# 3. Algoritmy pro práci s čísly

## BAB37ZPR – Základy programování

Stanislav Vítek

Katedra radioelektroniky
Fakulta elektrotechnická
České vysoké učení v Praze

# Přehled témat

- Část 1 – Funkce
  - Struktura programu
  - Vlastnosti funkcí
  - Ladění
- Část 2 – Textové řetězce
- Část 3 – Náhodná čísla
- Část 4 – Algoritmy pracující s čísly

# Část I

# Funkce

# I. Funkce

# Prostředky pro strukturování kódu

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

```
for i in range(10):
  print("Budu se pilne ucit.")
```

- **Programy** – soubory `jmeno.py`
- **Funkce**

## Příklady existujících funkcí

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

- Funkce **vrací hodnotu** vypočítanou ze **vstupních argumentů**.
- **Čistá funkce** (pure function) – výstup závisí pouze na vstupních parametrech, a to jednoznačně.

# Matematické funkce – knihovna `math`

- použití knihovny: `import math`
- zaokrouhlování: `round`, `math.ceil`, `math.floor`
- absolutní hodnota: `abs`
- `math.exp`, `math.log`, `math.sqrt`
- goniometrické funkce: `math.sin`, `math.cos`, ...
- konstanty: `math.pi`, `math.e`

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

# Definice uživatelských funkcí

- Příklad – druhá mocnina

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

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

- Obecně:

```
def jmeno ( parametry ):
  <blok kodu>
```

- `return(value)` – návrat do nadřazené funkce
- problémy bychom měli členit na podproblémy – zde např. můžeme funkci využít pro výpočet přepony

# Příklad – opakování textu

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

lec03/happy_birthday.py

```
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!
```

## Poznámky

- argument příkazového řádku získáme jako `sys.argv[1]`
- abychom nemuseli z IDLE, lze argumenty zfalšovat

```
sys.argv = [sys.argv[0], 'argument1']
```

# Exponenciální funkce

Ověřte, že pro dostatečně velké n platí

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

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
```

```
>>> 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
```

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

# Vyhodnocení přesnosti

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

lec03/exponencial.py

```
>>> 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
```

# I. Funkce

# Počet parametrů

```
def funkce(a, b, c):
  <blok kodu>
```

- Počet parametrů je dán hlavičkou funkce
- Existují i funkce bez parametrů

Nemůže být čistá, pokud není konstatní.

- Pamametry mohou mít defaultní hodnotu

Existují i funkce s proměnným počtem parametrů

# Návratová hodnota

Příkazů `return` může být ve funkci 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 nemusí vracet nic, pak často hovoříme o proceduře nebo void funkci. V proceduře můžeme explicitně specifikovat **return None**

# Více návratových funkcí

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

```
>>> a, b, c = f(10)
>>> a
10
>>> b
20
>>> c
30
```

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

# Rozsah platnosti proměnných

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

```
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)
NameError: name 'a' is not defined
```

# Funkce jako argument

- 1. příklad

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

def square(x):
  return x*x

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

- 2. příklad

```
def repeatNtimes(f,n):
  for i in range(n): f()

def ahoj():
  print("Ahoj")

repeatNtimes(ahoj,4)
```

# I. Funkce

Struktura programu

Vlastnosti funkcí

Ladění

# Poznámka o ladění

- laděním se nebudeme (na přednáškách) příliš zabývat
- to ale neznamená, že není důležité ...

> Ladění je dvakrát tak náročné, jak psaní vlastního kódu. Takže pokud napíšete program tak chytře, jak jen umíte, nebudete schopni jej odladit. (Brian W. Kernighan)

## Jak na to?

- ladící výpisy
  - např. v každé iteraci cyklu vypisujeme stav proměnných
  - doporučeno vyzkoušet na ukázkových programech z přednášek
- použití debuggeru
  - dostupný přímo v IDLE
  - sledování hodnot proměnných, spuštěných příkazů, breakpointy, ...
- komnpozice na funkce
  - chyba se daleko lépe hledá v dílčí funkci než v celém programu najednou

# Čtení chybových hlášek

```
Traceback (most recent call last):
File "sorting.py", line 63, in <module>
test_sorts()
File "sorting.py", line 59, in test_sorts
sort(a)
File "sorting.py", line 52, in insert_sort
a[j] = curent
NameError: name 'curent' is not defined
```

- kde je problém? (identifikace funkce, číslo řádku)
- co je za problém (typ chyby)

# Základní typy chyb

**SyntaxError**
- invalid syntax: zapomenutá dvojtečka či závorka, záměna `=` a `==`, ...
- EOL while scanning string literal: zapomenutá uvozovka

**NameError** – špatné jméno proměnné (překlep v názvu, chybějící inicializace)

**IndentationError** – špatné odsazení

**TypeError** – nepovolená operace (sčítání čísla a řetězce, ...)

**IndexError** – chyba při indexování řetězce, seznamu, ...

## Časté chyby

- zapomenuté formátovací znaky – dvojtečka, apostrof, ...
- špatný počet argumentů funkce
- `"True"` místo `True`
- záměna `print` a `return`

# Část II

## Textové řetězce

# Řetězce a znaky – ukázky operací

```
"kos" * 3
"petr" + "klic"
text = "velbloud"
len(text)
text[0]
text[2]
text[-1]
ord('b')
chr(99)
```

# Základní pravidla

## Uvozovky, apostrofy

- C, Java: uvozovky pro řetězce, apostrofy pro znaky
- Python: lze používat uvozovky i apostrofy
- PEP8: hlavně konzistence

## Proč indexujeme od 0?

- řada celkem dobrých důvodů
- více
  - Why numbering should start at zero (Edsger W. Dijkstra)
  - `http://www.cs.utexas.edu/users/EWD/transcriptions/EWD08xx/EWD831.html`
  - `http://programmers.stackexchange.com/questions/110804/why-are-zero-based-arrays-the-norm`

# Kódování

- Jak jsou znaky reprezentovány?

  ASCII, ISO 8859-2, Windows-1250, Unicode, UTF-8, ...

  `http://www.joelonsoftware.com/articles/Unicode.html`
- Python3 – Unicode řetězce
- My budeme používat jen znaky bez diakritiky
  - `ord`, `chr` – převod znaků na čísla a zpět
  - anglická abeceda má přiřazena po sobě jdoucí čísla

```
for i in range(26):
  print(chr(ord('A')+i))
```

# Další vlastnosti

- indexování

```
text = "velbloud"
text[:3] # prvni 3 znaky
text[3:] # od 3 znaku dale
text[1:8:2] # od 2.  znaku po 7.  krok po 2
text[::3] # od zacatku do konce po 3
```

- neměnitelné (immutable) – rozdíl oproti seznamům a oproti řetězcům v některých jiných jazycích
- změna znaku – vytvoříme nový řetězec

```
text = "kopec"
text[2] = "n" # chyba
text = text[:2] + "n" + text[3:]
```

# Formátovací funkce

```
text = "i Have a dream."
print(text.upper())
print(text.lower())
print(text.capitalize())
print(text.rjust(30))
print("X", text.center(30), "X")
print(text.replace("dream", "nightmare"))
```

... a mnoho dalších, více později (nebo dokumentace)

# Část III

## Náhodná čísla

# Náhodná čísla

- přesněji: pseudo-náhodná čísla
- opravdová náhodná čísla: `https://www.random.org/`
- bohaté využití v programování: výpočty, simulace, hry, ...
- Python – modul `random`
  - random.random() – float od 0 do 1
  - random.randint(a, b) – celé číslo mezi a, b
  - random.randrange(a, b) – náhodná celá čísla v rozsahu a, b

```
import random

n=10
random.randrange(0,n)
random.randrange(0,n)
random.randrange(0,n)
```

# Příklad: Simulace házení mincí

```
import random

def hod():
  if random.randrange(2)==0:
    print("Hlava")
  else:
    print("Orel")

# trochu jiny zapis funkce
def hod1():
  text = "O" if random.randrange(2) % 2 else "H"
  print(text)

for i in range(5):
  hod()
```

# 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 $N$?

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

# Simulace volebního průzkumu 1/2

- volební průzkumy se často liší – jaká je jejich přesnost?
- přístup 1: matematické modely, statistika
- přístup 2: simulace
- program:
  - vstup: preference stran, velikost vzorku
  - výstup: preference zjištěné v náhodně vybraném vzorku

# Simulace volebního průzkumu 2/2

```
def pruzkum(size, pref1, pref2, pref3):
  count1 = 0
  count2 = 0
  count3 = 0
  for i in range(size):
    r = random.randint(1, 100)
    if r <= pref1: count1 += 1
    elif r <= pref1 + pref2: count2 += 1
    elif r <= pref1 + pref2 + pref3: count3 += 1
  print("Strana 1:", 100 * count1 / size)
  print("Strana 2:", 100 * count2 / size)
  print("Strana 3:", 100 * count3 / size)
```

- řešení není dobré: funguje jen pro 3 strany
- lepší řešení – využití seznamů (příští přednáška)

# Kámen, nůžky, papír

```
def strategy_uniform():
  r = random.randint(1, 3)
  if r == 1:
    return "R"
  elif r == 2:
    return "S"
  else:
    return "P"

def strategy_rock():
  return "R"
```

# Kámen, nůžky, papír – vyhodnocení tahu

```
def evaluate(symbol1, symbol2):
  if symbol1 == symbol2:
    return 0
  if symbol1 == "R" and symbol2 == "S" or \
  symbol1 == "S" and symbol2 == "P" or \
  symbol1 == "P" and symbol2 == "R":
    return 1
  return -1
```

# Kámen, nůžky, papír – tah

```
def game(rounds):
  points = 0
  for i in range(1, rounds+1):
    print("Round ", i)
    symbol1 = strategy_uniform()
    symbol2 = strategy_uniform()
    print("Symbols:", symbol1, symbol2)
    points += evaluate(symbol1, symbol2)
    print("Player 1 points:", points)
```

# Obecnější strategie

```
def strategy(weightR, weightS, weightP):
  r = random.randint(1, weightR + weightS + weightP)
  if r <= weightR:
    return "R"
  elif r <= weightR + weightS:
    return "S"
  else:
    return "P"
```

## Jak hru ještě vylepšit?

- turnaj různých strategií
- strategie pracující s historií
  - kopírování posledního tahu soupeře
  - analýza historie soupeře (hraje vždy kámen? → hraj papír!)
- rozšíření na více symbolů

# Část IV

## Algoritmy pracující s čísly

# Číselné typy

- `int` – celá čísla
- `float`
  - floating-point number
  - čísla s plovoucí desetinnou čárkou
  - reprezentace: mantisa $\times$ báze $^{exponent}$
  - nepřesnosti, zaokrouhlování
- `complex` – komplexní čísla

## Nepřesnosti

$$((1 + \frac{1}{x}) - 1) * x = 1$$

```
>>> x = 2**50
>>> ((1 + 1 / x) - 1) * x
1.0
>>> x = 2**100
>>> ((1 + 1 / x) - 1) * x
0.0
```

# Číselné typy – poznámky

- explicitní přetypování: `int(x)`, `float(x)`
- automatické nafukování typu `int`
  - viz např. `2**100`
  - pomalejší, ale korektní
  - v ostatních programovacích jazycích zpravidla dochází k přetečení

## Kvíz

```
n = 1
while n > 0:
  print(n)
  n = n / 10
print("done")
```

- Co udělá program?
- Co když změníme výraz na `n=n*10`
- Co když změníme výraz na `n=n*10.0`
- Jaký bude výsledek programu v jiných jazycích?

# Příklad – ciferný součet

- vstup: číslo `x`
- výstup: ciferný součet čísla `x`
- příklady: $8 \to 8$, $15 \to 6$, $297 \to 18$

## Jak na to?

- opakovaně provádíme:
  - dělení `10` se zbytkem – hodnota poslední cifry
  - celočíselné dělení – okrajování čísla

## Naivní řešení

```
if n % 10 == 0:
  f = 0 + f
elif n % 10 == 1:
  f = 1 + f
elif n % 10 == 2:
  f = 2 + f
...
```

# Příklad – ciferný součet – řešení

```
def digit_sum(n):
  result = 0
  while n > 0:
    result += n % 10
    n = n // 10
  return result
```

## Pro zajímavost

```
def digit_sum(n):
  return sum(map(int, str(n)))
```

## Připomenutí

- `print` – výpis hodnoty
- `return` – návratová hodnota funkce, se kterou lze dále pracovat

# Příklad – Collatzova posloupnost

- vezmi přirozené číslo:
  - pokud je sudé, vyděl jej dvěma
  - pokud je liché, vynásob jej třemi a přičti jedničku
- tento postup opakuj, dokud nedostaneš číslo jedna

## Řešení

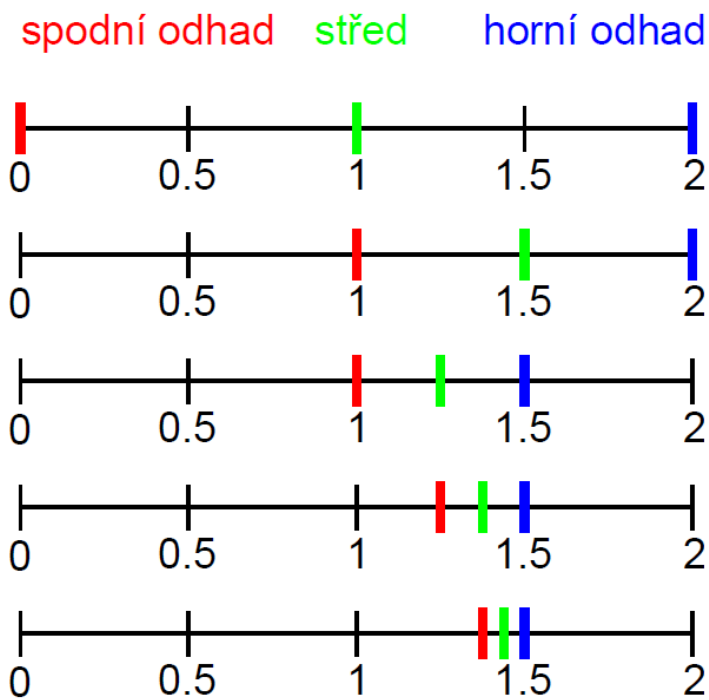
```
def collatz_sequence(n):
  while n != 1:
    print(n, end=", ")
    if n % 2 == 0:
      n = n // 2
    else:
      n = 3*n + 1
  print(1)
```

# Výpočet odmocniny

- vstup: číslo `x`
- výstup: odhad $\sqrt{(x)}$

## Jak na to?

- Existuje mnoho metod, ukázka metody binárního půlení



# Výpočet odmocniny – binární půlení

```
def square_root(x, precision=0.01):
  upper = x
  lower = 0
  middle = (upper + lower) / 2
  while abs(middle**2 - x) > precision:
    print(lower, upper, sep="\t")
    if middle**2 > x:
      upper = middle
    if middle**2 < x:
      lower = middle
    middle = (upper + lower) / 2
  return middle
```

# Výpočet odmocniny – binární půlení

Drobný problém: program není korektní
Kde je chyba?

- Funguje korektně jen pro čísla $x \geq 1$
- Co program udělá pro čísla $x < 1$
- Proč?
- Jak to opravit?