3. Složené datové typy - textové řetezce, pole

BAB37ZPR – Základy programování

Stanislav Vítek

Katedra radioelektroniky
Fakulta elektrotechnická
České vysoké učení v Praze

Přehled témat

- Část 1 – Pole
  - Textové řetězce
  - Pole
  - Hodnoty a reference
  - Dourozměrné pole
- Část 2 – Funkce a proměnné
  - Globální a lokální proměnné
  - Funkce
  - Ošetření chyb

Příklad na zopakování: 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

```
1  def factorial(n):
2    prod=1
3    for i in range(2,n+1):
4      prod*=i
5    return prod
6
```

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

Součet řady

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

```
1  def series_sum(x,n):
2    sum=0.
3    for i in range(n+1):
4      sum+=pow(x,i)/factorial(i)
5    return sum
6
```

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

Vyhodnocení přesnosti

```
1  def print_accuracy(x):
2    exact = math.exp(x)
3    print("x=%10g exact=%10g" % (x,exact))
4    for n in [5,10,100]: # smycka pres seznam
5      approx=series_sum(x,n)
6      relerr=abs(exact-approx)/exact
7      print("    n=%5d approx=%10g relerr=%10g" % (n,approx,relerr))
8
```

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
```

Část I

Složené datové typy

Datové typy v Pythonu

Jednoduché typy

- celé číslo, reálné číslo, logická hodnota

primitive data type

Složené typy / datové struktury

- textový řetězec, *n*-tice *(tuple)*, **pole**

composite/compound data type, data structures

- Hierarchicky sdružují data
- Související data jsou uložena a manipulována spolu
- Pro zvýšení efektivity programování i vykonávání
- Operace na datových strukturách
  - vytvoření
  - čtení a modifikace jednotlivých složek (elementů)
  - vyhledávání, přidávání, odebírání, ...

I. Složené datové typy

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`
  - `https://www.quora.com/Why-do-array-indexes-start-with-0-zero-in-many-programming-languages`

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

```
1  for i in range(26):
2    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

```
1  text = "i Have a dream."
2  print(text.upper())
3  print(text.lower())
4  print(text.capitalize())
5  print(text.rjust(30))
6  print("X", text.center(30), "X")
7  print(text.replace("dream", "nightmare"))
```

... a mnoho dalších, více v dokumentaci Pythonu

I. Složené datové typy

Pole

- Obsahuje `N` elementů (objektů, prvků), indexovaných od `0`
- Přímý přístup *(random access)*
  - Pro čtení i zápis, v konstantním čase
- Vytvoření

```
a = [0.3,0.6,0.1]
a
type(a)
```

- Čtení prvku

```
a[1]
a[0]
```

- Změna prvku

```
a[2] = 1.5
a
```

Odbočka – pole nebo seznam?

- Python nemá vestavěný typ, který by odpovídal standardní představě od poli
  - prvky pole jednoho datového typu
  - zaručena adresa a velikost prvků v paměti
- Seznam v Pythonu je významně flexibilnější
- Lze využít modul `array`

```
1 import array as arr
2 a = arr.array('d', [1.1, 3.5, 4.5])
3 print(a)
```

- Datové typy rozšiřují typy dostupné v Pythonu
  - `d` – double, `f` – float, `i` – signed int, `I` – unsigned int, `b` – signed char, `B` – unsigned char, `u` – Unicode, `h` – signed short, `H` – unsigned short, `l` – signed long, `L` – unsigned long

Seznamu s prvky stejného datového typu můžeme říkat pole.

Operace s polem

- Výpis pole

```
1 a=[0.3,0.6,0.1]
2 print(a)
```

```
[0.3, 0.6, 0.1]
```

- Index může být výraz

```
1 s=0.
2 for i in range(3):
3     s+=a[i]
4     print("a[%d]=%f" % (i,a[i]))
5 print(s)
```

```
a[0]=0.300000
a[1]=0.600000
a[2]=0.100000
0.9999999999999999
```

Pole různých typů

- Homogenní pole → všechny prvky jsou stejného typu.

```
1  a = [0.3, 0.6, 0.1]
2  print (a[0])
4  b = [3, 1, 4, 1, 5, 9, 2]
5  print (b[2])
7  barvy = ["srdce", "listy", "kule", "zaludy"]
8  print (barvy[3])
10 bits = [True,False]
11 print (bits)
```

- Nehomogenní pole

```
1  a = [1, 3.14, "ahoj", [1, 2, 3]]
2  print (a)
```

Funkce a pole – unární operace

```
>>> a = [0.3, 0.6, 0.1]
>>> print(a)
[0.3, 0.6, 0.1]
>>> len(a)
3
>>> sum(a)
1
>>> max(a)
0.6
>>> bits = [True, False]
>>> all(bits)
False
>>> any(bits)
True
```

Funkce a pole – binární operace

- Spojování, opakování

```
>>> a = [0.3, 0.6, 0.1]
>>> b = [0.7, 0.9]
>>> a+b
[0.3, 0.6, 0.1, 0.7, 0.9]
>>> b*3
[0.7, 0.9, 0.7, 0.9, 0.7, 0.9]
```

Vytvoření pole 1/3

- Výčtem

```
>>> a=[0.3,0.6,0.1]
```

- Opakováním prvků

```
>>> a=10*[0]
>>> a
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

- Opakování používejte pouze pro primitivní nebo neměnné typy.

```
>>> a=3*[[1,2]]
>>> a
[[1, 2], [1, 2], [1, 2]]
>>> a[1][0]=10
>>> a
[[10, 2], [10, 2], [10, 2]]
```

Vytvoření pole 2/3

- Z posloupnosti

```
1 list(range(1,11))
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

V Pythonu se tento druh pole jmenuje `list`

- Přidáváním na konec

```
1 a=[]
2 for i in range(10):
3     a+=[0.0]
4 print(a)
```

```
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

Přidávání může být časově náročné.

Vytvoření pole 3/3

- Výrazem *(list comprehension)*

```
[i for i in range(1,11)]
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
[i*i for i in range(1,11)]
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

```
[0. for i in range(1,11)]
```

```
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

Elegantní, ale specialita Pythonu — nemusíte si pamatovat.

Indexace

`x[i]`

- *i*-tý prvek pole `x`
- pro sekvenční typy: pole, řetězce, *n*-tice

```
1  a = [2, 7, 1]
2  print(a[2])
4  s = "Ferda"
5  print(s[2])
7  t = (1,2)
8  print(t[0])
```

Indexace – záporné indexy

`x[i] = x[len(x)+i]`

Specialita Pythonu.

```
1  a=[6,7,5,2,9]
```

```
>>> a[-1]
9
>>> a[2]
5
>>> a[-2]
2
```

Řezy pole

```
x[i:j] = [x[i], x[i+1], ..., x[j-1]
x[i:]  = x[i:len(x)]
x[:j] = x[0:j]
x[:]  = x[0:len(x)]=x
```

Specialita Pythonu, podobný přístup např. v Matlabu

- Příklad:

```
1 a=[6,7,5,2,9]
2 print(a[2:4])
```

```
[5, 2]
```

```
1 print(a[:3])
```

```
[6, 7, 5]
```

Příklad: jména dnů v týdnu

Úkol: převeďte $i \in \{0, \ldots, 6\}$ na jméno dne.

```
1  def jmeno_dne(i):
2    if i==0: return "pondeli"
3    elif i==1: return "utery"
4    elif i==2: return "streda"
5    elif i==3: return "ctvrtek"
6    elif i==4: return "patek"
7    elif i==5: return "sobota"
8    elif i==6: return "nedele"
9    else: return "???"
11 print(jmeno_dne(3))
```

```
ctvrtek
```

Příklad: jména dnů v týdnu – pomocí pole

```
1  jmena_dni=["pondeli","utery","streda","ctvrtek",
2             "patek","sobota","nedele"]
4  print(jmena_dni[3])
```

Uzávorkovaný výraz lze rozdělit na více řádek.

```
ctvrtek
```

```
1  def jmeno_dne(i):
2      return jmena_dni[i]
4  print(jmeno_dne(3))
```

```
ctvrtek
```

Příklad: průměr a směrodatná odchylka 1/3

Odhad ze vzorků

$$\mu_x = \frac{1}{N}\sum_{i=1}^{N} x_i, \qquad \sigma_x = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \mu_x)^2}$$

```
1  def mean(v):
2    "Calculate a mean of a vector"
3    s=0.
4    for i in range(len(v)):
5      s+=v[i]
6    return(s/len(v))
```

Řetězec za hlavičkou funkce slouží k dokumentaci. Lze ho zobrazit příkazem `help(mean)`.

Příklad: průměr a směrodatná odchylka 2/3

$$\mu_x = \frac{1}{N}\sum_{i=1}^{N} x_i, \qquad \sigma_x = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \mu_x)^2}$$

```
1  import math
3  def stdev(v):
4    "Calculate a corrected sample standard deviation"
5    m=mean(v)
6    s=0.
7    for i in range(len(v)):
8      s+=(v[i]-m)**2
9    return math.sqrt(s/(len(v)-1))
```

Příklad: průměr a směrodatná odchylka 3/3

Vypočítáme μ, σ pro $x_1 = 0, \ldots, x_{1001} = 1000$

```
1  a=list(range(1001))
2  print("mean=", mean(a), " sigma=", stdev(a))
```

```
mean= 500.0  sigma= 289.10811126635656
```

Pro spojitou uniformní distribuci $[0, 1000]$ je

- $\mu = 500$ a
- $\sigma = \frac{1000}{\sqrt{12}} \approx 288.67$

Pole jako argument cyklu 1/2

- Argumentem cyklu `for` může být *sekvence*, například pole.
  - místo

```
def mean(v):
  "Calculate a mean of a vector"
  s=0.
  for i in range(len(v)):
    s+=v[i]
  return(s/len(v))
```

  - můžeme psát

```
def mean(v):
  "Calculate a mean of a vector"
  s=0.
  for x in v:
    s+=x
  return(s/len(v))
```

Pole jako argument cyklu 2/2

- Výsledek je stejný

```
1  def mean(v):
2    "Calculate a mean of a vector"
3    s=0.
4    for x in v:
5      s+=x
6    return(s/len(v))
7
9  a=list(range(1001))
10 print("mean=", mean(a))
```

```
mean=500
```

Použití funkcí pole 1/3

- Pokud můžete, používejte existující funkce definované v rámci třídy `list`
  - místo

```
def mean(v):
  "Calculate a mean of a vector"
  s=0.
  for x in v:
    s+=x
  return(s/len(v))
```

  - můžeme psát

```
def mean(v):
  "Calculate a mean of a vector"
  return(sum(v)/len(v))
```

Použití funkcí pole 2/3

- Ještě jednou směrodatná odchylka
  - místo:

```
def stdev(v):
  "Calculate a corrected sample standard deviation"
  m=mean(v)
  s=0.
  for i in range(len(v)):
    s+=(v[i]-m)**2
  return math.sqrt(s/(len(v)-1))
```

  - můžeme psát

```
def stdev(v):
  "Calculate a corrected sample standard deviation"
  m=mean(v)
  s=sum([(x-m)**2 for x in v])
  return math.sqrt(s/(len(v)-1))
```

Použití funkcí pole 3/3

- Funkci

```
1  def stdev(v):
2    "Calculate a corrected sample standard deviation"
3    m=mean(v)
4    s=sum([(x-m)**2 for x in v])
5    return math.sqrt(s/(len(v)-1))
```

- lze dále zkrátit

```
1  def stdev(v):
2    return math.sqrt(sum([(x-mean(v))**2 for x in v])/
3                         (len(v)-1))
4  a=list(range(1001))
5  print("mean=",mean(a)," sigma=",stdev(a))
```

```
mean= 500.0  sigma= 289.10811126635656
```

Výsledek se nezměnil. Zkracování ovšem může zhoršit srozumitelnost kódu.

Příklad: Náhodná permutace 1/3

Problém

Vytvořte náhodnou permutace čísel $0, 1, \ldots, N-1$

Jak na to?

- Uložíme do pole počáteční permutací $0, 1, \ldots, N-1$
- Budeme *vyměňovat* vždy dva prvky
- Aktuální prvek $i = 0, \ldots, N-2$ vyměníme s náhodně vybraným prvkem na pozici $j = i, i+1, \ldots, N-1$

Příklad: Náhodná permutace 2/3

- Využijeme generátor náhodných čísel z modulu `random`

```
1  import random
3  def permutation(n):
4      "Create a random permutation of integers 0..n-1"
5      p=list(range(n))
6      for i in range(n-1):
7          r=random.randrange(i,n)
8          temp=p[r]
9          p[r]=p[i]
10         p[i]=temp
11     return(p)
```

Příklad: Náhodná permutace 3/3

- Vyzkoušíme:

```
print(permutation(10))
```

```
[9, 5, 3, 2, 4, 8, 0, 6, 1, 7]
```

```
print(permutation(10))
```

```
[7, 0, 6, 4, 2, 9, 3, 8, 5, 1]
```

```
print(permutation(10))
```

```
[5, 8, 1, 7, 0, 9, 6, 4, 2, 3]
```

Kontrolní tisky

- Velmi obecná a užitečná technika ověření funkčnosti programů.
- Jak to vlastně funguje? Doplníme na vhodné místo výpis aktuálního stavu výsledku

```
1  import random
3  def permutation(n):
4      "Create a random permutation of integers 0..n-1"
5      p=list(range(n))
6      print("p=",p)
7      for i in range(n-1):
8          r=random.randrange(i,n)
9          temp=p[r]
10         p[r]=p[i]
11         p[i]=temp
12         print("i=%d r=%d p=%s" % (i,r,str(p)))
13     return(p)
```

Kontrolní tisky 2/2

```
1 permutation(5)
```

```
p= [0, 1, 2, 3, 4]
i=0 r=0 p=[0, 1, 2, 3, 4]
i=1 r=4 p=[0, 4, 2, 3, 1]
i=2 r=3 p=[0, 4, 3, 2, 1]
i=3 r=4 p=[0, 4, 3, 1, 2]
```

Příklad: Permutace pole

- Vytiskneme prvky pole v náhodném pořadí:

```
1  barvy=["srdce","listy","kule","zaludy"]
2  p=permutation(len(barvy))
4  for i in range(len(barvy)):
5      print(barvy[p[i]], end=" ")
```

```
zaludy kule listy srdce
```

- Pole v novém pořadí:

```
1  print([barvy[i] for i in p])
```

```
['zaludy', 'kule', 'listy', 'srdce']
```

Příklad: Házení dvěma kostkami

Jaká je pravděpodobnost, že padne součet s?

$$P(s) = \frac{\text{počet příznivých}}{\text{počet celkem}} = \frac{6 - |s - 7|}{6^2}$$

s	počet možnosti	P(s)
2	1	0.028
3	2	0.056
4	3	0.083
5	4	0.111
6	5	0.139
7	6	0.167
8	5	0.139
9	4	0.111
10	3	0.083
11	2	0.056
12	1	0.028

Simulace házení dvěma kostkami 1/2

```
1  import random
3  h=[0]*13 # četnost výskytu součtu h[s]
4  n=100000
6  # Simulace n dvojic hodů
7  for i in range(n):
8    x=random.randrange(1,7)
9    y=random.randrange(1,7)
10   s=x+y
11   h[s]+=1
```

Simulace házení dvěma kostkami 2/2

```
for s in range(2,13): # Tisk pravděpodobností
  anal=(6-abs(s-7))/36
  simul=h[s]/n
  print("s=%2d  P(s)=analyticky %0.3f    "
        "simulace %0.3f chyba %6.3f" % (s,anal,simul,anal-simul))
```

```
s= 2  P(s)=analyticky 0.028  simulace 0.028  chyba -0.000
s= 3  P(s)=analyticky 0.056  simulace 0.056  chyba -0.000
s= 4  P(s)=analyticky 0.083  simulace 0.083  chyba  0.001
s= 5  P(s)=analyticky 0.111  simulace 0.113  chyba -0.001
s= 6  P(s)=analyticky 0.139  simulace 0.137  chyba  0.002
s= 7  P(s)=analyticky 0.167  simulace 0.168  chyba -0.001
s= 8  P(s)=analyticky 0.139  simulace 0.138  chyba  0.001
s= 9  P(s)=analyticky 0.111  simulace 0.113  chyba -0.002
s=10  P(s)=analyticky 0.083  simulace 0.083  chyba  0.001
s=11  P(s)=analyticky 0.056  simulace 0.054  chyba  0.001
s=12  P(s)=analyticky 0.028  simulace 0.028  chyba -0.000
```

I. Složené datové typy

Hodnotová semantika

- U objektu je důležitá hodnota, nikoliv identita. Proměnná reprezentuje hodnotu.
- Primitivní typy v Pythonu se chovají jako hodnoty *(values)*
- Přiřazení vytvoří nový objekt.

```
1  a=7
2  b=a
3  a=6
5  print(a)
```

```
6
```

```
1  print(b)
```

```
7
```

Referenční semantika

- Proměnná typu pole (seznam) je referencí/odkazem *(reference,link)*
- Přiřazení vytvoří nový *odkaz* na existující objekt.

```
1  a=[7,3]
2  b=a
3  a[1]=6
4  print(a)
```

```
[7, 6]
```

```
1  print(b)
```

```
[7, 6]
```

- Pole lze měnit *(mutable)*
- Sdílení odkazů *(sharing,aliasing)*

Neměnnost

- Vlastnost datového typu.
- Neměnné objekty po vytvoření změnit nelze — řetězce, *n*-tice

```
1 s="Ahoj"; s[0]="a"
```

```
TypeError: 'str' object does not support item assignment
```

- Neměnné typy (řetězce, *n*-tice) se také chovají jako hodnoty

```
1 r=s
2 r="Nazdar"
3 print(a)
```

```
Nazdar
```

```
1 print(b)
```

```
Ahoj
```

Proměnné v Pythonu nejsou hodnoty, ale **odkazy (references)**.

Práce s neměnnými objekty

- Jak lze s neměnnými objekty pracovat?
- Vytvoříme objekt nový, nezávislý na starém.

```
s="Ahoj"
r="a"+s[1:]
```

```
>>> r
'ahoj'
>>> s
'Ahoj'
```

Vedlejší efekty funkcí

- Funkce může změnit své změnitelné *(mutable)* parametry

```
1  def add_one(x):
2      for i in range(len(x)):
3          x[i]+=1
5  v=[1,2,3]
6  add_one(v)
7  print(v)
```

```
[2, 3, 4]
```

- Funkce není čistá.
- Vedlejším efektům se pokud možno vyhněte.

Nahrazení vedlejších efektů

```
1  def add_one_clean(x):
2      return [x[i]+1 for i in range(len(x))]
4  v=[1,2,3]
5  v=add_one_clean(v)
6  print(v)
```

```
[1, 2, 2]
```

Kopírování polí

- Přiřazení proměnné typu pole vytvoří nový odkaz na stejné pole

```
1 a=[7,3]
2 b=a
3 a[1]=6
4 print("a = ", a, "b = ", b)
```

```
a = [7, 6] b = [7, 6]
```

- Kopírováním se vytvoří nový objekt se stejným obsahem

```
1 a=[7,3]
2 b=a.copy() # Lze psát též b=a[:]
3 a[1]=6
4 print("a = ",a, "b = ",b)
```

```
a = [7, 6] b = [7, 3]
```

Kopírování je mělké, možná jen jedna úroveň.

Neměnost

Výhody

- Vyloučení vedlejších efektů
- Méně chyb
  - Kdy kopírovat, co lze přepsat
  - Vzdálený kód mění proměnné
- Snazší optimalizace a paralelizace

Nevýhody

- Trochu menší expresivita.
- Objektů vzniká velké množství, alokace/dealokace paměti.
- Nutnost kopírování – paměťová a výpočetní náročnost.

Existují techniky jak kopírování omezit.

I. Složené datové typy

Textové řetězce

Pole

Hodnoty a reference

Dourozměrné pole

Dvourozměrné pole – matice

- Pole polí

```
a=[[1,0,2,3],[0,2,3,1],[3,0,2,5]]
print(a)
```

```
[[1, 0, 2, 3], [0, 2, 3, 1], [3, 0, 2, 5]]
```

```
print(len(a))
```

```
3
```

```
print(a[1])
```

```
[0, 2, 3, 1]
```

```
print(a[1][2])
```

```
3
```

Tisk matice

```
1  def print_2d_matrix(a):
2      for i in range(len(a)):
3          print(a[i])
5  print_2d_matrix(a)
```

```
[1, 0, 2, 3]
[0, 2, 3, 1]
[3, 0, 2, 5]
```

Část II

Funkce a proměnné

II. Funkce a proměnné

Globální a lokální proměnné

Globální proměnné

- definovány globálně (tj. ne uvnitř funkce)
- jsou viditelné kdekoli v programu

Lokální proměnné

- definovány uvnitř funkce
- jsou viditelné jen ve své funkci

Rozsah proměnných obecněji

- proměnné jsou viditelné v rámci svého **rozsahu**
- rozsahem může být
  - funkce
  - moduly (soubory se zdrojovým kódem)
  - třídy
  - a jiné (záleží na konkrétním jazyce)
- terminologie: `namespace`, `scope`, ...

Příklad – globální a lokální proměnná

```
1  a = "This is global."
3  def example1():
4    b = "This is local."
5    print(a)
6    print(b)
8  example1() # This is global.
9             # This is local.
11 print(a) # This is global.
12 print(b) # ERROR!
14 # NameError:  name 'b' is not defined
```

Příklad – zastínění globální proměnné

```
1  a = "Think global."
3  def example2():
4    a = "Act local."
5    print(a)
7  print(a) # Think global.
8  example2() # Act local.
9  print(a) # Think global.
```

Příklad – změna globální proměnné

```
1  a = "Think global."
3  def example3():
4    global a
5    a = "Act local."
6    print(a)
8  print(a) # Think global.
9  example3() # Act local.
10 print(a) # Act local.
```

Lokální proměnné – deklarace

- lokální proměnná vzniká, pokud je přiřazení kdekoliv uvnitř těla funkce

```
1  a = "Think global."
3  def example4(change_opinion=False):
4    print(a)
5    if change_opinion:
6      a = "Act local."
7      print("Changed opinion:", a)
9  print(a) # Think global.
10 example4() # ERROR!
```

Rozsah proměnných – cyklus `for`

- Rozsah proměnné v Pythonu není pro dílčí blok kódu, ale pro celou funkci (resp. globální kód)
- Častá chyba: řídicí proměnná `for` cyklu použita po ukončení cyklu

```
1  n = 9
2
3  for i in range(n):
4    print(i)
6  if i % 2 == 0:
7    print("I like even length lists")
```

Slovník proměnných

- proměnné jsou uloženy ve slovniku
- výpis: `globals()`, `locals()`

```
1  def function():
2    x = 100
3    s = "dog"
5  print(locals())
7  a = [1, 2, 3]
8  x = 200
9    function()
10
11 print(globals())
```

Závěry

Doporučení

- spíše se vyhýbat globálním proměnným
- omezit na specifické případy, např. globální konstanty

Proč?

- horší čitelnost kódu
- náročnější testování, možný zdroj chyb

Obecně: lokalita kódu je užitečná

Alternativy

- předávání parametrů funkcím a využití návratových hodnot
- objekty

II. Funkce a proměnné

Globální a lokální proměnné

Funkce

Ošetření chyb

Funkce a vedlejší efekty

Čistá funkce

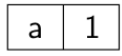
- funkce bez vedlejších efektů
- výstup závisí jen na vstupních parametrech

Vedlejší efekty

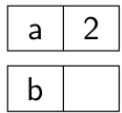
- změna měnitelných parametrů
  - OK, ale nemíchat s návratovou hodnotou, vhodně pojmenovat, dokumentovat
- změna globálních proměnných (které nejsou parametry)
  - většinou cesta do pekla
- změna stavu systému (libovolné výpisy, zápis do souboru, databáze, ...)
  - nutnost, ale nemíchat chaoticky s výpočty

Proměnné a pamět

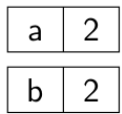
```
int a, b;
a = 1;
```




```
a = 2;
```




```
b = a;
```

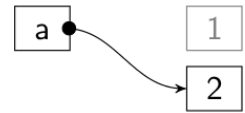


Jazyk C

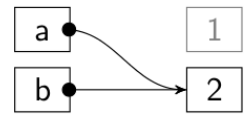
Proměnné jako hodnoty

```
a = 1;
```


```
a = 2;
```




```
b = a;
```



Jazyk Python

Proměnné jako odkazy

Identita a rovnost

- Funkce `id()` – vrací identitu objektu (adresa v paměti)

```
a = 1000
b = a
print(a, b)
print(id(a), id(b))
b += 1
print(a, b)
print(id(a), id(b))
```

```
a = [1]
b = a
print(a, b)
print(id(a), id(b))
b.append(2)
print(a, b)
print(id(a), id(b))
```

- Operátor `is` – stejná identita

```
a = [1, 2, 3]
b = [1, 2, 3]
print(a == b) # True
print(id(a) == id(b)) # False

```

Předávání parametrů funkcím 1/3

- hodnotou (call by value)
  - předá se hodnota proměnné (kopie)
  - standardní v C, C++, apod.
- odkazem (call by reference)
  - předá se odkaz na proměnnou
  - lze použít v C++
- jiné možnosti (jménem, hodnotou-výsledkem, ...)
- jazyk Python: něco mezi voláním hodnotou a odkazem
  - podobně funguje např. Java
  - někdy nazýváno `call by object sharing`

Předávání parametrů funkcím 2/3

Předávání parametrů hodnotou

- parametr je vlastně lokální proměnná
- funkce má svou vlastní lokální kopii předané hodnoty
- funkce nemůže měnit hodnotu předané proměnné

Předávání parametrů odkazem

- nepředává se hodnota, ale odkaz na proměnnou
- změny parametru jsou ve skutečnosti změny předané proměnné

Předávání parametrů funkcím 3/3

Předávání parametrů v Pythonu

- parametr drží odkaz na předanou proměnnou
- změna parametru změní i předanou proměnnou
- pro **neměnitelné typy** tedy v podstatě funguje jako předávání hodnotou
  - čísla, řetězce, n-tice (tuples)
- pro **měnitelné typy** jako předávání odkazem
  - pozor: přiřazení znamená změnu odkazu

Připomenutí

- neměnitelné typy: `int`, `str`, `tuple`, ...
- měnitelné typy: `list`, `dict`, ...

Předávání parametrů funkcím – příklad 1/4

- číselný parametr je neměnitelný, nestane se nic

```
1  def update_param_int(x):
2    x = x + 1
4  a = 1
5  print(a)
6  update_param_int(a)
7  print(a)
```

```
$ python update_param_int
1
1
```

Předávání parametrů funkcím – příklad 2/4

- seznam je měnitelný, změna se projeví i mimo funkci

```
1  def update_param_list(x):
2    x.append(3)
3
4  a = [1, 2]
5  print(a)
6  update_param_list(a)
7  print(a)
```

```
$ python update_param_list
[1, 2]
[1, 2, 3]
```

Předávání parametrů funkcím – příklad 3/4

- odkaz se změní na nový seznam, původní je nezměněn

```
1  def change_param_list(x):
2    x = [1, 2, 3]
3
4  a = [1, 2]
5  print(a)
6  change_param_list(a)
7  print(a)
```

```
$ python change_param_list
[1, 2]
[1, 2]
```

Předávání parametrů funkcím – příklad 4/4

- kvíz

```
1  def test(s):
2    s.append(3)
3    s = [42, 17]
4    s.append(9)
5    print(s)
6
7  t = [1, 2]
8  test(t)
9  print(t)
```

Práce s parametry

```
1  def change_list(alist, value):
2    alist.append(value)
4  def return_new_list(alist, value):
5    newlist = alist[:]
6    newlist.append(value)
7    return newlist
```

Práce s parametry

- operátor `+=` – různé chování pro neměnné typy a seznamy

```python
def increment(x):
  print(x, id(x))
  x += 1
  print(x, id(x))

p = 42
increment(p)
print(p, id(p))
```

lec03/increment.py

```
42 1906340608
43 1906340640
42 1906340608
```

```python
def add_to_list(s):
  print(s, id(s))
  s += [1]
  print(s, id(s))

t = [1, 2]
add_to_list(t)
print(t, id(t))
```

lec03/add_to_list.py

```
[1, 2] 2657138746184
[1, 2, 1] 2657138746184
[1, 2, 1] 2657138746184
```

Práce s parametry

- pozor na rozdíl mezi = a += u seznamů

```python
def add_to_list1(s):
  print(s, id(s))
  s += [1]
  print(s, id(s))

t = [1, 2]
add_to_list1(t)
print(t)
```

lec03/add_to_list1.py

```
[1, 2] 2319764595464
[1, 2, 1] 2319764595464
[1, 2, 1]
```

```python
def add_to_list2(s):
  print(s, id(s))
  s = s + [1]
  print(s, id(s))

t = [1, 2]
add_to_list2(t)
print(t)
```

lec03/add_to_list2.py

```
[1, 2] 1528528741192
[1, 2, 1] 1528528823752
[1, 2]
```

II. Funkce a proměnné

Blok `try` – `except`

- Ošetření se v Pythonu provádí pomocí bloku `try` – `except`

```
try:
  # blok prikazu
except jmeno_prvni_vyjimky:
  # blok prikazu
except jmeno_dalsi_vyjimky:
  # blok příkazů
# zde je bud konec, nebo zachyceni dalsich vyjimek
```

- Pokud chceme zachytit i chybovou zprávu, musíme napsat:

```
except jmeno_vyjimky as chyba:
  # text vyjimky se ulozi do promenne chyba
```

Výjimky

Základní výjimky v Pythonu

- **SyntaxError** – chyba ve zdrojovém kódu
- **ZeroDivisionError** – dělení nulou
- **TypeError** – nesprávné použití datových typů, např. sčítání řetězce a čísla apod.
- **ValueError** – nesprávná hodnota

Více o výjimkách: `https://docs.python.org/3/library/exceptions.html`

```
(x,y) = (5,0)
try:
  z = x/y
except ZeroDivisionError:
  print("divide by zero")
```

```
import sys
try:
  z = 5/0
except:
  print(sys.exc_info()[0])
```

Příklad

```
1  def nacti_cislo ():
2    spatne = True
3    while spatne:
4      try:
5        cislo = float(input("float: "))
6        spatne = False
7      except ValueError as err:
8        print(err)
9      else:
10       return cislo
12 print(nacti_cislo())
```

```
float: ahoj
could not convert string to float: 'ahoj'
float: 3.14
3.14
```

Závěr

- Pole
  - často používaná datová struktura
  - obsahuje n prvků (nejčastěji stejného typu)
  - k prvkům přistupujeme pomocí celočíselného indexu
  - prvky pole mohou být i složené datové typy
- Proměnné jsou reference, aliasing
- Neměnné (immutable) typy, hodnotová semantika
- Příklady použití polí