4. Složené datové typy – pole, textové řetězce

BAB37ZPR – Základy programování

Stanislav Vítek

Katedra radioelektroniky
Fakulta elektrotechnická
České vysoké učení v Praze

Přehled témat

- Část 1 – Složené datové typy
  - Pole
  - P4.1 Jména dnů v týdnu
  - P4.2 Průměr a směrodatná odchylka
  - P4.3 Náhodná permutace
  - P4.4 Házení dvěma kostkami
  - Textové řetězce
  - Hodnoty a reference
  - Dourozměrné pole
- Část 2 – Funkce a proměnné
  - Globální a lokální proměnné
  - Funkce

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

Pole

P4.1 Jména dnů v týdnu

P4.2 Průměr a směrodatná odchylka

P4.3 Náhodná permutace

P4.4 Házení dvěma kostkami

Textové řetězce

Hodnoty a reference

Dourozměrné pole

Pole

- Obsahuje `N` elementů (objektů, prvků), indexovaných od `0` – proč?
  - Why numbering should start at zero (Edsger W. Dijkstra)
  - Why do array indexes start with 0 (zero) in many programming languages?
- Přímý přístup *(random access)*
  - Pro čtení i zápis, v konstantním čase

```
>>> # Vytvoření
>>> a = [0.3,0.6,0.1]
>>> a
[0.3, 0.6, 0.1]
>>> type(a)
<class 'list'>
```

```
>>> # Čtení prvku
>>> a[1]
0.6
>>> a[0]
0.3
>>> # Změna prvku
>>> a[2] = 1.5
>>> a
[0.3, 0.6, 1.5]
```

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 (list) v Pythonu je významně flexibilnější
- Lze využít modul array

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

- 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

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

- Index může být výraz

```
>>> s=0.
>>> for i in range(3):
...     s+=a[i]
...     print("a[%d]=%f" % (i,a[i]))
...
a[0]=0.300000
a[1]=0.600000
a[2]=0.100000
>>> print(s)
0.9999999999999999
```

Pole různých typů

- Homogenní pole → všechny prvky jsou stejného typu.

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

- Nehomogenní pole

```
>>> a = [1, 3.14, "ahoj", [1, 2, 3]]
```

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)
0.9999999999999999
>>> 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]
>>> # co kdybychom chteli nasobit jednotlive prvky konstantou?
>>> [3*x for x in [111, 222, 333]]
[333, 666, 999]
```

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]
>>> [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[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

```
>>> 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

```
>>> a=[]
>>> for i in range(10):
...     a+=[0.0]
...
>>> 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

- *i*-tý prvek pole `x` – `x[i]`
- zároveň platí `x[i] = x[len(x)+i]`

Specialita Pythonu.

- pro sekvenční typy: pole, řetězce, *n*-tice

```
>>> a = [2, 7, 1]
>>> print("a[2]={}, a[-1]={}".format(a[2], a[-1]))
a[2]=1, a[-1]=1

>>> s = "Ferda"
>>> print(s[2])
r

>>> t = (1,2)
>>> print(t[0])
1
```

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:

```
>>> a=[6,7,5,2,9]
>>> print(a[2:4])
[5, 2]
>>> print(a[:3])
[6, 7, 5]
```

I. Složené datové typy

Pole

P4.1 Jména dnů v týdnu

P4.2 Průměr a směrodatná odchylka

P4.3 Náhodná permutace

P4.4 Házení dvěma kostkami

Textové řetězce

Hodnoty a reference

Dourozměrné pole

4.1 Jména dnů v týdnu

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

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

4.1 Jména dnů v týdnu – pomocí pole

```
>>> jmena_dni=["pondeli","utery","streda","ctvrtek",
...            "patek","sobota","nedele"]

>>> print(jmena_dni[3])
ctvrtek
```

Uzávorkovaný výraz lze rozdělit na více řádek.

```
>>> def jmeno_dne(i):
...     return jmena_dni[i]
...
>>> print(jmeno_dne(3))
ctvrtek
```

I. Složené datové typy

Pole

P4.1 Jména dnů v týdnu

P4.2 Průměr a směrodatná odchylka

P4.3 Náhodná permutace

P4.4 Házení dvěma kostkami

Textové řetězce

Hodnoty a reference

Dourozměrné pole

Odhad ze vzorků

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

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

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

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

```
>>> import math

>>> 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))
...
```

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

```
>>> a=list(range(1001))
>>> 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 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:

```
>>> def mean(v):
...   "Calculate a mean of a vector"
...   s=0.
...   for x in v:
...     s+=x
...   return(s/len(v))
...
>>> a=list(range(1001))
>>> print("mean=", mean(a))
mean= 500.0
```

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 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

- 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 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

```
>>> 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))
...
```

- lze dále zkrátit

```
>>> def stdev(v):
...     return math.sqrt(sum([(x-mean(v))**2 for x in v])/
... (len(v)-1))
...
>>> a=list(range(1001))
>>> print("mean=",mean(a)," sigma=",stdev(a))
mean= 500.0  sigma= 289.10811126635656
```

Výsledek se nezměnil. Zkracování ovšem může zhoršit srozumitelnost kódu

Použití funkcí pole 3/3

- Funkci

```
>>> 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))
...
```

- lze dále zkrátit

```
>>> def stdev(v):
...     return math.sqrt(sum([(x-mean(v))**2 for x in v])/
... (len(v)-1))
...
>>> a=list(range(1001))
>>> print("mean=",mean(a)," sigma=",stdev(a))
mean= 500.0  sigma= 289.10811126635656
```

Výsledek se nezměnil. Zkracování ovšem může zhoršit srozumitelnost kódu.

I. Složené datové typy

Pole

P4.1 Jména dnů v týdnu

P4.2 Průměr a směrodatná odchylka

P4.3 Náhodná permutace

P4.4 Házení dvěma kostkami

Textové řetězce

Hodnoty a reference

Dourozměrné pole

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 $j = i, i+1, \ldots, N-1$

- Využijeme generátor náhodných čísel z modulu `random`

```
>>> import random

>>> def permutation(n):
...     "Create a random permutation of integers 0..n-1"
...     p=list(range(n))
...     for i in range(n-1):
...         r=random.randrange(i,n)
...         temp=p[r]
...         p[r]=p[i]
...         p[i]=temp
...     return(p)
...
```

- Vyzkoušíme:

```
>>> print(permutation(10))
[3, 9, 1, 8, 4, 6, 7, 2, 0, 5]
>>> print(permutation(10))
[3, 9, 0, 6, 5, 2, 1, 7, 4, 8]
>>> print(permutation(10))
[4, 6, 2, 1, 7, 9, 8, 0, 5, 3]
```

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

```
>>> import random

>>> def permutation(n):
...     "Create a random permutation of integers 0..n-1"
...     p=list(range(n))
...     print("p=",p)
...     for i in range(n-1):
...         r=random.randrange(i,n)
...         temp=p[r]
...         p[r]=p[i]
...         p[i]=temp
...         print("i=%d r=%d p=%s" % (i,r,str(p)))
...     return(p)
...
```

```
>>> permutation(5)
p= [0, 1, 2, 3, 4]
i=0 r=3 p=[3, 1, 2, 0, 4]
i=1 r=1 p=[3, 1, 2, 0, 4]
i=2 r=4 p=[3, 1, 4, 0, 2]
i=3 r=4 p=[3, 1, 4, 2, 0]
[3, 1, 4, 2, 0]
>>> permutation(5)
p= [0, 1, 2, 3, 4]
i=0 r=0 p=[0, 1, 2, 3, 4]
i=1 r=3 p=[0, 3, 2, 1, 4]
i=2 r=4 p=[0, 3, 4, 1, 2]
i=3 r=4 p=[0, 3, 4, 2, 1]
[0, 3, 4, 2, 1]
```

4.3 Permutace pole

- Vytiskneme prvky pole v náhodném pořadí:

```
>>> barvy=["srdce","listy","kule","zaludy"]
>>> p=permutation(len(barvy))
p= [0, 1, 2, 3]
i=0 r=3 p=[3, 1, 2, 0]
i=1 r=1 p=[3, 1, 2, 0]
i=2 r=3 p=[3, 1, 0, 2]

>>> for i in range(len(barvy)):
...     print(barvy[p[i]], end=" ")
...
zaludy listy srdce kule
```

- Pole v novém pořadí:

```
>>> print([barvy[i] for i in p])
['zaludy', 'listy', 'srdce', 'kule']
```

I. Složené datové typy

Pole

P4.1 Jména dnů v týdnu

P4.2 Průměr a směrodatná odchylka

P4.3 Náhodná permutace

P4.4 Házení dvěma kostkami

Textové řetězce

Hodnoty a reference

Dourozměrné pole

P4.4 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

```
>>> import random

>>> h=[0]*13 # četnost výskytu součtu h[s]
>>> n=100000

>>> # Simulace n dvojic hodů
>>> for i in range(n):
...   x=random.randrange(1,7)
...   y=random.randrange(1,7)
...   s=x+y
...   h[s]+=1
...
>>> 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 0.057 chyba -0.001
s= 4  P(s)=analyticky 0.083    simulace 0.085 chyba -0.002
s= 5  P(s)=analyticky 0.111    simulace 0.109 chyba  0.003
s= 6  P(s)=analyticky 0.139    simulace 0.140 chyba -0.001
s= 7  P(s)=analyticky 0.167    simulace 0.167 chyba  0.000
s= 8  P(s)=analyticky 0.139    simulace 0.140 chyba -0.001
s= 9  P(s)=analyticky 0.111    simulace 0.110 chyba  0.001
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 0.029 chyba -0.001
```

I. Složené datové typy

Pole

P4.1 Jména dnů v týdnu

P4.2 Průměr a směrodatná odchylka

P4.3 Náhodná permutace

P4.4 Házení dvěma kostkami

Textové řetězce

Hodnoty a reference

Dourozměrné pole

Řetězce a znaky – ukázky operací

```
>>> "kos" * 3
'koskoskos'
>>> "petr" + "klic"
'petrklic'
>>> text = "velbloud"
>>> len(text)
8
>>> text[2]
'l'
>>> text[-1]
'd'
>>> ord('b')
98
>>> chr(99)
'c'
```

Základní pravidla

Uvozovky, apostrofy

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

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), end=' ')
...
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

Další vlastnosti – řezy

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

Další vlastnosti – neměnnost

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

```
>>> text = "kopec"
>>> text[2] = "n"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> text = text[:2] + "n" + text[3:]
```

Formátovací funkce

```
>>> text = "i Have a dream."
>>> print(text.upper())
I HAVE A DREAM.
>>> print(text.lower())
i have a dream.
>>> print(text.capitalize())
I have a dream.
>>> print(text.rjust(30))
                i Have a dream.
>>> print("X", text.center(30), "X")
X         i Have a dream.          X
>>> print(text.replace("dream", "nightmare"))
i Have a nightmare.
```

... a mnoho dalších, více v dokumentaci Pythonu

I. Složené datové typy

Pole

P4.1 Jména dnů v týdnu

P4.2 Průměr a směrodatná odchylka

P4.3 Náhodná permutace

P4.4 Házení dvěma kostkami

Textové řetězce

Hodnoty a reference

Dourozměrné pole

Hodnotová sémantika

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

```
>>> a=7
>>> b=a
>>> a=6

>>> print(a)
6
>>> print(b)
7
```

Referenční semantika

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

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

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

Neměnnost

- Vlastnost datového typu, neměnné objekty po vytvoření nelze změnit (řetězce, *n*-tice)

```
>>> s="Ahoj"
>>> s[0]="a"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

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

```
>>> r=s
>>> r="Nazdar"
>>> print(a)
[7, 6]
>>> print(b)
[7, 6]
```

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

```
>>> def add_one(x):
...     for i in range(len(x)):
...         x[i]+=1
...
>>> v=[1,2,3]
>>> add_one(v)
>>> print(v)
[2, 3, 4]
```

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

Nahrazení vedlejších efektů

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

Kopírování polí

- Přiřazení proměnné typu pole vytvoří nový odkaz na stejné pole

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

- Kopírováním se vytvoří nový objekt se stejným obsahem

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

Pole

P4.1 Jména dnů v týdnu

P4.2 Průměr a směrodatná odchylka

P4.3 Náhodná permutace

P4.4 Házení dvěma kostkami

Textové řetězce

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

```
>>> def print_2d_matrix(a):
...     for i in range(len(a)):
...         print(a[i])
...
>>> 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

Funkce

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`, ...

Globální a lokální proměnná

```
>>> del a; del b
>>> a = "je globální"
>>> def example1():
...     b = "je lokální"
...     print("a", a, ", b", b)
...
>>> example1()
a je globální , b je lokální
>>> print("a", a)
a je globální
>>> print("b", b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'b' is not defined
```

Zastínění globální proměnné

```
>>> del a; del b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'b' is not defined
>>> a = "je globální"
>>> def example2():
...     a = "je lokální"
...     print("a", a)
...
>>> print("a", a)
a je globální
>>> example2()
a je lokální
>>> print("a", a)
a je globální
```

Změna globální proměnné

```
>>> del a
>>> a = "je globální"

>>> def example3():
...     global a
...     a = "je lokální"
...     print("a", a)
...
>>> print("a", a)
a je globální
>>> example3()
a je lokální
>>> print("a", a)
a je lokální
```

Lokální proměnné – deklarace

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

```
>>> a = "je globální"
>>> def example4(zmena=False):
...     print(a)
...     if zmena:
...         a = "je lokální"
...         print("zmena: a", a)
...
>>> print("a", a)
a je globální
>>> example4()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in example4
UnboundLocalError: local variable 'a' referenced before assignment
```

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

```
>>> n = 9
>>> for i in range(n):
...     print(i, end=' ')
...
0 1 2 3 4 5 6 7 8
>>> if i % 2 == 0:
...     print("I like even length lists", i)
...
I like even length lists 8
```

Slovník proměnných

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

```
>>> x = [30, 50]
>>> s = 10

>>> def function():
...     global x
...     s = "dog"
...     print(locals())
...
>>> function()
{'s': 'dog'}
```

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

Funkce a vedlejší efekty

Čistá funkce

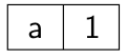
- funkce bez vedlejších efektů
- výstup závisí jen na vstupních parametrech

Vedlejší efekty

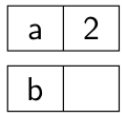
- změna měnitelných parametrů
  - OK, ale nemíchat s návratovou hodnotou, vhodně pojmenovat, dokumentovat
- změna globálních proměnných (které nejsou parametry)
  - většinou cesta do pekla
- změna stavu systému (libovolné výpisy, zápis do souboru, databáze, ...)
  - nutnost, ale nemíchat chaoticky s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

Proměnné a pamět

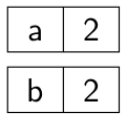
```
int a, b;
a = 1;
```




```
a = 2;
```




```
b = a;
```

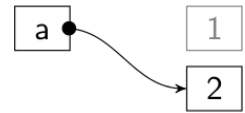


Jazyk C

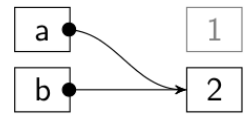
Proměnné jako hodnoty

```
a = 1;
```


```
a = 2;
```




```
b = a;
```



Jazyk Python

Proměnné jako odkazy

Identita a rovnost 1/2

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

```
>>> a = 1000
>>> b = a
>>> print(a, b)
1000 1000
>>> print(id(a), id(b))
2579831275984 2579831275984
>>> b += 1
>>> print(a, b)
1000 1001
>>> print(id(a), id(b))
2579831275984 2579831275792
```

```
>>> a = [1]
>>> b = a
>>> print(a, b)
[1] [1]
>>> print(id(a), id(b))
2579831576576 2579831576576
>>> b.append(2)
>>> print(a, b)
[1, 2] [1, 2]
>>> print(id(a), id(b))
2579831576576 2579831576576
```

Identita a rovnost 2/2

- Operátor `is` – stejná identita

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a == b
True
>>> id(a) == id(b)
False
>>> a is 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

```
>>> def update_param_int(x):
...     x = x + 1
...
>>> a = 1
>>> print(a)
1
>>> update_param_int(a)
>>> print(a)
1
```

Předávání parametrů funkcím – příklad 2/4

- seznam je měnitelný, změna se projeví i mimo funkci

```
>>> def update_param_list(x):
...     x.append(3)
...
>>> a = [1, 2]
>>> print(a)
[1, 2]
>>> update_param_list(a)
>>> print(a)
[1, 2, 3]
```

Předávání parametrů funkcím – příklad 3/4

- odkaz se změní na nový seznam, původní je nezměněn

```
>>> def change_param_list(x):
...     x = [1, 2, 3]
...
>>> a = [1, 2]
>>> print(a)
[1, 2]
>>> change_param_list(a)
>>> print(a)
[1, 2]
```

Předávání parametrů funkcím – příklad 4/4

- kvíz

```
>>> def test(s):
...     s.append(3)
...     s = [42, 17]
...     s.append(9)
...     print(s)
...
>>> t = [1, 2]
>>> test(t)
[42, 17, 9]
>>> print(t)
[1, 2, 3]
```

Práce s parametry

```
>>> def change_list(alist, value):
...     alist.append(value)
...
>>> def return_new_list(alist, value):
...     newlist = alist[:]
...     newlist.append(value)
...     return newlist
...
```

Práce s parametry

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

```
>>> def increment(x):
...     print(x, id(x))
...     x += 1
...     print(x, id(x))
...
>>> p = 42
>>> increment(p)
42 2579817170448
43 2579817170480
>>> print(p, id(p))
42 2579817170448
```

lec04/increment.py

```
>>> def add_to_list(s):
...     print(s, id(s))
...     s += [1]
...     print(s, id(s))
...
>>> t = [1, 2]
>>> add_to_list(t)
[1, 2] 2579831586688
[1, 2, 1] 2579831586688
>>> print(t, id(t))
[1, 2, 1] 2579831586688
```

lec04/add_to_list.py

Práce s parametry

- pozor na rozdíl mezi = a += u seznamů

```
>>> def add_to_list1(s):
...     print(s, id(s))
...     s += [1]
...     print(s, id(s))
...
>>> t = [1, 2]
>>> add_to_list1(t)
[1, 2] 2579831259840
[1, 2, 1] 2579831259840
>>> print(t)
[1, 2, 1]
```

lec04/add_to_list1.py

```
>>> def add_to_list2(s):
...     print(s, id(s))
...     s = s + [1]
...     print(s, id(s))
...
>>> t = [1, 2]
>>> add_to_list2(t)
[1, 2] 2579831586688
[1, 2, 1] 2579831667968
>>> print(t)
[1, 2]
```

lec04/add_to_list2.py

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