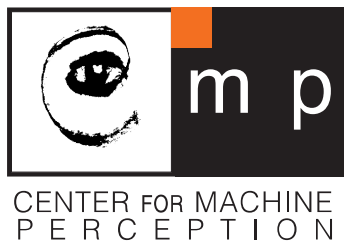
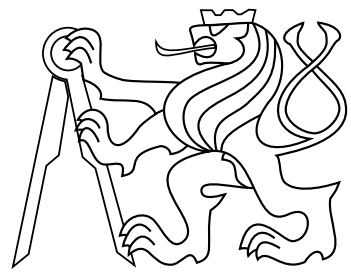
Funkce, řetězce, moduly

Jan Kybic

http://cmp.felk.cvut.cz/~kybic
kybic@fel.cvut.cz

2016





Funkce

Moduly

Řetězce

Náhodná čísla

Prostředky pro strukturování kódu

- **Bloky kódu** *(oddělené odsazením)*, např:

```
for i in range(10):
    print("Budu se pilně učit.")
```

- **Programy** = soubory `jméno.py`

Prostředky pro strukturování kódu

- **Bloky kódu** *(oddělené odsazením)*, např:

```
for i in range(10):
    print("Budu se pilně učit.")
```

- **Programy** = soubory `jméno.py`
- **Funkce**

Funkce

Příklady existujících funkcí:

```
>>> max(2,3)
3
>>> abs(-3.4)
3.4
>>> pow(2,3)
8
```

Funkce **vrací hodnotu** vypočítanou ze **vstupních argumentů**.

čistá funkce (pure function) = výstup závisí pouze na vstupních parametrech, a to jednoznačně.

Matematické funkce

Jsou k dispozici základní matematické funkce a konstanty

```
>>> import math

>>> math.sqrt(4.0)
2.0
>>> math.sin(30./180.*math.pi)
0.49999999999999994
>>> math.exp(1.0)
2.718281828459045
>>> math.log(math.e)
1.0
```

`math` je jméno modulu — vysvětlíme později.

Definice uživatelských funkcí

Příklad: $x \to x^2$, druhá mocnina

```
def square(x):
    return x*x
```

```
>>> square(3)
9
```

Definice uživatelských funkcí

Příklad: $x \rightarrow x^2$, druhá mocnina

```
def square(x):
    return x*x
```

```
>>> square(3)
9
```

Obecně:

```
  def name( parameters ):
    <code_block>
```

`return(value)` — návrat do nadřazené funkce

Příklad: Délka odvěsny

Pythagorova věta:

$$c = \sqrt{a^2 + b^2}$$

```
def hypotenuse(a,b):
    return math.sqrt(square(a)+square(b))
```

```
>>> hypotenuse(3,4)
5.0
```

Příklad: Délka odvěsny

Pythagorova věta:

$$c = \sqrt{a^2 + b^2}$$

```
def hypotenuse(a,b):
    return math.sqrt(square(a)+square(b))
```

```
>>> hypotenuse(3,4)
5.0
```

Složitější funkce volá jednodušší - skládání funkcí.

Příklad: Délka odvěsny

Pythagorova věta:

$$c = \sqrt{a^2 + b^2}$$

```
def hypotenuse(a,b):
    return math.sqrt(square(a)+square(b))
```

```
>>> hypotenuse(3,4)
5.0
```

Složitější funkce volá jednodušší - skládání funkcí.

Tato funkce už existuje...

```
>>> math.hypot(3,4)
5.0
```

Příklad 2: Opakování textu

Napište program, který vypíše text písně *Happy birthday* pro danou osobu, jejíž jméno dostane jako argument.

```
Terminal> python3 happy_birthday.py Mary
```

```
Happy birthday to you!
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Mary!
Happy birthday to you!
```

Příklad 2: Opakování textu – řešení

happy_birthday.py

```
# -*- coding: utf-8 -*-#
# Vytiskne text pisne 'Happy Birthday' pro danou osobu

import sys

def print_happy():
  print("Happy birthday to you!")

def print_happy_name(name):
  print("Happy birthday, dear %s!" % name)

# Zde začíná samotný program
print_happy()
print_happy()
print_happy()
print_happy_name(sys.argv[1])
print_happy()
```

Příklad 3: Exponenciální funkce

Ověřte, že pro dostatečně velké n:

$$e^x \approx \underbrace{\sum_{i=0}^{n} \frac{x^i}{i!}}_{A_n(x)}$$

Problém rozdělíme na podproblémy:

- Výpočet faktoriálu
- Výpočet součtů $A_n(x)$
- Tisk chyby pro různá n
- Tisk chyby pro různá x

Faktoriál

```
def factorial(n):
  prod=1
  for i in range(2,n+1):
    prod*=i
  return prod
```

Faktoriál

```
def factorial(n):
  prod=1
  for i in range(2,n+1):
    prod*=i
  return prod
```

Hned to vyzkoušíme $5! = ?$

```
>>> factorial(5)
120
```

Součet řady

$$A_n(x) = \sum_{i=0}^{n} \frac{x^i}{i!}$$

```
def series_sum(x,n):
  sum=0.
  for i in range(n+1):
    sum+=pow(x,i)/factorial(i)
  return sum
```

Součet řady

$$A_n(x) = \sum_{i=0}^{n} \frac{x^i}{i!}$$

```
def series_sum(x,n):
  sum=0.
  for i in range(n+1):
    sum+=pow(x,i)/factorial(i)
  return sum
```

Vyzkoušíme na $\mathrm{e}^1 = \mathrm{e} = ?$

Součet řady

$$A_n(x) = \sum_{i=0}^{n} \frac{x^i}{i!}$$

```
def series_sum(x,n):
  sum=0.
  for i in range(n+1):
    sum+=pow(x,i)/factorial(i)
  return sum
```

Vyzkoušíme na $e^1 = e =?$

```
>>> math.exp(1.0)
2.718281828459045
>>> series_sum(1.0,10)
2.7182818011463845
```

Vyhodnocení přesnosti pro různá *n*

```
def print_accuracy(x):
    exact=math.exp(x)
    print("x=%10g exact=%10g" % (x,exact))
    for n in [5,10,100]: # smyčka přes seznam
      approx=series_sum(x,n)
      relerr=abs(exact-approx)/exact
      print("    n=%5d approx=%10g relerr=%10g" %
         (n,approx,relerr))
```

```
>>> print_accuracy(1.0)
x=          1 exact=   2.71828
    n=    5 approx=   2.71667 relerr=0.000594185
    n=   10 approx=   2.71828 relerr=1.00478e-08
    n=  100 approx=   2.71828 relerr=1.63371e-16
>>> print_accuracy(3.0)
x=          3 exact=   20.0855
    n=    5 approx=       18.4 relerr= 0.0839179
    n=   10 approx=   20.0797 relerr=0.000292337
    n=  100 approx=   20.0855 relerr=3.53758e-16
>>> print_accuracy(-0.7)
x=       -0.7 exact=  0.496585
    n=    5 approx=  0.496437 relerr=0.000298815
    n=   10 approx=  0.496585 relerr=9.42331e-10
    n=  100 approx=  0.496585 relerr=1.11786e-16
```

Celý program je v souboru `exponencial.py`.

Postup vyhodnocování
(flow of execution)

Python

- zpracuje definice funkcí (ale neprovádí je)
- začne vykonávat hlavní program
- zavolá `print_accuracy` *(opakovaně)*
- `print_accuracy` volá `series_sum` *(opakovaně)*
- `series_sum` volá `factorial` *(opakovaně)*
- `factorial` vrací výsledek
- `series_sum` vrací výsledek
- `print_accuracy` končí
- hlavní program končí

úrovně odsazení = hloubka v *grafu volání (call graph)*

Otázky na doma

- Ve funkci `series_sum` počítejte sestupně od n do 1. Co pozorujete?
- Zkuste napsat kód, aby byl efektivnější, bez umocňování a bez funkce faktoriál.

Počet parametrů

```
def f(a,b,c):
    <code block>
```

- Počet parametrů je dán hlavičkou funkce.
- Existují i funkce bez parametrů
  - Nemůže být čistá, není-li konstantní

Počet parametrů

```
def f(a,b,c):
    <code block>
```

- Počet parametrů je dán hlavičkou funkce.
- Existují i funkce bez parametrů
  - Nemůže být čistá, není-li konstantní

 Existují i funkce s proměnným počtem parametrů: `print`, `max`, ...

Návratová hodnota

Příkazů `return` může být několik:

```
def isprime(n):
  p=2
  while p*p<=n:
    if n % p == 0:
        return False
    p+=1
  return True
```

Návratová hodnota

Příkazů `return` může být několik:

```
def isprime(n):
  p=2
  while p*p<=n:
    if n % p == 0:
        return False
    p+=1
  return True
```

```
>>> isprime(16)
False
>>> isprime(17)
True
```

Funkce nevracející výsledek
(void functions)

```
def f(x):
    y=2*x
```

Funkce nevracející výsledek
(void functions)

```
def f(x):
    y=2*x
```

```
>>> f(3)
```

`None` = "nic"
lze psát i explicitně: `return None`

Více návratových hodnot

```
def f(x):
    return x,2*x,3*x

a,b,c=f(10)
```

Výsledek a=10,b=20,c=30.

Více návratových hodnot

```
def f(x):
    return x,2*x,3*x

a,b,c=f(10)
```

Výsledek a=10,b=20,c=30.

`(x,2*x,3*x)` je objekt typu *n*-tice *(tuple)*.

Rozsah platnosti proměnných
(variable scope)

Proměnné definované

- v hlavním programu (mimo funkce) jsou **globální**, viditelné všude
- uvnitř funkce jsou **lokální**, viditelné pouze uvnitř této funkce
- lokální proměnná se stejným jménem **zastíní** proměnnou globální

Rozsah platnosti proměnných (2)

```
b=3 # je globální

def f(x):
    a=2*x # a je lokální
    print(a,b)
```

Rozsah platnosti proměnných (2)

```
b=3 # je globální

def f(x):
    a=2*x # a je lokální
    print(a,b)


>>> f(1)
(2, 3)
>>> print(b)
3
>>> print(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

☝

- Omezte viditelnost proměnných na nejnutnější část kódu.
- Omezte použití globálních proměnných.

Vedlejší efekty

Kromě vrácení hodnoty může funkce i **ovlivnit stav** (prostředí):

1. Výpis na obrazovku, zápis do souboru...
2. Změnu stavu proměnných volajícího

Už známe `print`, `print_accuracy`:

- Funkce není čistá *(pure)*, má **vedlejší efekty**.

☝ Je lepší vedlejší efekty omezit pouze na operace vstupu a výstupu.

Funkce jako argument

funkce vyššího řádu (high order functions)

```
def twice(f,x):
    return f(f(x))

def square(x):
    return x*x

print(twice(square,10))
```

Funkce jako argument

funkce vyššího řádu (high order functions)

```
def twice(f,x):
    return f(f(x))

def square(x):
    return x*x

print(twice(square,10))
```

```
10000
```

Funkce jako argument (2)

```
# volá funkci 'f()' 'n'-krát
def repeatNtimes(f,n):
    for i in range(n):
        f()

def ahoj():
    print("Ahoj")

repeatNtimes(ahoj,4)
```

Funkce jako argument (2)

```
# volá funkci 'f()' 'n'-krát
def repeatNtimes(f,n):
    for i in range(n):
        f()

def ahoj():
    print("Ahoj")

repeatNtimes(ahoj,4)
```

```
Ahoj
Ahoj
Ahoj
Ahoj
```

Anonymní funkce
lambda notace

Funkci lze definovat přímo v místě volání *(inline)*:

```
# aplikuje f(f(f(...f(x)..) 'n'-krát
def iterateNtimes(f,n,x):
    for i in range(n):
        x=f(x)
    return x

print(iterateNtimes(lambda x : 1. + 1./x,100,1.))
```

Anonymní funkce
lambda notace

Funkci lze definovat přímo v místě volání *(inline)*:

```
# aplikuje f(f(f(...f(x)..) 'n'-krát
def iterateNtimes(f,n,x):
    for i in range(n):
        x=f(x)
    return x

print(iterateNtimes(lambda x : 1. + 1./x,100,1.))
```

```
1.618033988749895
```

Anonymní funkce
lambda notace

Funkci lze definovat přímo v místě volání *(inline)*:

```
# aplikuje f(f(f(...f(x)..) 'n'-krát
def iterateNtimes(f,n,x):
    for i in range(n):
        x=f(x)
    return x

print(iterateNtimes(lambda x : 1. + 1./x,100,1.))
```

```
1.618033988749895
```

Toto číslo $\phi = \left(1+\sqrt{5}\right)/2$ *se nazývá 'zlatý řez'.*

Anonymní funkce
lambda notace

Funkci lze definovat přímo v místě volání *(inline)*:

```
# aplikuje f(f(f(...f(x)..) 'n'-krát
def iterateNtimes(f,n,x):
    for i in range(n):
        x=f(x)
    return x

print(iterateNtimes(lambda x : 1. + 1./x,100,1.))
```

```
1.618033988749895
```

ℹ Funkcionální programování = čisté funkce, funkce vyššího řádu, jednorázové přiřazení.

Funkce — shrnutí

- Pojmenovaná posloupnost příkazů,
- Má vstupní argumenty *(nemusí)*
- Vrací hodnotu *(nemusí)*

Matematická/čistá funkce = Zobrazení vstupních argumentů na výstupy

Příklad: Kvadratická rovnice

Řešení rovnice

$$ax^2 + bx + c = 0$$

je

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Napište funkci, která pro zadaná a,b,c vrátí reálná řešení x_1,x_2, pokud existují. Pokud neexistují, vrátí `None`.

Řešení kvadratické rovnice $ax^2 + bx + c = 0$

```
def solve_quadratic(a,b,c):
  d=b*b-4*a*c
  if d>=0.:
    ds=math.sqrt(d)
    return (-b-ds)/(2*a),(-b+ds)/(2*a)
  else:
    return None
```

Řešení kvadratické rovnice $ax^2 + bx + c = 0$

```
def solve_quadratic(a,b,c):
  d=b*b-4*a*c
  if d>=0.:
    ds=math.sqrt(d)
    return (-b-ds)/(2*a),(-b+ds)/(2*a)
  else:
    return None
```

Kontrola vyhodnocením $ax^2 + bx + c$

```
def eval_quadratic(a,b,c,x):
  return c+(b+a*x)*x
```

Vyzkoušíme

```
a=1.
b=-3.
c=2.
x1,x2=solve_quadratic(a,b,c)
print("x1=",x1,"x2=",x2)
print("Residuál pro x1 = ",eval_quadratic(a,b,c,x1))
print("Residuál pro x2 = ",eval_quadratic(a,b,c,x2))
print(solve_quadratic(1.,0.,1.))
```

Kód je v souboru `quadratic.py`

Vyzkoušíme

```
a=1.
b=-3.
c=2.
x1,x2=solve_quadratic(a,b,c)
print("x1=",x1,"x2=",x2)
print("Residuál pro x1 = ",eval_quadratic(a,b,c,x1))
print("Residuál pro x2 = ",eval_quadratic(a,b,c,x2))
print(solve_quadratic(1.,0.,1.))
```

Kód je v souboru `quadratic.py`

```
Terminal> python3 quadratic.py

x1= 1.0 x2= 2.0
Residuál pro x1 =  0.0
Residuál pro x2 =  0.0
None
```

Funkce

Moduly

Řetězce

Náhodná čísla

Moduly
(Modules)

- Modul obsahuje tématicky související funkce.
- Modul = soubor `.py`

Proč používat moduly:

- Program ve více souborech.
- Sdílení kódu mezi programy (znovupoužitelnost).
- Definice **rozhraní** *(API)*, skrytí implementačních detailů.
- Odstranění konfliktů jmen funkcí.

Moduly — jak je použít
(klient modulu)

1. Zpřístupnění modulu
   - soubor v aktuálním adresáři
   - soubor v systémových adresářích známých Pythonu
2. Import modulu:

   ```
   import math
   import sys
   ```

3. Použití *kvalifikovaného jména (tečková notace)*:
   `math.cos`, `sys.argv`

Moduly — jak je použít
(klient modulu)

1. Zpřístupnění modulu
   - soubor v aktuálním adresáři
   - soubor v systémových adresářích známých Pythonu
2. Import modulu:

```
import math
import sys
```

3. Použití *kvalifikovaného jména (tečková notace)*:
   `math.cos`, `sys.argv`

Existuje způsob jak oznámit Pythonu, kde moduly hledat.

Existuje způsob jak se kvalifikovaným jménům vyhnout.

Moduly — jak je psát

- Každý `.py` soubor lze `importovat`.
- Tím se vykonají všechny příkazy v souboru.

Moduly — jak je psát

- Každý `.py` soubor lze `importovat`.
- Tím se vykonají všechny příkazy v souboru.

```
>>> import quadratic
x1= 1.0 x2= 2.0
Residuál pro x1 =  0.0
Residuál pro x1 =  0.0
None
```

Moduly — jak je psát (2)

- Odstraníme globální kód
  - Testovací kód do zvláštních funkcí
  - Demonstrační kód do funkce `main()`

Moduly — jak je psát (2)

- Odstraníme globální kód
  - Testovací kód do zvláštních funkcí
  - Demonstrační kód do funkce `main()`

Zavoláme `main()` pokud je soubor volán jako skript, nikoliv pomocí `import`.

```
if __name__ == "__main__":
  main()
```

Moduly — jak je psát (3)

‘Modularizovaná’ verze `quadratic.py` je v `quadsolve.py`

Soubor `quadsolve.py` lze používat jako modul

```
>>> import quadsolve
>>> quadsolve.solve_quadratic(1.,-3.,2.)
(1.0, 2.0)
```

a lze ho spustit i samostatně:

```
Terminal> python3 quadsolve.py
```

```
Řešíme kvadratickou rovnici x^2-3x+2=0
x1= 1.0 x2= 2.0
```

Strukturovaný kód

- Dobrý kód $\neq$ dlouhá sekvence příkazů.
- Kód má být **hierarchicky strukturován**
  - **krátké sekvence příkazů** (kódu) řešící **dobře definovanou část úlohy** jsou sdružovány do **větších celků**
  - ty slouží jako **stavební kameny** pro složitější a větší podúlohy na **vyšší úrovni** atd.
  - *podobně sdružujeme i* **datové struktury**

Strukturovaný kód

- Dobrý kód $\neq$ dlouhá sekvence příkazů.
- Kód má být **hierarchicky strukturován**
  - **krátké sekvence příkazů** (kódu) řešící **dobře definovanou část úlohy** jsou sdružovány do **větších celků**
  - ty slouží jako **stavební kameny** pro složitější a větší podúlohy na **vyšší úrovni** atd.
  - *podobně sdružujeme i* **datové struktury**

☝ **D**o not **R**epeat **Y**ourself — neopakuj

☝ Pokud lze část kódu logicky oddělit, měla by být oddělena.

Výhody strukturovaného kódu

- přehlednost
- zkrácení kódu, úspora práce
- snadnost jednotlivých kroků
- znovupoužitelnost
- modifikovatelnost
- testovatelnost
- rychlost vývoje

Funkce

Moduly

Řetězce

Náhodná čísla

Řetězce (znaků)

Datové typy v Pythonu:

- Celá čísla (`int` / *integer*): `3`, `8`, ...
- Reálná čísla (`float` / *floating point number*): `128.`, `11.5`, ...
- Logické hodnoty (`bool` / *boolean*): `True`, `False`, ...
- **Řetězce** (`str` / *string*): `"Ahoj!"`, `'Ahoj!'`, ...

Řetězce (znaků)

Datové typy v Pythonu:

- Celá čísla (`int` / *integer*): `3`, `8`, ...
- Reálná čísla (`float` / *floating point number*): `128.`, `11.5`, ...
- Logické hodnoty (`bool` / *boolean*): `True`, `False`, ...
- **Řetězce** (`str` / *string*): `"Ahoj!"`, `'Ahoj!'`, ...

Všimněte si dvou možných oddělovačů řetězců.

Řetězec = složený typ

- **Jednoduché typy:** celé číslo, reálné číslo, logická hodnota,
  - Nelze rozložit na podobjekty
- **Složený typ:** řetězec = posloupnost znaků

Řetězec = složený typ

- **Jednoduché typy:** celé číslo, reálné číslo, logická hodnota,
  - Nelze rozložit na podobjekty
- **Složený typ:** řetězec = posloupnost znaků

Příklad:

```
>>> s='Ahoj lidi'
>>> s[0]
'A'
>>> s[1]
'h'
>>> len(s)
9
```

Řetězec = složený typ

- **Jednoduché typy:** celé číslo, reálné číslo, logická hodnota,
  - Nelze rozložit na podobjekty
- **Složený typ:** řetězec = posloupnost znaků

Příklad:

```
>>> s='Ahoj lidi'
>>> s[0]
'A'
>>> s[1]
'h'
>>> len(s)
9
```

☝ Pozice jsou číslované od 0.

Spojování řetězců

```
a='Czech Technical University'
b='Prague'
c=a+' in '+b
```

Jaká je hodnota *c*?

Spojování řetězců

```
a='Czech Technical University'
b='Prague'
c=a+' in '+b
```

Jaká je hodnota *c*?

```
>>> c
'Czech Technical University in Prague'
```

Spojení řetězců může být pomalé.

Spojování řetězců

```
a='Czech Technical University'
b='Prague'
c=a+' in '+b
```

Jaká je hodnota *c*?

```
>>> c
'Czech Technical University in Prague'
```

Spojení řetězců může být pomalé.

Další vlastnosti řetězců

Python nemá zvláštní typ pro znaky.

Řetězce v Python jsou neměnné *(immutable)*.

Funkce

Moduly

Řetězce

Náhodná čísla

Náhodná čísla

Generování náhodných celých čísel mezi 0 a $n-1$:

`random.randrange(0,n)`

```
>>> import random
>>> n=10
>>> random.randrange(0,n)
1
>>> random.randrange(0,n)
4
>>> random.randrange(0,n)
1
```

Příklad: Simulace házení mincí

```
def hod():
    if random.randrange(2)==0:
        print("Hlava")
    else:
        print("Orel")
for i in range(5):
  hod()
```

Příklad: Simulace házení mincí

```
def hod():
    if random.randrange(2)==0:
        print("Hlava")
    else:
        print("Orel")
for i in range(5):
  hod()
```

```
Orel
Hlava
Hlava
Orel
Hlava
```

Příklad: Průměr diskrétní náhodné proměnné

Vybíráme náhodně čísla $x_i \in \{1, 2, \ldots, 10\}$.
Jaký bude jejich průměr $\frac{1}{N}\sum_{i=1}^{N} x_i$ pro velké N?

```
N=10000
s=0.
for i in range(N):
  s+=random.randrange(1,11)
print("average=",s/N)
```

Příklad: Průměr diskrétní náhodné proměnné

Vybíráme náhodně čísla $x_i \in \{1, 2, \ldots, 10\}$.
Jaký bude jejich průměr $\frac{1}{N}\sum_{i=1}^{N} x_i$ pro velké N?

```
N=10000
s=0.
for i in range(N):
  s+=random.randrange(1,11)
print("average=",s/N)
```

```
average= 5.544
```

Funkce

Moduly

Řetězce

Náhodná čísla