

C2184 Úvod do programování v Pythonu (2021)

7. Chyby a testování

Chyby v kódu

- **Syntaktické chyby** (*syntax errors*) – program nelze vůbec spustit
- **Výjimky** (*exceptions*) – program běží, ale nastane chyba v průběhu vykonávání
- **Systematické chyby** – z pohledu Pythonu je všechno v pořádku, program běží, ale nedělá to, co chceme

Syntaktické chyby

- Špatné odsazení, chybějící nebo nadbytečné závorky, dvojtečky apod.
- Program nelze vůbec spustit

```
[1]: a = # Chybí pravá strana přiřazení
```

```
File "/tmp/ipykernel_29306/1037938702.py", line 1
a = # Chybí pravá strana přiřazení
    ^
SyntaxError: invalid syntax
```

```
[2]: if True:
    print('Hello World!') # Chybí odsazení
```

```
File "/tmp/ipykernel_29306/2517544079.py", line 2
print('Hello World!') # Chybí odsazení
^
IndentationError: expected an indented block
```

```
[3]: if True:
      print('Hello World!') # Odsazeno 4 mezerami
      print('') # Odsazeno tabulátorem
```

```
File "/tmp/ipykernel_29306/1622295480.py", line 3
print('') # Odsazeno tabulátorem
      ^
TabError: inconsistent use of tabs and spaces in indentation
```

- Často se chyba odhalí až na následujícím řádku

```
[4]: import math
a = abs(math.sqrt((10 - 5)**2)) # Chybí konec závorky
print(a)
```

```
File "/tmp/ipykernel_29306/2806914473.py", line 3
print(a)
^
SyntaxError: invalid syntax
```

Výjimky (*exceptions*)

- Za běhu programu (*runtime*) se zjistí, že něco nelze vykonat → vyhodí se výjimka
- Typ výjimky upřesňuje, proč to nelze:
 - Vestavěné typy výjimek: <https://docs.python.org/3/library/exceptions.html>
 - Většina knihoven obsahuje vlastní
 - Lze zadefinovat vlastní

Běžné typy výjimek

- `NameError` – snažíme se použít proměnnou, která neexistuje

```
[5]: print(b)
```

→

```
NameError                                Traceback (most recent call last)
```

```
/tmp/ipykernel_29306/324135452.py in <module>
----> 1 print(b)
```

```
NameError: name 'b' is not defined
```

- TypeError – snažíme se udělat něco s hodnotou špatného typu

```
[6]: 'abc' + 5
```

```
TypeError                                Traceback (most recent call last)
```

```
/tmp/ipykernel_29306/475208171.py in <module>
----> 1 'abc' + 5
```

```
TypeError: can only concatenate str (not "int") to str
```

- ValueError – dáváme funkci argument správného typu, ale špatnou hodnotu

```
[7]: int('12a')
```

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

```
/tmp/ipykernel_29306/762838879.py in <module>
----> 1 int('12a')
```

```
ValueError: invalid literal for int() with base 10: '12a'
```

- ZeroDivisionError – snažíme se dělit nulou

```
[8]: x = 5
     y = 1 / (x - 5)
```

```
↳ -----
ZeroDivisionError                                Traceback (most
↳ recent call last)

/tmp/ipykernel_29306/2405872419.py in <module>
      1 x = 5
----> 2 y = 1 / (x - 5)
```

ZeroDivisionError: division by zero

- IndexError – index je mimo rozsah

```
[9]: ciska = [1, 2, 8]
     ciska[3]
```

```
↳ -----
IndexError                                Traceback (most
↳ recent call last)

/tmp/ipykernel_29306/3525130419.py in <module>
      1 ciska = [1, 2, 8]
----> 2 ciska[3]
```

IndexError: list index out of range

- KeyError – klíč chybí ve slovníku

```
[10]: slovník = {'a': 1, 'c': 4}
     slovník['b']
```

```
↳ -----
KeyError                                Traceback (most
↳ recent call last)
```

```

/tmp/ipykernel_29306/1506280042.py in <module>
      1 slovník = {'a': 1, 'c': 4}
----> 2 slovník['b']

```

KeyError: 'b'

- NotImplementedError – funkce nebo její část ještě není naimplementovaná

```

[11]: from typing import Iterable

def median(numbers: Iterable[float]) -> float:
    sorted_numbers = sorted(numbers)
    n = len(sorted_numbers)
    if n % 2 == 1:
        return sorted_numbers[n//2]
    else:
        raise NotImplementedError() # TODO add implementation for
    even n

print(median([2, 8, 5]))
print(median([2, 8, 5, 7]))

```

5

NotImplementedError Traceback (most recent call last)

```

/tmp/ipykernel_29306/1089636490.py in <module>
     10
     11 print(median([2, 8, 5]))
--> 12 print(median([2, 8, 5, 7]))

```

```

/tmp/ipykernel_29306/1089636490.py in median(numbers)
      7         return sorted_numbers[n//2]
      8     else:
----> 9         raise NotImplementedError() # TODO add
implementation for even n
     10
     11 print(median([2, 8, 5]))

```

NotImplementedError:

- StopIteration - chceme další hodnotu od iterátoru, který se už vyčerpал

```
[12]: iterator = reversed('ab')
      next(iterator)
      next(iterator)
      next(iterator)
```

```
↳ -----
StopIteration                                Traceback (most recent call last)
  /tmp/ipykernel_29306/616370458.py in <module>
      2 next(iterator)
      3 next(iterator)
----> 4 next(iterator)
```

StopIteration:

- RecursionError - rekurzivní funkce se zacyklila

```
[13]: def factorial(n: int) -> int:
      return n * factorial(n-1)

factorial(5)
```

```
↳ -----
RecursionError                                Traceback (most recent call last)
  /tmp/ipykernel_29306/1772359378.py in <module>
      2 return n * factorial(n-1)
      3
----> 4 factorial(5)

/tmp/ipykernel_29306/1772359378.py in factorial(n)
      1 def factorial(n: int) -> int:
```

```

----> 2         return n * factorial(n-1)
      3
      4 factorial(5)

```

... last 1 frames repeated, from the frame below ...

```

/tmp/ipykernel_29306/1772359378.py in factorial(n)
      1 def factorial(n: int) -> int:
----> 2         return n * factorial(n-1)
      3
      4 factorial(5)

```

RecursionError: maximum recursion depth exceeded

Traceback

- Popisuje, kde nastala výjimka a jak se tam program dostal

```

[14]: from typing import List

def print_reciprocals(numbers: List[float]) -> None:
    for number in numbers:
        print_reciprocal(number)

def print_reciprocal(x: float) -> float:
    rec_x = reciprocal(x)
    print(f'Reciprocal of {x} = {rec_x}')

def reciprocal(x: float) -> float:
    return 1 / x

print_reciprocals([3, 2, 0, 4])

```

Reciprocal of 3 = 0.3333333333333333

Reciprocal of 2 = 0.5

```

↳ -----
ZeroDivisionError                                Traceback (most
↳ recent call last)

```

/tmp/ipykernel_29306/125771647.py in <module>

```

12     return 1 / x
13
---> 14 print_reciprocals([3, 2, 0, 4])

/tmp/ipykernel_29306/125771647.py in print_reciprocals(numbers)
3 def print_reciprocals(numbers: List[float]) -> None:
4     for number in numbers:
----> 5         print_reciprocal(number)
6
7 def print_reciprocal(x: float) -> float:

/tmp/ipykernel_29306/125771647.py in print_reciprocal(x)
6
7 def print_reciprocal(x: float) -> float:
----> 8     rec_x = reciprocal(x)
9     print(f'Reciprocal of {x} = {rec_x}')
10

/tmp/ipykernel_29306/125771647.py in reciprocal(x)
10
11 def reciprocal(x: float) -> float:
---> 12     return 1 / x
13
14 print_reciprocals([3, 2, 0, 4])

```

ZeroDivisionError: division by zero

Otázky:

Následující program vyhodí výjimku. Jaký bude traceback?

- A) <module>
foo(5)
spam(0)
- B) <module>
foo(5)
spam(0)
eggs(1)
- C) <module>
foo(5)
eggs(1)
- D) <module>


```
foo(5)
eggs(1)
spam(0)
```

```
[ ]: def foo(n):
    for i in range(n):
        if i % 2 == 0:
            spam(i)
        else:
            eggs(i)

def spam(i):
    return 1 / (i+1)

def eggs(i):
    result = []
    for j in range(i):
        spam(i)
    return result[i]

foo(5)
```

Ošetření výjimky

- Když se umíme s konkrétní výjimkou nějak vyrovnat
- Výjimku odchytíme pomocí bloku `try...except`

```
[15]: import math

def reciprocal(x: float) -> float:
    print(f'Calculating 1/{x}...')
    try:
        result = 1 / x
        print('Division completed without problems')
        return result
    except ZeroDivisionError:
        print("Sorry, can't divide by zero `_([])_/'")
        return math.nan # Not-a-number
```

```
[16]: reciprocal(5)
```

```
Calculating 1/5...
Division completed without problems
```

```
[16]: 0.2
```

```
[17]: reciprocal(0)
```

```
Calculating 1/0...  
Sorry, can't divide by zero "\_()\_/"
```

```
[17]: nan
```

Kombinace výjimek

```
[18]: try:  
    number = int(input())  
    result = 1 / number  
    print(f'OK: {result}')  
except ZeroDivisionError:  
    print('Nulou nelze dělit!')  
except ValueError:  
    print('Nemůžu převést zadaný vstup na číslo!')  
except (RuntimeError, TypeError, NameError):  
    pass # jako by se chyba nestala (nedoporučuje se)  
except:  
    print('Chyba vole!')  
    raise # výjimka se znovu vyvolá
```

Nemůžu převést zadaný vstup na číslo!

Použití odchycené výjimky

- Pomocí `except...as...`

```
[19]: try:  
    result = 5 / 0  
    print(result)  
except ZeroDivisionError as err:  
    print(f'Došlo k chybě ({err})!')
```

Došlo k chybě (division by zero)!

try...except...else...finally

```
[20]: def reciprocal(x: float) -> float:  
    try:  
        result = 1 / x  
    except ZeroDivisionError:  
        print('Nulou nelze dělit')  
        result = math.inf  
    else:  
        print('Proběhlo bez výjimky!')  
    finally:
```

```
print('Já se provedu pokaždé')  
return result
```

```
[21]: reciprocal(0)
```

Nulou nelze dělit
Já se provedu pokaždé

```
[21]: inf
```

```
[22]: reciprocal(5)
```

Proběhlo bez výjimky!
Já se provedu pokaždé

```
[22]: 0.2
```

```
[23]: def reciprocal(x: float) -> float:  
      try:  
          return 1 / x  
      except ZeroDivisionError:  
          print('Nulou nelze dělit')  
          return math.inf  
      else:  
          print('Proběhlo bez výjimky!')  
      finally:  
          print('Já se FAKT provedu pokaždé')
```

```
[24]: reciprocal(0)
```

Nulou nelze dělit
Já se FAKT provedu pokaždé

```
[24]: inf
```

```
[25]: reciprocal(5)
```

Já se FAKT provedu pokaždé

```
[25]: 0.2
```

Vyhození výjimky

- Pomocí raise
- Výjimku vytvoříme:
 - ExceptionType()

- ExceptionType('Error message')

```
[26]: def factorial(n: int) -> int:
      '''Return factorial of a non-negative integer n.'''
      if not isinstance(n, int):
          raise TypeError('n must be int')
      if n < 0:
          raise ValueError('Cannot calculate factorial of a negative_
↪number.')
      result = 1
      for i in range(1, n+1):
          result *= i
      return result
```

```
[27]: factorial(10)
```

```
[27]: 3628800
```

```
[28]: factorial(-5)
```

```
↪-----
ValueError                                Traceback (most_
↪recent call last)

/tmp/ipykernel_29306/3591406149.py in <module>
----> 1 factorial(-5)

/tmp/ipykernel_29306/3438635131.py in factorial(n)
      4         raise TypeError('n must be int')
      5     if n < 0:
----> 6         raise ValueError('Cannot calculate factorial of a_
↪negative number.')
      7     result = 1
      8     for i in range(1, n+1):
```

ValueError: Cannot calculate factorial of a negative number.

```
[29]: factorial(2.5)
```

```
↪-----
```

```

TypeError                                Traceback (most recent call last)

/tmp/ipykernel_29306/198808047.py in <module>
----> 1 factorial(2.5)

/tmp/ipykernel_29306/3438635131.py in factorial(n)
      2     '''Return factorial of a non-negative integer n.'''
      3     if not isinstance(n, int):
----> 4         raise TypeError('n must be int')
      5     if n < 0:
      6         raise ValueError('Cannot calculate factorial of a
negative number.')

```

TypeError: n must be int

return vs raise

- Oba ukončují běh funkce
- return x – úspěšné ukončení, **vrátí se** návratová hodnota x
- raise x – selhání, **vyhodí se** výjimka x

```

[30]: def fluffy(n):
      if n == 0:
          return 'OK'
      elif n == 1:
          return ValueError(n)
      else:
          raise ValueError(n)

```

```
[31]: fluffy(0)
```

```
[31]: 'OK'
```

```
[32]: fluffy(1)
```

```
[32]: ValueError(1)
```

```
[33]: fluffy(2)
```

→ -----

ValueError
↪ recent call last)

Traceback (most

```
/tmp/ipykernel_29306/1639457622.py in <module>
----> 1 fluffy(2)
```

```
/tmp/ipykernel_29306/1536240589.py in fluffy(n)
      5         return ValueError(n)
      6     else:
----> 7         raise ValueError(n)
```

ValueError: 2

- Pouhé raise lze využít v bloku except - znovu vyhazuje odchycenou výjimku

```
[34]: def reciprocal(x: float) -> float:
      try:
          return 1 / x
      except:
          print('Nastala výjimka.')
          raise
```

```
[35]: reciprocal(0)
```

Nastala výjimka.

↪ -----

ZeroDivisionError
↪ recent call last)

Traceback (most

```
/tmp/ipykernel_29306/367152496.py in <module>
----> 1 reciprocal(0)
```

```
/tmp/ipykernel_29306/1020027371.py in reciprocal(x)
      1 def reciprocal(x: float) -> float:
      2     try:
----> 3         return 1 / x
      4     except:
      5         print('Nastala výjimka.')
```

ZeroDivisionError: division by zero

Otázky:

Co vypíše následující program?

- A) Foo
0
Yikes
Ni
- B) Foo
0
Yikes
Boo
Ni
- C) Foo
0
Yikes
Foo
2
- D) Foo
0
Foo
Yikes
Boo
Oh

```
[ ]: def foo(x, y):  
    result = x // y  
    print('Foo')  
    return result  
  
def bar(n):  
    try:  
        a = foo(n, n-1)  
    except ZeroDivisionError:  
        print('Yikes')  
    finally:  
        return a  
  
def blob(n):  
    try:  
        for i in range(n):  
            print(bar(i))  
    except ZeroDivisionError:
```

```

        print('Boo')
    except ValueError:
        print('Bang')
    except:
        print('Ni')

try:
    blob(3)
except:
    print('Oh')

```

Testování

- Umožňuje automaticky zkontrolovat, jestli se funkce chová tak, jak očekáváme
- Test = popis očekávaného chování na konkrétním vstupu
- Výsledek testu:
 - *pass* - test prošel, chová se podle očekávání :)
 - *fail* - test neprošel, chová se jinak :(
- Moduly na testování: doctest, pytest, unittest...

doctest

- Testy v dokumentaci funkce
 - Vypadají jako interaktivní Python
- Modul doctest kontroluje, jestli se funkce chová tak, jak to popisují testy
- Více na <https://docs.python.org/3/library/doctest.html>
- Testy jsou zároveň dokumentací :)
- Nerozlišuje return, print a raise :(
- Neumí input :(

```

[1]: def cube_area(a: float) -> float:
      '''Vrací povrch krychle o straně a.
      >>> cube_area(1)
      6
      >>> cube_area(0.5)
      1.5
      ...
      return 6 * a**3

```

```

[2]: import doctest
      doctest.testmod()

```



```

*****
File "__main__", line 5, in __main__.cube_area
Failed example:
    cube_area(0.5)
Expected:
    1.5
Got:
    0.75
*****
1 items had failures:
  1 of  2 in __main__.cube_area
***Test Failed*** 1 failures.

```

[2]: `TestResults(failed=1, attempted=2)`

- Testování vyhození výjimky:
 - Kontroluje se pouze první a poslední řádek tracebacku

```

[1]: def factorial(n: int) -> int:
      '''Return the factorial of n, an exact integer >= 0.
      >>> [factorial(n) for n in range(6)]
      [1, 1, 2, 6, 24, 120]
      >>> factorial(-1)
      Traceback (most recent call last):
      ...
      ValueError: n must be >= 0
      ...
      if not n >= 0:
          raise ValueError('n must be >= 0')
      result = 1
      factor = 2
      while factor <= n:
          result *= factor
          factor += 1
      return result

```

```

[2]: import doctest
      doctest.testmod()

```

[2]: `TestResults(failed=0, attempted=2)`

- Testování hodnot typu float:
 - Potřeba zaokrouhlení nebo naformátování kvůli numerickým nepřesnostem

```

[1]: def foo(n, x):
      ...
      >>> foo(10, 1/10)

```

```

1.0
>>> round(foo(10, 1/10), 6)
1.0
...

suma = 0.0
for i in range(n):
    suma += x
return suma

```

```

[2]: import doctest
doctest.testmod()

```

```

*****
File "__main__", line 3, in __main__.foo
Failed example:
    foo(10, 1/10)
Expected:
    1.0
Got:
    0.9999999999999999
*****
1 items had failures:
  1 of  2 in __main__.foo
***Test Failed*** 1 failures.

```

```

[2]: TestResults(failed=1, attempted=2)

```

- Spuštění doctestu z příkazové řádky:
python -m doctest my_script.py
- Ukecaný mód (*verbose*) - vždy vypíše detaily ke každému testu a počet prošlých a všech testů:
python -m doctest -v my_script.py
- Doctesty mohou být v samostatném souboru, testy pak spustíme:
python -m doctest my_tests.txt
- I pokud projdou všechny testy, neznamená to, že funkce je správná!

```

[1]: def add(a, b):
    '''Vrať součet parametrů a, b.
    >>> add(2, 2)
    4
    >>> add(0, 0)
    0
    ...
    return a * b

```

```
[2]: import doctest
doctest.testmod()
```

```
[2]: TestResults(failed=0, attempted=2)
```

Debugging

- Proces odstraňování chyb (*bug* = chyba)
- Většina vývojových prostředí poskytuje pomůcky pro debugování

Debugging ve VSCode

- Funguje pouze ve skriptu (ne v Jupyter notebooku)
- Vytvoření *breakpointu* (červený puntík): kliknutím před číslo řádku (F9)
- Spuštění debugovacího režimu: Debug > Start Debugging (F5)
 - Zastaví vykonávání programu na breakpointu nebo při vyhození výjimky
 - Zobrazuje traceback a aktuální obsah proměnných
 - Možnost krokování

Příkaz assert

- Ověřuje, že náš předpoklad je splněn, jinak skončí chybou
- Spuštěním Pythonu s přepínačem `-0` (tj. `python3 -0 script.py`) ignorujeme všechny asserty

```
[1]: import math

a = 5
assert a >= 0, 'a must be non-negative'
print(math.sqrt(a))
```

```
2.23606797749979
```

```
[2]: a = -5
assert a >= 0, 'a must be non-negative'
print(math.sqrt(a))
```

```
↳ -----
AssertionError                                Traceback (most recent call last)
```

```

/tmp/ipykernel_29615/674718260.py in <module>
      1 a = -5
----> 2 assert a >= 0, 'a must be non-negative'
      3 print(math.sqrt(a))

```

AssertionError: a must be non-negative

```

[3]: def celsius_to_kelvin(temperature_C: float) -> float:
      ZERO_C = -273.15
      assert isinstance(temperature_C, (float, int)), 'temperature_C
      ↳ must be float'
      assert temperature_C >= ZERO_C, 'Colder than absolute zero!'
      return temperature_C - ZERO_C

```

```

[4]: celsius_to_kelvin(-300)

```

↳ -----

AssertionError Traceback (most recent call last)

```

/tmp/ipykernel_29615/2135464965.py in <module>
----> 1 celsius_to_kelvin(-300)

```

```

/tmp/ipykernel_29615/1847178153.py in
↳ celsius_to_kelvin(temperature_C)
      2 ZERO_C = -273.15
      3 assert isinstance(temperature_C, (float, int)),
↳ 'temperature_C must be float'
----> 4 assert temperature_C >= ZERO_C, 'Colder than absolute
↳ zero!'
      5 return temperature_C - ZERO_C

```

AssertionError: Colder than absolute zero!

Statická typová kontrola - modul mypy

- Umožňuje zkontrolovat, jestli funkce pracují se správnými typy
- Statická kontrola = kontrolujeme kód, aniž bychom ho spouštěli

Spuštění mypy

- Spuštění kontroly z příkazové řádky:
`python -m mypy my_program.py`
- Pokud nemáme nainstalovaný mypy, musíme ho nejdřív nainstalovat (pomocí modulu pip):
`python -m pip install mypy`
(Pokud nemáme ani pip: `sudo apt install python3-pip` (Ubuntu))

Rozšiřující učivo

pytest

- Modul pro testování (alternativa doctestu)
- Prochází všechny soubory `test_*` a složky `test*`
- Používá `assert` pro validaci
- Více na <https://docs.pytest.org/en/latest/>

```
[ ]: # inc.py:  
  
def inc(x):  
    return x + 1
```

```
[ ]: # test_inc.py:  
  
def test_answer():  
    assert inc(3) == 5
```

pytest - parametrize

```
[ ]: import pytest  
@pytest.mark.parametrize("test_input,expected", [  
    ("3+5", 8),  
    ("2+4", 6),  
    ("6*9", 42),  
)  
def test_eval(test_input, expected):  
    assert eval(test_input) == expected
```