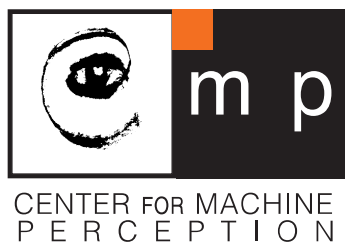
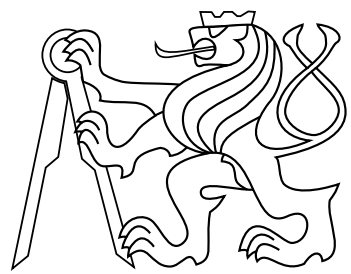
Pole / Arrays

Jan Kybic

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

2016





Datové typy
v Pythonu

- **Jednoduché typy:** *(primitive data type)*
  - celé číslo, reálné číslo, logická hodnota,
- **Složené typy / datové struktury:** *(composite/compound data type, data structures)*
  - řetězec, *n*-tice *(tuple)*, **pole**

Složené typy / datové struktury

- Hierarchicky sdružují data
- Slouží k organizaci dat, související data jsou uložena a manipulována spolu
- Pro zvýšení efektivity programování i vykonávání
- Operace na datových strukturách
  - Vytvoření
  - Čtení jednotlivých složek (elementů)
  - Modifikace jednotlivých složek
  - Vyhledávání
  - Přidávání, odebírání, ...

Pole
(Array)

- Obsahuje N elementů (objektů,prvků)
- Přímý přístup *(random access)*
  - Pro čtení i zápis
  - V konstantním čase

Pole v Pythonu

- **Vytvoření**

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

- **Čtení prvku**

```
>>> a[1]
0.6
>>> a[0]
0.3
```

V Pythonu má první prvek index 0.

Pole v Pythonu

- **Vytvoření**

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

- **Čtení prvku**

```
>>> a[1]
0.6
>>> a[0]
0.3
```

- **Změna prvku**

```
>>> a[2]=1.5
>>> a
[0.3, 0.6, 1.5]
```

V Pythonu má první prvek index 0.

Pole bez polí

- **Vytvoření**

```
>>> a0=0.3 ; a1=0.6 ; a2=0.1
```

- **Čtení prvku**

```
>>> a1
0.6
>>> a0
0.3
```

- **Změna prvku**

```
>>> a2=1.5
```

S polem je to pohodlnější...

- Operace s celým polem

```
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]))
print(s)
```

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

Pole různých typů

```
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","žaludy"]
print(barvy[3])
```

```
žaludy
```

```
bits=[True,False]
print(bits)
```

```
[True, False]
```

Homogenní pole = všechny prvky jsou stejného typu.

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
```

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

- Výčtem

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

- opakováním prvků

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

☝ Opakování používejte pouze pro primitivní nebo neměnné typy.

Vytvoření pole

- Výčtem

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

- opakováním prvků

  ```
  >>> a=10*[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)

- z posloupnosti

```
>>> list(range(1,11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

V Pythonu se tento druh pole jmenuje `list`

Vytvoření pole (2)

- z posloupnosti

```
>>> list(range(1,11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

ℹ 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.

ℹ V jiných jazycích je délka pole pevná.

Vytvoření pole (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 *x*
- pro sekvenční typy: pole, řetězce, *n*-tice

Indexace

`x[i]`

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

```
a=[2,7,1]
print(a[2])
```

Indexace

`x[i]`

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

```
a=[2,7,1]
print(a[2])
```

```
1
```

Indexace

`x[i]`

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

```
a=[2,7,1]
print(a[2])
```

```
1
```

```
s="Ferda"
print(s[2])
```

Indexace

`x[i]`

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

```
a=[2,7,1]
print(a[2])
```

```
1
```

```
s="Ferda"
print(s[2])
```

```
r
```

Indexace

`x[i]`

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

```
a=[2,7,1]
print(a[2])
```

```
1
```

```
s="Ferda"
print(s[2])
```

```
r
```

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

```
1
```

Indexace (2)

Záporné indexy

Specialita Pythonu — nemusíte si pamatovat.

`x[i]` pro $i < 0$

Indexace (2)

Záporné indexy

Specialita Pythonu — nemusíte si pamatovat.

`x[i]` pro $i < 0$

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

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

Řezy pole
(array slicing)

Specialita Pythonu — nemusíte si pamatovat.

$$\texttt{x[i:j]} = \left[\texttt{x[i]}, \texttt{x[i+1]}, \ldots, \texttt{x[j-1]}\right]$$

Příklad:

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

Řezy pole
(array slicing)

Specialita Pythonu — nemusíte si pamatovat.

$$\texttt{x[i:j]} = \left[\texttt{x[i]}, \texttt{x[i+1]}, \ldots, \texttt{x[j-1]}\right]$$

Příklad:

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

```
[5, 2]
```

Řezy pole (2)
(vynechané argumenty)

```
x[i:]=x[i:len(x)]
   x[:j]=x[0:j]
x[:]=x[0:len(x)]=x
```

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

Řezy pole (2)
(vynechané argumenty)

```
x[i:]=x[i:len(x)]
   x[:j]=x[0:j]
x[:]=x[0:len(x)]=x
```

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

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

Řezy pole (2)
(vynechané argumenty)

```
x[i:]=x[i:len(x)]
   x[:j]=x[0:j]
x[:]=x[0:len(x)]=x
```

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

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

```
print(a[-2:])
```

Řezy pole (2)
(vynechané argumenty)

```
x[i:]=x[i:len(x)]
   x[:j]=x[0:j]
x[:]=x[0:len(x)]=x
```

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

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

```
print(a[-2:])
```

```
[2, 9]
```

Řezy pole (2)
(vynechané argumenty)

```
x[i:]=x[i:len(x)]
   x[:j]=x[0:j]
x[:]=x[0:len(x)]=x
```

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

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

```
print(a[-2:])
```

```
[2, 9]
```

```
s="Rakousko"
print(s[3:])
```

```
ousko
```

Příklad: Jména dnů v týdnu

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

```
def jmeno_dne(i):
  if i==0:
      return "pondělí"
  elif i==1:
      return "úterý"
  elif i==2:
      return "středa"
  elif i==3:
      return "čtvrtek"
  elif i==4:
      return "pátek"
  elif i==5:
      return "sobota"
  elif i==6:
      return "neděle"
  else:
      return "???"
```

Příklad: Jména dnů v týdnu (2)

```
print(jmeno_dne(3))
```

```
čtvrtek
```

Příklad: Jména dnů v týdnu (3)

Řešení pomocí pole

```
jmena_dni=["pondělí","úterý","středa","čtvrtek",
           "pátek","sobota","neděle"]

print(jmena_dni[3])
```

Příklad: Jména dnů v týdnu (3)

Řešení pomocí pole

```
jmena_dni=["pondělí","úterý","středa","čtvrtek",
           "pátek","sobota","neděle"]

print(jmena_dni[3])
```

```
čtvrtek
```

Příklad: Jména dnů v týdnu (3)

Řešení pomocí pole

```
jmena_dni=["pondělí","úterý","středa","čtvrtek",
           "pátek","sobota","neděle"]

print(jmena_dni[3])
```

```
čtvrtek
```

```
def jmeno_dne(i):
    return jmena_dni[i]

print(jmeno_dne(3))
```

Příklad: Jména dnů v týdnu (3)

Řešení pomocí pole

```
jmena_dni=["pondělí","úterý","středa","čtvrtek",
           "pátek","sobota","neděle"]

print(jmena_dni[3])
```

```
čtvrtek
```

```
def jmeno_dne(i):
    return jmena_dni[i]

print(jmeno_dne(3))
```

```
čtvrtek
```

Příklad: Jména dnů v týdnu (3)

Řešení pomocí pole

```
jmena_dni=["pondělí","úterý","středa","čtvrtek",
           "pátek","sobota","neděle"]

print(jmena_dni[3])
```

```
čtvrtek
```

```
def jmeno_dne(i):
    return jmena_dni[i]

print(jmeno_dne(3))
```

```
čtvrtek
```

Uzávorkovaný výraz lze rozdělit na více řádek.

Příklad: průměr a směrodatná odchylka

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))
```

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

Příklad: průměr a směrodatná odchylka (2)

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

```
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))
```

Příklad: průměr a směrodatná odchylka (3)

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

Průměr podruhé

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))
```

Argumentem cyklu `for` je *sekvence*, například pole.

Pole jako argument cyklu (2)

Výsledek je stejný

```
a=list(range(1001))
print("mean=",mean(a))
```

```
mean= 500.0
```

Použití funkcí pole

Průměr potřetí

Místo:

```
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))
```

☝ Pokud můžete, používejte existující funkce.

Použití funkcí pole (2)

Směrodatná odchylka podruhé

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)

```
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))
```

Použití funkcí pole (2)

```
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))
```

Výsledek se nezměnil

```
print("mean=",mean(a)," sigma=",stdev(a))
mean= 500.0  sigma= 289.10811126635656
```

☝ Zkracování může zhoršit srozumitelnost kódu.

Příklad: Náhodná permutace

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

Příklad: Náhodná permutace

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

Myšlenka

- 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)

```
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)
```

Příklad: Náhodná permutace (3)

Vyzkoušíme:

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

Kontrolní tisky

Jak to vlastně funguje

```
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)
```

Kontrolní tisky (2)

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

☝ Toto je velmi obecná a užitečná technika ověření funkčnosti programů.

Permutace pole

Vytiskneme prvky pole v náhodném pořadí:

```
barvy=["srdce","listy","kule","žaludy"]
p=permutation(len(barvy))

for i in range(len(barvy)):
    print(barvy[p[i]])
```

Permutace pole

Vytiskneme prvky pole v náhodném pořadí:

```
barvy=["srdce","listy","kule","žaludy"]
p=permutation(len(barvy))

for i in range(len(barvy)):
    print(barvy[p[i]])
```

```
žaludy
kule
listy
srdce
```

Permutace pole

Vytiskneme prvky pole v náhodném pořadí:

```
barvy=["srdce","listy","kule","žaludy"]
p=permutation(len(barvy))

for i in range(len(barvy)):
    print(barvy[p[i]])
```

```
žaludy
kule
listy
srdce
```

Pole v novém pořadí:

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

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

Příklad: Házení dvěma kostkami

Jaká je pravděpodobnost, že padne součet s?

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 možností	$P(s)$
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

```
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
```

Soubor `dve_kostky.py`

Simulace házení dvěma kostkami (2)

```
# Tisk výsledných pravděpodobností
for s in range(2,13):
    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))
```

Soubor `dve_kostky.py`

Simulace házení dvěma kostkami (2)

```
Terminal> python3 dve_kostky.py
```

```
s= 2  P(s)=analyticky 0.028    simulace 0.028    chyba -0.001
s= 3  P(s)=analyticky 0.056    simulace 0.056    chyba -0.000
s= 4  P(s)=analyticky 0.083    simulace 0.083    chyba  0.000
s= 5  P(s)=analyticky 0.111    simulace 0.112    chyba -0.001
s= 6  P(s)=analyticky 0.139    simulace 0.139    chyba -0.001
s= 7  P(s)=analyticky 0.167    simulace 0.166    chyba  0.000
s= 8  P(s)=analyticky 0.139    simulace 0.139    chyba -0.001
s= 9  P(s)=analyticky 0.111    simulace 0.112    chyba -0.000
s=10  P(s)=analyticky 0.083    simulace 0.081    chyba  0.003
s=11  P(s)=analyticky 0.056    simulace 0.056    chyba -0.000
s=12  P(s)=analyticky 0.028    simulace 0.028    chyba -0.000
```

Pole

Hodnoty a reference

Další příklady

Hodnotová semantika
(value semantics)

- 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)
```

Hodnotová semantika
(value semantics)

- 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
```

Hodnotová semantika
(value semantics)

- 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)
```

Hodnotová semantika
(value semantics)

- 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
(reference semantics)

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

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

Referenční semantika
(reference semantics)

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

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

Referenční semantika
(reference semantics)

- Proměnná typu pole 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)
```

Referenční semantika
(reference semantics)

- Proměnná typu pole 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
(immutability)

- Vlastnost datového typu.
- Neměnné objekty po vytvoření změnit nelze — 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
```

Neměnnost
(immutability)

- Vlastnost datového typu.
- Neměnné objekty po vytvoření změnit nelze — 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

```
a="raz"
b=a
a="dva"
print(a)
dva
```

Neměnnost
(immutability)

- Vlastnost datového typu.
- Neměnné objekty po vytvoření změnit nelze — 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

```
a="raz"
b=a
a="dva"
print(a)
```

```
dva
```

```
print(b)
```

Neměnnost
(immutability)

- Vlastnost datového typu.
- Neměnné objekty po vytvoření změnit nelze — 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

```
a="raz"
b=a
a="dva"
print(a)
dva
print(b)
raz
```

Neměnnost
(immutability)

- Vlastnost datového typu.
- Neměnné objekty po vytvoření změnit nelze — 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

```
a="raz"
b=a
a="dva"
print(a)
dva
print(b)
raz
```

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

Práce s neměnnými objekty

Jak lze s neměnnými objekty pracovat?

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)
```

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]
```

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 *(impure)*.

☝ 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]
```

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

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

Lze psát též `b=a[:]`

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

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

Lze psát též `b=a[:]`

Kopírování je mělké *(shallow)*, jen jedna úroveň.

Výhody

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

Neměnost
Immutability

Výhody

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

Nevýhody

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

Neměnost
Immutability

Výhody

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

Nevýhody

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

Existují techniky jak kopírování omezit.

Pole

Hodnoty a reference

Další příklady

Příklad: Problém sběratele

Coupon collector

Kolikrát musíme náhodně s opakováním vybrat z množiny N prvků, než vybereme každý prvek alespoň jednou?

Příklad: Problém sběratele

Coupon collector

Kolikrát musíme náhodně s opakováním vybrat z množiny N prvků, než vybereme každý prvek alespoň jednou?

Matematická analýza

$$\text{počet výběrů} = T(N) = \lceil N\, H_N \rceil \approx N(\log N + 0.577) + 0.5$$
$$T(50) = 225$$

coupon_collector.py

```
# -*- coding: utf-8 -*-
# Simulace problému sběratele,
import random
import sys

n=int(sys.argv[1])      # počet různých pruků
collectedCount=0        # počet již vybraných různých pruků
isCollected=[False]*n # již jsme viděli tento prvek
count=0                 # kolik pruků jsme již vybrali

while collectedCount < n:
  r=random.randrange(n) # náhodný prvek 0..n-1
  count+=1
  if not isCollected[r]: # nový prvek
    collectedCount+=1
    isCollected[r]=True

print("Pro n=%d bylo potřeba %d výběrů." % (n,count))
```

```
Terminal> python3 coupon_collector.py 50

Pro n=50 bylo potřeba 158 výběrů.

Terminal> python3 coupon_collector.py 50

Pro n=50 bylo potřeba 156 výběrů.

Terminal> python3 coupon_collector.py 50

Pro n=50 bylo potřeba 258 výběrů.

Terminal> python3 coupon_collector.py 50

Pro n=50 bylo potřeba 196 výběrů.
```

Prvočísla

prvocislo3.py

```
# prvocislo3.py - Vypíše prvočísla menší než zadaný limit
import sys
m=int(sys.argv[1])
for n in range(2,m): # cyklus 2..m-1
  p=2 # začátek testu
  while p*p<=n:
    if n % p == 0:
        break
    p+=1
  if p*p > n: # n je prvočíslo
    print(n,end=", ")
print() # závěrečný konec řádky
```

```
Terminal> python3 prvocislo3.py 1000

2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43,
    47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97,
   101, 103, 107, 109, 113, 127, 131, 137, 139,
   149, 151, 157, 163, 167, 173, 179, 181, 191,
   193, 197, 199, 211, 223, 227, 229, 233, 239,
   241, 251, 257, 263, 269, 271, 277, 281, 283,
   293, 307, 311, 313, 317, 331, 337, 347, 349,
   353, 359, 367, 373, 379, 383, 389, 397, 401,
   409, 419, 421, 431, 433, 439, 443, 449, 457,
   461, 463, 467, 479, 487, 491, 499, 503, 509,
   521, 523, 541, 547, 557, 563, 569, 571, 577,
   587, 593, 599, 601, 607, 613, 617, 619, 631,
   641, 643, 647, 653, 659, 661, 673, 677, 683,
   691, 701, 709, 719, 727, 733, 739, 743, 751,
   757, 761, 769, 773, 787, 797, 809, 811, 821,
   823, 827, 829, 839, 853, 857, 859, 863, 877,
   881, 883, 887, 907, 911, 919, 929, 937, 941,
   947, 953, 967, 971, 977, 983, 991, 997,
```

Eratosthenovo síto

eratosthenes.py

```
# -*- coding: utf-8 -*-
# eratosthenes3.py - Vypíše prvočísla menší než zadaný limit

import sys
n=int(sys.argv[1])

p=[True]*n #  p[i] = je 'i' prvočíslo?

for i in range(2,n):
  if p[i]: # je to prvočíslo
      print(i,end=", ")
      j=2*i # označ j=2i,3i,... < n
      while j<n:
        p[j]=False
        j=j+i

print() # závěrečný konec řádky
```

```
Terminal> python3 eratosthenes.py 1000

2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43,
    47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97,
   101, 103, 107, 109, 113, 127, 131, 137, 139,
   149, 151, 157, 163, 167, 173, 179, 181, 191,
   193, 197, 199, 211, 223, 227, 229, 233, 239,
   241, 251, 257, 263, 269, 271, 277, 281, 283,
   293, 307, 311, 313, 317, 331, 337, 347, 349,
   353, 359, 367, 373, 379, 383, 389, 397, 401,
   409, 419, 421, 431, 433, 439, 443, 449, 457,
   461, 463, 467, 479, 487, 491, 499, 503, 509,
   521, 523, 541, 547, 557, 563, 569, 571, 577,
   587, 593, 599, 601, 607, 613, 617, 619, 631,
   641, 643, 647, 653, 659, 661, 673, 677, 683,
   691, 701, 709, 719, 727, 733, 739, 743, 751,
   757, 761, 769, 773, 787, 797, 809, 811, 821,
   823, 827, 829, 839, 853, 857, 859, 863, 877,
   881, 883, 887, 907, 911, 919, 929, 937, 941,
   947, 953, 967, 971, 977, 983, 991, 997,
```

Porovnání rychlosti

```
time -p python3 prvocislo3.py 10000 >/dev/null
time -p python3 eratosthenes.py 10000 >/dev/null
```

	čas [s]	
N	`prvocislo3`	`Eratosthenes`
10^3	0.05	0.05
10^4	0.15	0.06
10^5	2.11	0.24
10^6	53.11	2.19

Dvourozměrné 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]]
```

Dvourozměrné 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))
```

Dvourozměrné 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
```

Dvourozměrné 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])
```

Dvourozměrné 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]
```

Dvourozměrné 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])
```

Dvourozměrné 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]
```

Vytvoření dvourozměrné matice

```
a=[[0.]*4]*2
print_2d_matrix(a)
```

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

Vytvoření dvourozměrné matice

```
a=[[0.]*4]*2
print_2d_matrix(a)
```

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

```
a[0][1]=1.0
print_2d_matrix(a)
```

Vytvoření dvourozměrné matice

```
a=[[0.]*4]*2
print_2d_matrix(a)
```

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

```
a[0][1]=1.0
print_2d_matrix(a)
```

```
[0.0, 1.0, 0.0, 0.0]
[0.0, 1.0, 0.0, 0.0]
```

Vytvoření dvourozměrné matice

```
a=[[0.]*4]*2
print_2d_matrix(a)
```

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

```
a[0][1]=1.0
print_2d_matrix(a)
```

```
[0.0, 1.0, 0.0, 0.0]
[0.0, 1.0, 0.0, 0.0]
```

Nefunguje jak chceme, kvůli aliasingu.

Vytvoření dvourozměrné matice (2)

```
def create_2d_matrix(n,m,v):
    "Creates  a n-by-m matrix with initial value 'v'"
    a=[]
    for i in range(n):
        a+=[ [v]*m ]
    return a
```

Vytvoření dvourozměrné matice (2)

```
def create_2d_matrix(n,m,v):
    "Creates  a n-by-m matrix with initial value 'v'"
    a=[]
    for i in range(n):
        a+=[ [v]*m ]
    return a


a=create_2d_matrix(2,4,0.0)
a[0][1]=1.0
print_2d_matrix(a)
```

```
[0.0, 1.0, 0.0, 0.0]
[0.0, 0.0, 0.0, 0.0]
```

Vytvoření dvourozměrné matice (2)

```
def create_2d_matrix(n,m,v):
    "Creates  a n-by-m matrix with initial value 'v'"
    a=[]
    for i in range(n):
        a+=[ [v]*m ]
    return a
```

```
a=create_2d_matrix(2,4,0.0)
a[0][1]=1.0
print_2d_matrix(a)
```

```
[0.0, 1.0, 0.0, 0.0]
[0.0, 0.0, 0.0, 0.0]
```

Toto je specialita Pythonu

Vytvoření dvourozměrné matice (3)
(array/list comprehension)

```
def create_2d_matrix(n,m,v):
    "Creates  a n-by-m matrix with initial value 'v'"
    return [ [v]*m for i in range(n) ]
```

Vytvoření dvourozměrné matice (3)
(array/list comprehension)

```
def create_2d_matrix(n,m,v):
    "Creates  a n-by-m matrix with initial value 'v'"
    return [ [v]*m for i in range(n) ]


a=create_2d_matrix(2,4,0.0)
a[0][1]=1.0
print_2d_matrix(a)
[0.0, 1.0, 0.0, 0.0]
[0.0, 0.0, 0.0, 0.0]
```

Binomický koeficient
(kombinační číslo)

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \quad \text{for } n \geq k \geq 0$$

- počet k prvkových podmnožin z n

$$\binom{3}{1} = 3, \quad \binom{4}{2} = 6$$

Binomický koeficient
(kombinační číslo)

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \quad \text{for } n \geq k \geq 0$$

- počet k prvkových podmnožin z n

$$\binom{3}{1} = 3, \quad \binom{4}{2} = 6$$

- koeficient v binomické větě

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k$$

$$(x+y)^1 = x + y$$

$$(x+y)^2 = x^2 + 2xy + y^2$$

$$(x+y)^3 = x^3 + 3x^2y + 3y^2x + y^3$$

$$(x+y)^4 = x^4 + 4x^3y + 6x^2y^2 + 4xy^3 + y^3$$

Pascalův trojúhelník
Pascal's triangle

	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$
$n=0$	1				
$n=1$	1	1			
$n=2$	1	2	1		
$n=3$	1	3	3	1	
$n=4$	1	4	6	4	1

Platí rekurze

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Pascalův trojúhelník v Pythonu

Vypočítejte pole `p`, tak aby `p[n][k]` $= \binom{n}{k}$

```
def pascal_triangle(N):
  p=[[1]]
  for n in range(2,N+1):
     prev=p[n-2] # předchozí řada
     p+=[[1] +
          [ prev[k-1] + prev[k]  for k in range(1,n-1) ]
          + [1]]
  return p
```

```
print_2d_matrix(pascal_triangle(5))
```

```
[1]
[1, 1]
[1, 2, 1]
[1, 3, 3, 1]
[1, 4, 6, 4, 1]
```

Řádky pole nemusí mít stejnou délku.

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