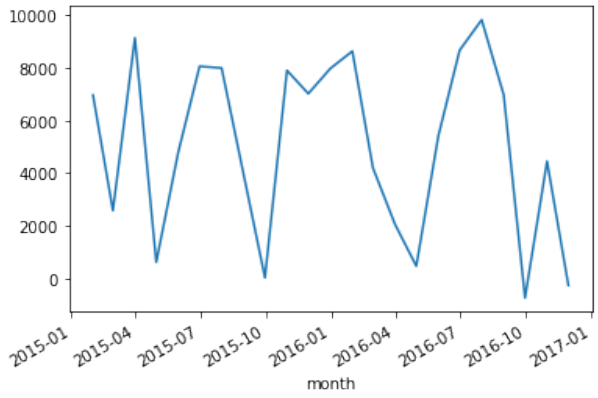
```
>>> unstacked.loc[Clothing]

        month
        2015-01-31    6961.0
        2015-02-28    2578.0
        2015-03-31    9131.0
        2015-04-30     618.0
        2015-05-31    4796.0
        2015-06-30    8052.0
        2015-07-31    7989.0
        2015-08-31       NaN
        2015-09-30      31.0
        2015-10-31    7896.0
        2015-11-30    7016.0
        2015-12-31    7969.0
        2016-01-31    8627.0
        2016-02-29    4194.0
        2016-03-31    2059.0
        2016-04-30     471.0
        2016-05-31    5410.0
        2016-06-30    8663.0
        2016-07-31    9817.0
        2016-08-31    6969.0
        2016-09-30    -735.0
        2016-10-31    4448.0
        2016-11-30    -259.0
        Name: Clothing, dtype: float64
```

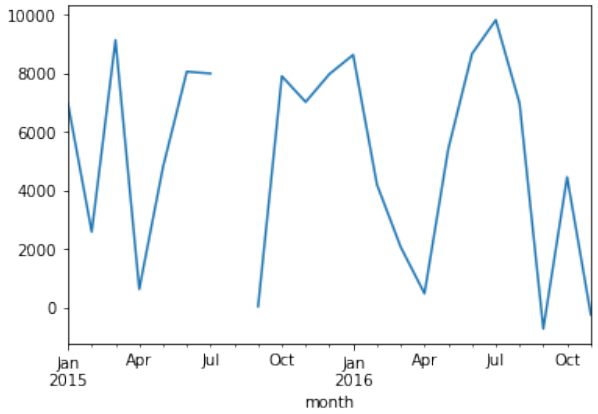
11.1.5 14.1 Grafy a Pandas

```
>>> import matplotlib
>>> matplotlib inline
#       Zapnout zobrazování grafů (procento uvozuje "magickou" zkratku IPythonu)
```

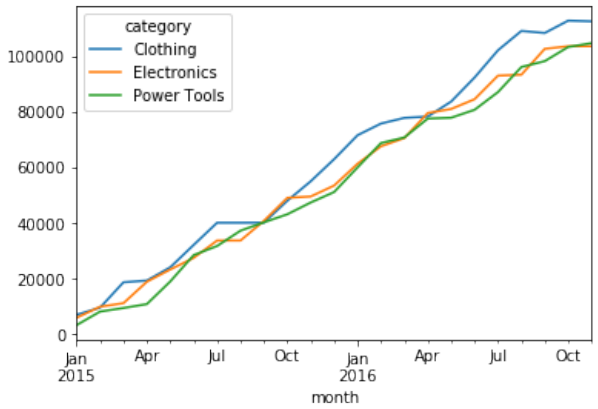
```
>>> unstacked.loc[Clothing].dropna().plot()

<matplotlib.axes.\_subplots.AxesSubplot at 0x7faa93f39c18>
```

```
        Setup
>>> import matplotlib.pyplot

        Plot
>>> unstacked.loc[Clothing].plot()
>>> matplotlib.pyplot.show()
>>> matplotlib.pyplot.savefig(graph.png)  ulozi graf do souboru
```

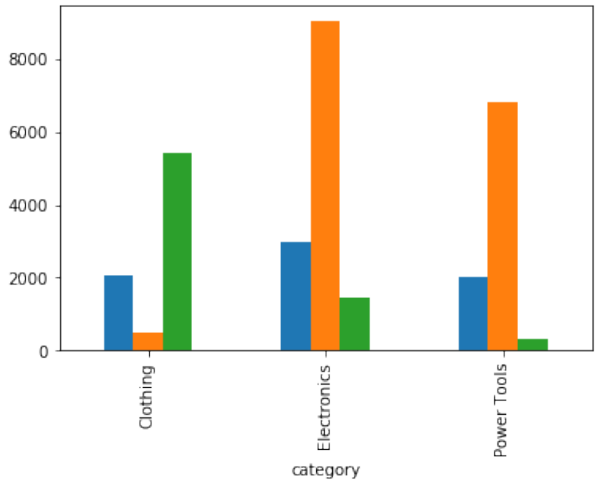





```
        Jak se postupně vyvíjely zisky z oblečení?
        '.T' udělá transpozici tabulky (vymění řádky a sloupce)
        'cumsum()' spočítá průběžný součet po sloupcích
>>> unstacked.T.fillna(0).cumsum().plot()

<matplotlib.axes._subplots.AxesSubplot at 0x7faa91c40a90>
```




```
>>>     unstacked.loc[:, 201603:201605].plot.bar(legend=False)

<matplotlib.axes._subplots.AxesSubplot at 0x7faa91b858d0>
```



11.1.6 14.5 Groupby

slouci dohromady radky se stejnou hodnotou v nekterem sloupci a sloucena data nejak agreguje

```
>>> data.head()

        month     category  sales
        0 2015-01-31  Electronics   5890
        1 2015-01-31  Power Tools   3242
        2 2015-01-31     Clothing   6961
        3 2015-02-28  Electronics   3969
        4 2015-02-28  Power Tools   4866
```

```
>>> data.groupby(category)  vysledkem groupby je jen objekt

<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7faa91b85e10>
```

```
>>> data.groupby(category).sum()  nutno na nem zavolat agregujici funkci

        sales
        category
        Clothing     112701
        Electronics  103742
        Power Tools  104859
```

```
>>> data.groupby(category).count()

        month  sales
        category
        Clothing        22     22
        Electronics     22     22
        Power Tools     23     23
```

```
>>> data.groupby([category, month]).sum()  lze agregovat i pres vice sloupcu

        sales
        category    month
        Clothing    2015-01-31   6961
```

```
        2015-02-28   2578
        2015-03-31   9131
        2015-04-30    618
        2015-05-31   4796
        {\ldots}                        {\ldots}
        Power Tools 2016-07-31   6382
        2016-08-31   9039
        2016-09-30   2119
        2016-10-31   5095
        2016-11-30   1397

        [67 rows x 1 columns]
```

```
>>> agg predava jakoukoli agregujici funkci, nejen obvykle jako sum nebo mean
>>> data.groupby(category).agg([mean, median, sum, pandas.Series.kurtosis])

        sales
        mean  median     sum      kurt
        category
        Clothing     5122.772727  6185.5  112701 -1.298035
        Electronics  4715.545455  4294.5  103742 -1.353210
        Power Tools  4559.086957  3769.0  104859 -1.044767
```

```
>>> g = data.groupby(month)  objekt lze nazvat a ulozit
>>> g.describe()

        sales
\textbackslash{}
        count          mean           std     min     25\%     50\%
75\%
        month
        2015-01-31   3.0  5364.333333  1914.414880  3242.0  4566.0  5890.0
6425.5
        2015-02-28   3.0  3804.333333  1152.853995  2578.0  3273.5  3969.0
4417.5
        2015-03-31   3.0  3900.333333  4529.891978  1281.0  1285.0  1289.0
5210.0
        2015-04-30   3.0  3250.000000  3895.490855   618.0  1012.5  1407.0
4566.0
        2015-05-31   3.0  5792.000000  2069.341200  4409.0  4602.5  4796.0
6483.5
        2015-06-30   3.0  7241.333333  2747.220656  4180.0  6116.0  8052.0
8772.0
        2015-07-31   3.0  5836.333333  2388.415653  3267.0  4760.0  6253.0
7121.0
        2015-08-31   1.0  5534.000000          NaN  5534.0  5534.0  5534.0
5534.0
        2015-09-30   3.0  3371.000000  3542.417960    31.0  1513.5  2996.0
5041.0
        2015-10-31   3.0  6367.666667  3002.029702  2909.0  5402.5  7896.0
8097.0
        2015-11-30   3.0  3917.666667  3273.148688   494.0  2368.5  4243.0
```

```
5629.5
        2015-12-31   3.0  5225.333333  2377.587082  3769.0  3853.5  3938.0
5953.5
        2016-01-31   3.0  8453.666667   536.431108  7852.0  8239.5  8627.0
8754.5
        2016-02-29   3.0  6417.666667  2290.170372  4194.0  5242.0  6290.0
7529.5
        2016-03-31   3.0  2345.666667   537.738164  2012.0  2035.5  2059.0
2512.5
        2016-04-30   3.0  5439.000000  4444.797408   471.0  3639.0  6807.0
7923.0
        2016-05-31   3.0  2391.333333  2675.235566   314.0   882.0  1450.0
3430.0
        2016-06-30   3.0  5012.000000  3178.877632  2858.0  3186.5  3515.0
6089.0
        2016-07-31   3.0  8232.000000  1732.765131  6382.0  7439.5  8497.0
9157.0
        2016-08-31   3.0  5452.333333  4539.188621   349.0  3659.0  6969.0
8004.0
        2016-09-30   3.0  3569.333333  5183.962802  -735.0   692.0  2119.0
5721.5
        2016-10-31   3.0  3487.333333  2247.644174   919.0  2683.5  4448.0
4771.5
        2016-11-30   3.0   385.333333   887.008643  -259.0  -120.5
18.0   707.5


        max
        month
        2015-01-31  6961.0
        2015-02-28  4866.0
        2015-03-31  9131.0
        2015-04-30  7725.0
        2015-05-31  8171.0
        2015-06-30  9492.0
        2015-07-31  7989.0
        2015-08-31  5534.0
        2015-09-30  7086.0
        2015-10-31  8298.0
        2015-11-30  7016.0
        2015-12-31  7969.0
        2016-01-31  8882.0
        2016-02-29  8769.0
        2016-03-31  2966.0
        2016-04-30  9039.0
        2016-05-31  5410.0
        2016-06-30  8663.0
        2016-07-31  9817.0
        2016-08-31  9039.0
        2016-09-30  9324.0
        2016-10-31  5095.0
```

```
        2016-11-30  1397.0
```

agregovat lze i podle sloupce, ktere nejsou v tabulce

```
        Následující kód rozloží data na slabé, průměrné a silné měsíce podle toh
        kolik jsme v daném měsíci vydělali celých tisícikorun, a zjistí celkový
        ze slabých, průměrných a silných měsíců:

>>> bin_size} = 10000
>>> by_month} = data.groupby(month).sum()
>>> by_thousands} = by_month}.groupby(by_month}[sales] // bin_size} * bin_size})
>>> by_thousands}
sales
count     sum
sales
0          5   30651
10000     15  218870
20000      3   71781
```

```
>>> by_thousands}[(sales, sum)].plot()
<matplotlib.axes.\_subplots.AxesSubplot at 0x7faa91a22748>
```