

Python pro matematickou (pato)fyzologii

Michal Šitina

14. února 2022

Ústav patologické fyziologie
Lékařská fakulta
Masarykova univerzita Brno

Obsah

1 Úvod	4
2 Datové typy	6
2.1 Čísla	6
2.2 Sekvence	7
2.3 Množiny	9
2.4 Dictionary	9
2.5 Ostatní datové typy	10
2.6 Změna datového typu	10
2.7 Porovnání	11
3 Programové konstrukty	12
3.1 Podmínka	12
3.1.1 Podmíněný výraz	12
3.2 Cykly	13
3.2.1 break a continue	14
3.3 Comprehensions	14
4 Funkce	16
4.1 Předání parametru funkci	17
4.1.1 Defaultní parametry	17
4.1.2 Libovolný počet parametrů	18
4.2 Jmenné prostory	18
4.3 Rekurze	20
4.4 Lambda funkce	20
5 Práce se soubory	22
5.1 Funkce file objektu	22
5.2 Zápis do souboru	22
5.3 Načtení ze souboru	23
5.4 with..as	24
5.5 Nastavení polohy kurzoru	25
5.6 Odchytávání chyb	25
6 String Objekty	27
6.1 Standardní string metody	27
6.2 Aplikace string metod na soubor	28
6.3 Automatické doplňování a formátování textu	28
6.3.1 Konkatenace	29
6.3.2 % notace	29
6.3.3 format() notace	29
6.3.4 f - notace	29
6.4 Datový formát CSV	30

7	Systémové programování	32
7.1	sys	32
7.2	os	33
7.2.1	Propojení na operační systém	34
7.3	time	36
7.3.1	Měření trvání běhu programu	36
8	Importování modulu	37
8.1	Průběh importování modulu	37
8.2	Vytvoření vlastního modulu	38
8.3	Instalace externích paketů	38

1 Úvod

Python je v současnosti jedním z nejpoužívanějších 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 soubor 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 v podstatě 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 programovacích 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

¹Python 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 nainstalovat 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.

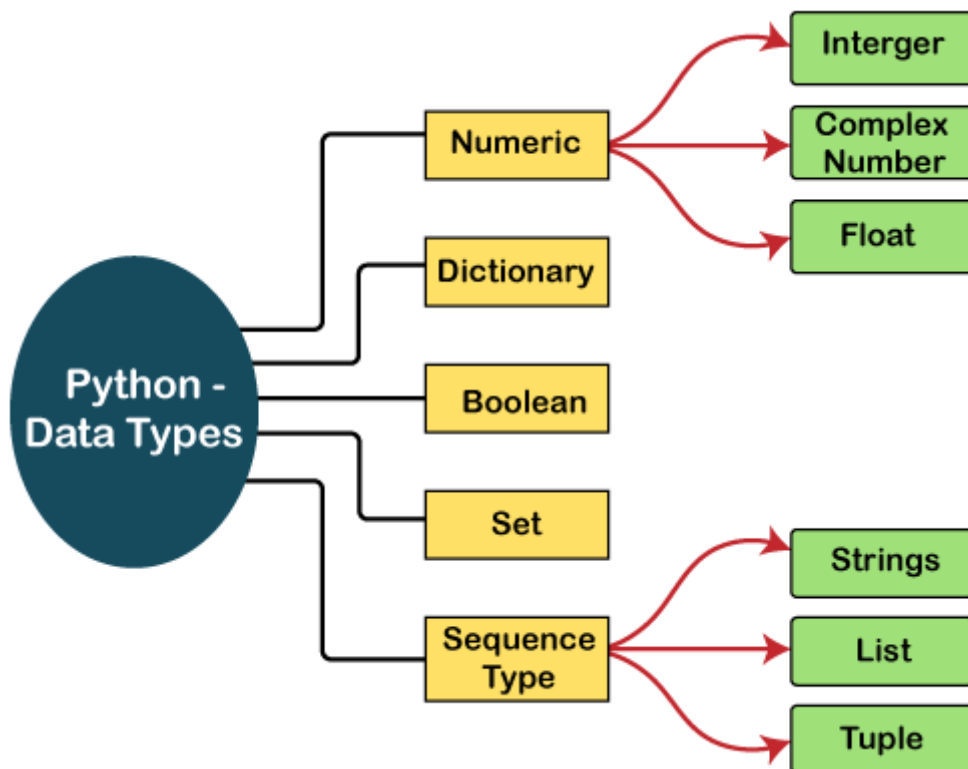
35 keywords in python 3

False	continue	global	pass
None	def	if	raise
True	del	import	return
and	elif	in	try
as	else	is	while
assert	except	lambda	with
async	finally	nonlocal	yield
await	for	not	class
break	from	or	

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 hlásí 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

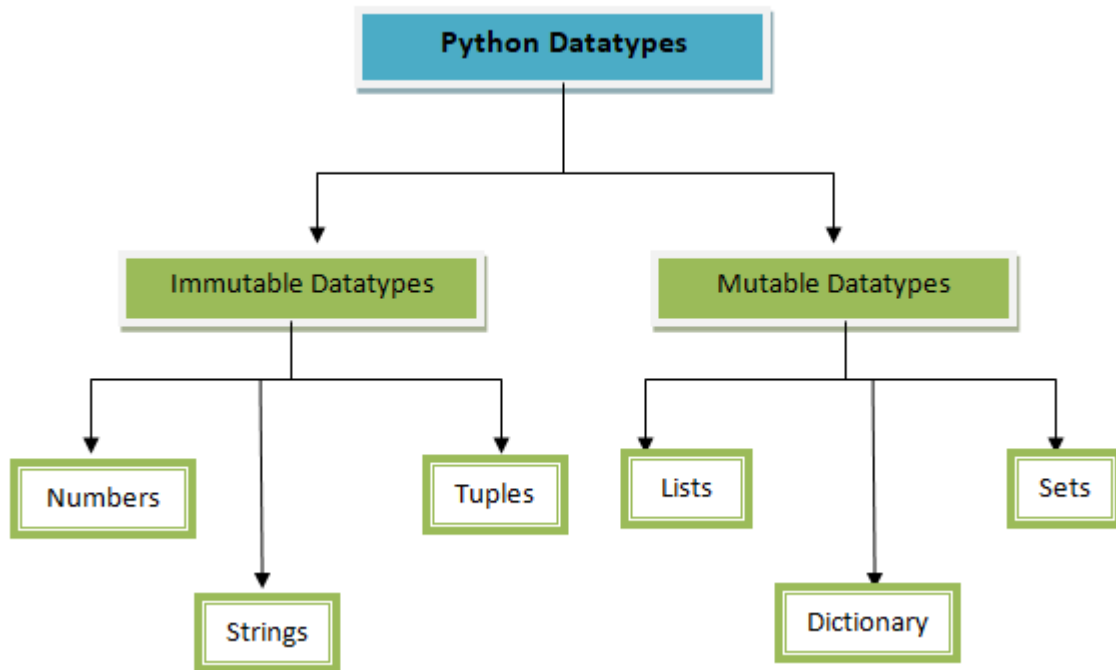
2.1 Čísla

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 řetězec, *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
```

Základní operace se sekvencemi

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

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

```
>>> l = [[1,2,3], 4.5, dfdfd]
>>> l[-1] # poslední 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

<code>s1.union(s2)</code>	sjednocení
<code>s1.intersection(s2)</code>	průnik
<code>s1.difference(s2)</code>	rozdíl
<code>s1.symmetric_difference(s2)</code>	symetrický rozdíl
<code>s1.issubset(s2)</code>	Je podmnožina?
<code>s1.isuperset(s2)</code>	Je nadmnožina?
<code>x in s</code>	Obsahuje množina s prvek x?
<code>x not in s</code>	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

<code>d[k]</code>	value s key == k
<code>d[k] = x</code>	do key == k vloží hodnotu x
<code>d.clear()</code>	vše odstraní, zůstane prázdné dictionary
<code>del d[k]</code>	smaže value s key = k
<code>d.get(k, x)</code>	vrátí d[k], pokud k je v d, jinak vrátí x
<code>k in d</code>	Je key k v d?
<code>k not in</code>	Není key k v d?
<code>d.keys()</code>	vrátí list s keys
<code>d.values()</code>	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' # konkatence = 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 součástí podmínky je logický výraz, který je vyhodnocen jako pravdivý (True) nebo nepravdivý (False). Logický výraz jsou vyhodnocováni vždy zleva doprava, až je situace jasná, dále už ne. Porovnání má přednost před 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 sekvencí, 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 pocet 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 i%6 == 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 rychleji 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: ')
>>> pocetSamohlasek(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 = viceVystupu()
>>> 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 používá 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, tuple)

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 číslo

>>> 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 zadávají 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 existují 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, že proměnná není lokální.

```
>>> 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(n-1)) # 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 lambda funkce používá uvnitř následujících funkcí:

<code>map(funkce, sekvence)</code>	map na každý prvek sekvence aplikuje funkci
<code>filter(funkce, sekvence)</code>	filter vybere ze sekvence hodnoty definované ve funkci
<code>list.sort(klic)</code>	sort seřadí sekvenci podle klíče zadaného funkcí
<code>sorted(list, klic)</code>	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)	zapiše 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 Zápís 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 už je zavřený
```

```
-----
ValueError Traceback (most recent call last)
```

```
<ipython-input-138-3ab1c9ca5a00> in <module>
```

```
--> 1 output.read () # nefunguje, soubor nebyl otevřen ke čtení
```

```
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 запиše na disk, ale nezavře
>>> output.closed # False .... není zavřený
>>> output.write('do File objektu lze dále přepisovat')
>>> output.close() # nyní se запиše 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 aktuální 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 uplatňuje 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
finally 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)
```

```
<python-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 jiným `id()`. String objekty mají řadu in-built funkcí, umožňujících modifikace 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_číslo

```
zarovnání <  zarovnání doleva
           >  zarovnání doprava
           ^  centrováný
minimální šířka  minimální počet rezervovaných míst
```

presnost	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 tabulkami lze csv soubor načíst pomocí funkce `pandas.read_csv()` z balíku *pandas* (viz kapitola ...).

```
>>> import csv
# načtení csv souboru
>>> reader = csv.reader(open(c:UsersPublicsoubor.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 s Pythonu.

K dispozici jsou (nejméně) 3 moduly:

- sys** zajišťuje přístup k a umožňuje 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. Spustíme 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` simulují příkazovou řádku operačního systému, umožňuje zhruba totéž. Tedy např. vytvářet a mazat adresáře, spouštět programy, měnit parametry operačního systému apod.

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

Operace se soubory a adresáři

<code>chdir(path)</code>	změna adresáře (change directory)
<code>getcwd()</code>	aktuální adresář (get current working directory)
<code>listdir(path)</code>	výpis obsahu aktuálního adresáře (list of directory)
<code>path.isdir(path)</code>	vrací <code>True</code> , pokud <code>path</code> je adresář
<code>path.isfile(path)</code>	vrací <code>True</code> , pokud <code>path</code> je soubor
<code>access(path, mode)</code>	prověří přístupová práva a existenci souboru
<code>chmod(path, mode)</code>	mění přístupová práva k souboru či adresáři (change mode)
<code>mkdir(path)</code>	vytvoří adresář daný v <code>path</code>
<code>makedirs(path)</code>	vytvoří adresář daný v <code>path</code> se všemi podadresáři
<code>remove(path)</code>	odstraní soubor daný v <code>path</code>
<code>rmdir(path)</code>	odstraní prázdný adresář daný v <code>path</code>
<code>removedirs(path)</code>	odstraní všechny prázdné adresáře dané v <code>path</code>
<code>rename(old, new)</code>	přejmenuje adresář či soubor <code>old</code> na <code>new</code>
<code>renames(old, new)</code>	přejmenuje adresář či soubor <code>old</code> na <code>new</code> i všechny meziadresáře
<code>path.exists(path)</code>	<code>true</code> , pokud <code>path</code> 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 Propojení 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 aktuální čas v 24-znakovém formátu
ctime(secs)	mění sekundy na 24 znaků nebo udá aktuální čas v 24-znakovém formátu
gmtime(secs)	mění sekundy na tuple, nebo udá aktuální čase jako tuple
localtime(secs)	udá aktuální 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, které umožňují snadno provést řadu jinak složitých činností. V jiných programovacích jazycích se používají i další synonyma - např. balíčky, 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 adresářů, 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\\loket-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 nějaké funkce, s příponou `.py`. Vytvoříme soubor `mujmodul.py`, v programu definujeme funkci mojí funkce a soubor uložíme 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, že 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 conda instaluje příkazem `conda 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)