# KMI/PHT - Programování v jazyce Python

January 14, 2019

## 1 Vlastnosti

- interpretovaný jazyk s dynamickou kontrolu datových typů
- podporuje různá programovací paradigmata (objektově orientovaný, imperované, imperativní, procedurální, funkcionální)
- dvě nekompatibilní verze 2.X a 3.X
- jednoduchá syntaxe
- pro oddělování bloků se používá tabulátor (nepoužívají se závorky)
- dobře spolupracuje s ostatními programovací jazyky - dá se využít jako skriptovací jazyk.
- má více implementací (PyPy, IronPython, Jython, Boost.Python)
- python cachuje malé hodnoty (čísla, stringy (asi), tuply)

## 2 Datové typy

### 2.1 Základní rozdělení

- None (třída NoneType):
- Čísla:
  - Reálné (třída float)
  - Komplexni (třída complex)
  - Integral
    - Integer (třída int)
    - Boolean (třída bool)
- Sekvence:
  - Imutabilní
    - Řetězce (třída str)
    - N-tice (třída tuple)
    - Byty (třída bytes)
  - Mutabilní
    - List (třída list)
    - Pole bytů (třída bytearray)
- Množiny:
  - Set (třída set)

  - Frozen set (třída frozenset)

- Slovník:
  - Dictionary (třída dict)

### 2.1.1 Čísla

**Desetinná čísla, zlomky**

- 1.1 a 2.2 nemají přesnou reprezentaci v desetinné čárce
- Nutnost použít decimal => modul pro podporu správně zaokrouhledné aritmetiky plovoucí řád. čárky
- Díky tomu stejná aritmetika jako se učíme ve škole.
- Nuly na konci jsou pak ve výsledku 1.30 * 1.20 = 1.5600
- fractions:
  - Podporuje aritmetiku s racionálními čísly
  - Můžeme ho sestrojit z páru čísel, jiné „fraction“, čísel a stringu
- Python nemá omezenou velikost čísel

```
In [ ]: 3.1415
        3+4j

        3/4
        1/3 + 1/2

        # výpočet mocniny
        2**100, math.pow(2, 100)
```

```
In [ ]: # více na: https://docs.python.org/2/library/decimal.html
        from decimal import Decimal

        # nastavení počtu míst za řádovou čárkou
        getcontext().prec = 3
        Decimal(1) / Decimal(7)
```

```
In [ ]: # více na: https://docs.python.org/2/library/fractions.html
        from fractions import Fraction
        from decimal import Decimal

        Fraction(16, -10)
        Fraction(1.1)
        Fraction(Decimal('1.1'))
```

**Podpora různých číselných soustav:**

```
In [ ]: # binární zápis
        0b010101
        # octální zápis
```

```
0o1777
# hecadecimální zápis
0x9ff

# můžeme používat binární operátory >>, <<, &, |
x = 99
print(bin(x))
(256).bit_length()
int('1101', 2)      # výsledkem je: 13
```

### Operace

```
In [ ]: # Matematické operace
+    sčítání
-    odčítání
*    násobení
/    dělení
%    modulo
**   exponent
//   floor division

# Porovnávání
==   rovno
!=   není rovno
<>   není rovno
>    větší než
<    menší než
>=   větší rovno
<=   menší rovno

# Přiřazení
=    operátor přiřazení
+=   příčíst k hodnotě a přiřadit
-=   odečíst od hodnoty a přiřadit
**=  vypočítat mocninu a přiřadit

# Bitové operace
&    logický and (and)
|    logický or (or)
^    logický xor
~    negace (not)
<<   bitový posun vlevo
>>   bitový posun vpravo

# Další
and, or, not, in, not in
is, is not (když id(x) se rovná id(y)
```

```
        # Pokročilejší matematické operace
        ## více na: https://docs.python.org/2/library/math.html
        import math

        math.pi
        math.sqrt(25)
```

**Různá úskalí a zajímavosti**

```
In [ ]: x = 2
        y = 4
        z = 6

        print(x < y < z)       # výsledkem je: True
        print(x == y < z)      # výsledkem je: False => priorita znamínek

        # Zásadní problém s desetinými čísly
        print(1.1 + 2.2)       # výsledkem je: 3.3000000000000003
        print(x / y)           # výsledkem je: 0.5
        print(x // y)          # výsledkem je: 0 => krácení

        import math
        math.floor(2.5)        # výsledkem je: 2
        math.floor(-2.5)       # výsledkem je: -3
        math.trunc(2.5)        # výsledkem je: 2
        math.trunc(-2.5)       # výsledkem je: -2

        # Řešení zásadního problému s desetinými čísly
        from decimal import Decimal
        Decimal('1.1') + Decimal('2.2')  # výsledkem je: 3.3

        # Prohození hodnot
        a = 10
        b = 20
        print('a:', a, 'b:', b)
        a, b = b, a
        print('a:', a, 'b:', b)

        # operátory += atp. jsou obecně rychlejší nez klasický způsob
        a = 1
        a += 1
```

### 2.1.2 Řetězce

- Stringové literály jsou v Pythonu obklopeny jednoduchými nebo dvojitými uvozovkami
- je to jedno které uvozovky se použijí, ale zvykem je používat jednoduché
- Zbytek v strings.py
- Je imutabilní prvek

In [ ]:
```
# řezy

s = 'spam'
s[0]         # výsledkem je: 's'
s[len(s)-1] # výsledkem je: 'm'
s[-1]        # výsledkem je: 'm'
s[1:3]       # výsledkem je: 'pa'
s[1:]        # výsledkem je: 'pam'
s[:3]        # výsledkem je: 'spa'
s[:]         # vrací celý řetězec => ne původní, ale kopii řetězce

s ='abcdefghi'
s[0:9:3]     # výsledkem je: 'adg'
s[::-1]      # výsledkem je: 'ihgfedcba' => převrátí řetězec
s[::2]       # výsledkem je: 'acegi'
s[5:1:-1]    # výsledkem je: 'fedc'

m = 1
n = 1
retezec = 'SPAM'

a, b, c = retezec[0], retezec[1], retezec[2]  # uloží do a='S', b='P', c='A'
a, b, c, d = list(retezec[:2] + retezec[2:])  # uloží do a='S', b='P', c='A', d='M'
a, b = retezec[2]                             # chyba => málo hodnot pro uložení

seq = [1, 2, 3, 4]
a, b, c, d = seq        # uloží do a='1', b='2', c='3', d='4'
a, b = seq              # chyba => 2 proměnné a ukládáme 4 hodnoty

# Řešení pro případ, kdy máme méně hodnot než proměnných
# podmínka: * může být na levé straně jen jednou
a, *b = seq              # výsledkem je: a=1, b=[2, 3, 4]
*a, b = seq              # výsledkem je: a=[1,2,3], b=4
a, *b = 'spam'           # výsledkem je:  a='s', b=['p', 'a', 'm']
```

In [ ]:
```
# zřetězení

s = 'Spam'
s + 'Spam'        # výsledkem je: 'SpamSpam'
s*4               # výsledkem je: 'SpamSpamSpamSpam'
s[0] = 'z'        # spadne, protože seznam je imutabilní prvek
s = 'z' + s[1:]   # výsledkem je: 'zpam'
```

In [ ]:
```
# funkce pro práci s řetězci

s = 'spam'
s1 = 'spam and spam'
s2 = 'SPAM'
```

```
        len(s)            # výsledkem je: 4
        s.find('pa')      # vrací index, na kterém byl nalezen podřetězec
        s.replace(s, 'z') # výsledkem je: 'z'
        s1.split(' ')     # výsledkem je: ['spam', 'and', 'spam']
        s.upper()         # výsledkem je: 'SPAM'
        s2.lower()        # výsledkem je: 'spam'
        s.strip()         # odřezává bílé znaky
```

```
In [ ]: # formátování řetězců

        'k' in 'kulicka'                  # výsledkem je: True
        '%s p a %s' %('s','m')            # výsledkem je: 's p a m'
        '{} p a {}'.format('s', 'm')      # výsledkem je: 's p a m'

        # f-strings
        a = 'bob'
        b = 'spam'
        print(f'{a} eat {b}')             # výsledkem je: bob eat spam
        print(f'{a.upper()} eat {b}')     # výsledkem je: BOB eat spam

        # funkce print podrobněji
        print(a, b, c, sep=', ', end='\n', file=soubor)
```

### 2.1.3 ID

- vše v Pythonu je objekt (dokonce čísla i třídy)
- unikátní id má tedy i 5, toto id zůstává neměnné po dobu celého života
- Pokaždé, když se definuje nový objekt, vytvoří se nový objekt s novou identitou
- Výjimkou jsou pak malá čísla a malé řetězce (jeden objekt s několika pointery)
- u čísel a řetězců je to kvůli integer cachování a internování řetězce, funguje i float a komplexních čísel

```
In [ ]: x = 42
        y = 42

        y == y  # výsledkem je: True
        x is y  # výsledkem je: True
```

### 2.1.4 Regulární výrazy

```
In [ ]: # více na: https://docs.python.org/3/library/re.html
        import re

        s = 'The rain in Spain'
        result = re.search('^The.*Spain$', s)
```

### 2.1.5 Konverze datových typů

```
In [ ]: int('42')    # výsledkem je: 42
        str(42)      # výsledkem je: '42'
        ord('a')     # výsledkem je: 97
        chr(97)      # výsledkem je: 'a'
```

### 2.1.6 Kolekce

**Seznam (list)**

- seřazený, měnitelný (mutabilní), umožňuje duplikace
- seznamy je možné řezat, spojovat, připovat

```
In [ ]: L = [123, 'spam', 1.23]
        [1, ['tpi', 2], 4.5]       # výsledkem je: [1, ['tpi', 2], 4.5]
        len(L)                     # výsledkem je: 3 => počet pruků v seznamu
        L.count(123)               # výsledkem je: 1 => počet nalezených pruků v seznamu
        L[0]                       # výsledkem je: 123
        L + ['asdf', 456]          # výsledkem je: [123, 'spam', 1.23, 'asdf', 456]
        L * 2                      # výsledkem je: [123, 'spam', 1.23, 123, 'spam', 1.23]
        L.append(1)                # vloží nový prvek nakonec seznamu L
        L.insert(1, 'toast')       # vloží nový prvek (řetězec 'toast') na pozici 1
        L.extend([4.2])            # rozšíří seznam o nový seznam
        L.pop(2)                   # odstraní prvek ze seznamu na indexu 2 a vrátí ho
        del L[2]                   # odstraní prvek ze seznamu na indexu 2
        L[0] = []                  # seznam je opravdu mutabilní prvek

        # zipování
        zipped = zip([1, 2, 3], ['a', 'b', 'c'])
        c, v = zip(*list(zipped))
        print(f'original lists: {c} and {v}')

        # matice
        M = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
        M[1]                       # výsledkem je: [4,5,6]
        M[1][2]                    # výsledkem je: 6

        # uložení seznamu v paměti
        L = [1, 2, 3]
        M = L
        L == M          # výsledkem je: True
        L is M          # výsledkem je: True

        L = [1, 2, 3]
        M = [1, 2, 3]
        L == M          # výsledkem je: True
        L is M          # výsledkem je: False => seznamy nejsou v paměti na stejném místě
```

```
L=[4, 5, 6]
x = L*4          # vyhodnotí se na [4, 5, 6][4, 5, 6][4, 5, 6][4, 5, 6]
y = [L] * 4      # vyhodnotí se na [[4, 5, 6][4, 5, 6][4, 5, 6][4, 5, 6]]
L[0] = 1         # x zůstane nezměněné, ale y se změní
```

**N-tice (tuple)**

- seřazené, neměnitelné (imiutabilní), umožňuje duplikace

```
In [ ]: (1, 2, 3)          # výsledkem je: (1, 2, 3)
        (2, 3)             # výsledkem je: (2, 3)
        tuple('spam')      # výsledkem je: ('s', 'p', 'a', 'm')
        (1, 2) + (3, 4)    # výsledkem je: (1, 2, 3, 4)

        T = (1, 2, 3, 'a', 'A')
        T[3]               # výsledkem je: 3
        T.index(3)         # výsledkem je: 2 => vrací index, kde byl prvek nalezen
        T.count('a')       # výsledkem je: 1 => počet prvků 'a' nalezených v n-tici
```

**Množiny (set, frozenset)**

- neseřazené, neindexované, neumožňuje duplikace
- množiny nesmí obsahovat imutabilní prvky

```
In [ ]: # SETS - mutabilní
        # více na: https://docs.python.org/2/library/sets.html

        {'a', 'b', 'c'}     # výsledkem je: {'a', 'b', 'c'}
        set([1, 2, 3])      # výsledkem je: {1, 2, 3}

        # Množinové operace
        X = {'a', 'b'}
        Y = {'c', 'd'}

        X & Y    X.intersection(Y)          # průnik
        X | Y    X.union(Y)                 # sjednocení
        X - Y    X.difference(Y)            # rozdíl
        X >= Y   X.issuperset(Y)            # nadmnožina
        X <= Y   X.issubset(Y)              # podmnožina
        # jako rozdíl ale do množiny vloží prvky buď z X nebo Y, ale ne z obou
        X ^ Y    X.symmetric_difference(Y)

        'e' in X                            # výsledkem je: False
        X.add('e')                          # přidá do množiny prvek
        X.remove('a')                       # odstraní prvek z množiny
        X.update({'f', 'g'})                # rozšíří množinu o novou množinu

        S = {1, 2, 3}
```

```
S|[2, 4]                              # nebude to fungovat
S.union([2, 4])                       # bude to fungovat


# FROZEN SETS - imutabilní

{frozenset({2, 3})}                   # bude to fungovat
{{2, 3}}                              # nebude to fungovat
```

**Slovník (dict)**

- neseřazené, měnitelná, indexovaná, neumožňuje duplikace
- formát slovníku => { klic1: hodnota1, klic2: hodnota2, .., klicn: hodnotan }

```
In [ ]: dict(hour = 8)        # výsledkem je: {'hour': 8}
        D = {}                # výsledkem je: {}
        D['name'] = 'Bob'     # přidá do slovníku prvek s klíčem 'name' a hodnotou Bob
        dict(zip([1, 2, 3], ['a', 'b', 'c']))   # výsledkem je: {1: 'a', 2: 'b', 3: 'c'}

        D['name'] in D                          # výsledkem je: False
        'name' in D                             # výsledkem je: True
        D.items()                               # vrací seznam hodnot ze slovníku
        D.keys()                                # vrací seznam klíčů ze slovníku

        choice = 'ham'
        print({'asdf': 1, 'ham': 2}[choice])    # výsledkem je: 2
```

# 3 Comprehensions

- velmi užitečný konstrukt jazyka
- údajně efektivnější než cykly
- mají lokální rozsah platnosti

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

        second_column = [row[1] for row in M]
        # výsledkem je: [2, 5, 8] => druhý sloupec matice

        [row[1] for row in M if row[1] % 2 == 0]
        # výsledkem je: [2, 8] => sudé hodnoty druhého sloupce

        [M[i][i] for i in [0, 1, 2]]
        # výsledkem je: [1, 5, 9] => prvky na diagonále

        list(map(sum, M))
        # výsledkem je: [6, 15, 24] => součty v jednotlivých řádcích

        G = (sum(row) for row in M)   # vytváří nějaký generátor
```

```
next(G)                          # výsledkem je: 6
next(G)                          # výsledkem je: 15
next(G)                          # výsledkem je: 24
next(G)                          # spadne to, protože tam není další prvek
```

# 4 Větvení

- v Pythonu není switch

```
In [ ]: # Podmínka: if - else
        x = 1
        if x <= 2:
            print(x, '<=', 2)
        else:
             print(x, '>', 2)                # výsledkem je: 1 <= 2
```

```
In [ ]: # Podmínka: if - elseif - else
        x = 1
        if x < 2:
            print(x, '<', 2)
        elif x > 2:
            print(x, '>', 2)
        else:
             print(x, '=', 2)               # výsledkem je: 1 < 2
```

# 5 Smyčky

- pokud možno nejdřív vyzkoušet jestli nelze použít Comprehensions
- klasická klíčová slova: break, continue, pass (nedělá nic)
- pokud má tělo cyklu jen jeden výraz nemusí se zalamovat na nový řádek
- specialita: else-blok

## 5.1 cyklus while

```
In [ ]: L = [1, 2, 3, 4, 5]

        # Klasický while cyklus
        while L:
            front, L = L[0], L[1:]
            # nebo unpacking

        while True:
            print('Ctrl-C')


        # Cyklus while s else-blokem
        while pominka:
```

```
        # tělo cyklu
    else:
        # vykoná se pokud smyčka není ukončena breakem

    # příklad použití: cyklus while vs. cyklus while s else-blokem
    found = False

    while L and not found:
        if L[0] == 6:
            found = True
        else:
            found = False
    print(found)

    while L:
        if L[0] == 6:
            break
        L = L[1:]
    else:
        print('not found')
```

## 5.2 cyklus for

```
In [ ]: L = [1, 2, 3, 4, 5]

    # Cyklus foreach
    for item in L:
        print(item)

    for i in range(len(L)):
        L[i] += 1
        print(L[i])

    # náhrada za klasický for, tak jak ho známe z C while cyklem
    j = 0
    while i < len(L):
        L[i] += 1
        i += 1

    # správné řešení pomocí Comprehensions
    [x + 1 for x in L]

    # funkce enumerate převede objekt do iterovatelné podoby
    S = 'spam'
    index = 0
    for item in S:
        print('Item', item, 'on index ', index)
        offset += 1
```

```
        for (index, item) in enumerate(S):
            print('Item', item, 'on index ', index)
```

### 5.2.1 Příklady

```
In [ ]: items = [1, 2, 3, 4, 5, 6]
        test =  [2, 4, 7]
        S = 'abcdefghijklmnopqrstuvwxyz'

        # nalezení pruku v seznamu
        # pomocí for cyklu s else-blokem
        for i in test:
            for j in items:
                if i == j:
                    print(i, 'nalezeno')
                    break
            else:
                print(i, 'nenalezeno')

        # pomocí for cyklu
        for t in test:
            if t in items:
                print(t, 'nalezeno')
            else:
                print(t, 'nenalezeno')

        # pomocí comprehensions
        for item in test:
            print(item, 'nalezeno' if item in items else 'nenalezeno')

        # pomocí průniku množin = nejlepší řešení
        set(test) & set(items)


        # Další příklad
        for (i,c) in enumerate(S):
            if i % 2 == 0:
                print(c, end=' ')

        for i in range(0, len(S), 2):
            print(S[i], end=' ')

        for c in S[::2]:
            print(c, end=' ')

        [print(c, end=' ') for c in S[::2]]
```

```
[print(' '.join(S[::2]), end=' ')]



# Příklad s maticemi
size = 8
M = [0]*size
for i in range(0, size):
    M[i] = [0]*size

M = [[0]*size]*size

print(M, '\n')

#M[[0:size:-1]*size] = 1
print(M, '\n')
for (i,j) in zip(range(0,size), range(-1,-size-1, -1)):
    M[i][j] = 1

i = -1
for row in M:
    row[i] = 1
    i -= 1

for (i,j) in [(i,j) for i in range(2,6) for j in range(2,6)]:
    M[i][j] = 1
```

# 6 Funkce

- "def" je vykonavatelná instrukce
- LEGB - pravidlo říkající jak se hledají vazby proměnných
  - lokální
  - enclose (rozsah platnosti pro nadřazené entity)
  - globalní
  - built-in - zabudované v jazyku, získáme pomoci dir (builtin) např. dir(button)

```
In [ ]: # základní definice pomocí slova def
        def funkce (args): tělo s jedním výrazem
        def funkce (args):
            .
            .
            ... tělo funkce ...
            .
            .
        konec funkce je tam kde končí odsazení

        # malá anonymní funkce, bere i víc argumentů, ale může mít jen jeden výraz
```

```
lambda args: tělo
lambda args:
    .
    .
    ... tělo ...
    .
    .
konec výrazu

# návrát hodnoty z funkce pomocí return
return h1, h2, h3


if test:
    def f1():
        ...
else:
    def f2():
        ...

def f1():
    def f2():
        ...

# překrytí vazeb
a = 10
def f1():
    a = 100   # je v podstatě lokální proměnná
    print(a)
```

In [ ]:
```
# Přístup ke globální proměnné pomocí global
x = 9
def f1():
    global x
    x += 1      # změní se hodnota globální proměnné

# Přístup do nadřazeného pomocí nonlocal

# Funkce ve funkci
def f1():
    def f2():
        print('ahoj')
    return f2              # vrátí ahoj

def f1():
    def f2():
        print('ahoj')      # nedochází k volání (vrací se handler na funkci)
    return f2()
```

## 6.1 Unpacking pro funkce

```
In [ ]: def f(a, b, c): print (a, b, c)
        f(1, 2, 3)
        f(c=3, b=2, a=1)
        f(1, b=2, c=3)

        def f(a, b=1, c=3)

        # funkce s proměnným počtem argumentů
        def f(*arg): print(x)
        def f(**arg): print(x)  # bere argumenty ve tvaru slovníku např. {'a': 10, 'b': 15}

        # bere tři argumenty
        # ty po hvězdičku jsou podle pozice a za hvězdičkou můžou být podle jména
        def f(a, *, b, c)
```

## 6.2 Generátory

- např. range je generátor

```
In [ ]: def ctverce(n):
            for i in range(n):
                yield i**2

        a = ctverce(5)
        next(a)
        next(a)
        next(a)
        a.send(5)

        # např. range je generátor
        def gen():
            for i in range(10):
                x = yield i
                print(x)

        a = gen()
        next(a)
        a.send(100)

        # generátory v použití s comprehensions
        a = (x**2 for x in range(10))
```

# 7 Moduly a balíčky

## 7.1 Moduly

```
In [ ]: from math import sqrt
        import random as r
        import math # bude jen v daném prostředí a v nadřazeném ne

In [ ]: # soubor a.py
        def f(x):
            print(x)

        # soubor b.py
        import a
        a.f('ham')

        from a import f
        f('ham')

        from a import *
        dir(a) # ukáže co všechno je importované
```

## 7.2 Balíčky

- Jsou to v podstatě moduly v nějaké adresářové struktuře např. /dir1/dir2/modul
- Moduly je možné v Pythonu združovat do balíčků (packages)
- Balíček je adresář se soubory Pythonu (moduly)
- Balíčky a moduly v balíčcích se importují podobně jako moduly (příkazem import)

```
In [ ]: #  Aby byl adresář považován za balíček, musí být v adresáři soubor __init__.py
        import dir1.dir2.modul

        # soubor x.py
        a, _b = 1
        import x
        x._b     # v pořádku


        # druhá varianta
        from x import *
        _b       # chyba
```

# 8 Objektově orientované programování

```
In [ ]: # definice třídy
        class Auto (Vozidlo):     # class název (děděná třída):
            """ komentář, co daná třída představuje """
```

```
    # konstruktor
    def __init__(self, spotreba)
        self.spotreba = spotreba  #self - samotný objekt (to samé jako this v Javě)

    # metody, které je dobré mít definované (aby třída zapadla do ekosystému Pythonu)

    # abychom objekt mohli vytisknout jako řetězec (standartně se volá přes print)
    def __str__ (...):
        #... tělo ...

    # detailnější popis objektu (v nějakém interaktivnějším prostředí)
    def __repr__(...):
        #... tělo ...

    # destruktor
    def __del__(...):
         #... tělo ...

    # další metody
    # __add__, __or__, __lt__, __gt__, __lo__, __go__, __eq__, __ne__
```

# 9 Výjimky a práce se soubory

## 9.1 Práce se soubory

více na: https://docs.python.org/3/tutorial/inputoutput.html#reading-and-writing-files - 'r' - Soubor bude otevřen pro čtení (neexistuje-li tak IOError) - 'r+' - Soubor bude otevřen pro čtení a zápis (neexistuje-li tak IOError) - 'w' - Soubor bude otevřen pro zápis (neexistuje-li => vytvoří se, existuje-li => zkrácen na nulovou délku) - 'a' - Soubor bude otevřen pro zápis na konec souboru (append). (neexistuje-li => vytvoří se) - 'b' - Soubor bude otevřen v binárním režimu - 't' - Soubor bude otevřen v textovém režimu

```
In [ ]: M = [[r.randint(1,100) for j in range(cols)] for i in range(rows)]


    with open('input.txt', 'w') as f:                    # Otevření souboru input.txt
        for line in M:
                f.write(','.join(str(l) for l in line)) # Zápis do souboru
                f.write("\n")
```

## 9.2 Výjimky

```
In [ ]: # více na: https://docs.python.org/2/tutorial/errors.html
    dir(__builtins__)    # Všechhny výjimky definované Pythonem


    # Try-catch blok
    try:
```

```
        # blok, kde může dojít k výjimce
    except:
        # blok, který se provede, když dojde k výjimce
    else:
        # blok, který se provede, když nedojde k výjimce


    # Vyvolání výjimky
    raise NameError('Ahoj!')


    # Definice vlastních výjimek
    class MyError(Exception):

        def __init__(self, value):
            self.value = value

        def __str__(self):
            return repr(self.value)

    try:
        raise MyError(2*2)
    except MyError as e:
        print('My exception occurred, value:', e.value)
```

```
In [ ]: try:
            print(3 + 3)
            print('a' + 3)
            print(2 + 2)
        except:
            print('Doslo k chybe')
        else:
            print('Nedoslo k chybe')      # výsledkem je: 6 a 'Doslo k chybe'
```

## 10 Dekorátory

- funkce nebo třída, která rozšiřuje nebo mění funkčnost jiné třídy nebo funkce
- Používají se tam, kde nechceme funkčnost původní funkce/třídy měnit, ale chceme, aby se funkce/třída daného jména chovala trochu jinak
- Python má definované některé vlastní dekorátory
  - např. @classmethod dekorátor se používá pro vytváření třídních metod tříd (metody, které jako první argument nedostanou instanci třídy, ale třídu).

```
In [ ]: # Příklad
        def decorator_print():
            fce = print
            def xprint(*args, **kwargs):
```

```
            try:
                if(debug == True):
                    fce(*args, **kwargs)
            except NameError: # debug neni definovan
                pass
        return xprint

    print("Hello World")
    print = decorator_print()
    print("Hello World")
    debug = True
    print("Hello World")
```

In [ ]:
```
    # Příklad
    def log(func):
        def wrapper(*args, **kwargs):
            print(func.__name__)
            return func(*args, **kwargs)
        return wrapper


    class stack:
        def __init__(self):
            self.data = []

        @log
        def pop(self):
            value = self.data[-1]
            del self.data[-1]
            return value

        @log
        def top(self):
            return self.data[-1]

        @log
        def push(self, value):
            self.data.append(value)


    zasobnik = stack()
    zasobnik.push(10)
    zasobnik.push(11)
    print(zasobnik.top())
    zasobnik.pop()
    print(zasobnik.top())
```