Algoritmy a programování

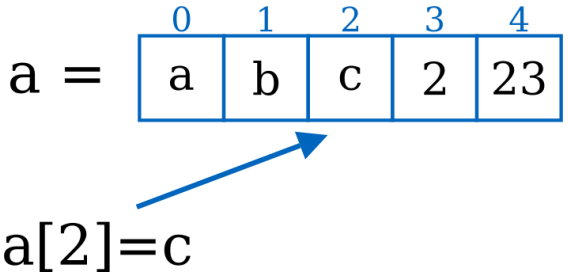
Pole

Vojtěch Vonásek

Department of Cybernetics
Faculty of Electrical Engineering
Czech Technical University in Prague

Pole

- Složená datová struktura
- Obsahuje nula nebo více položek (buněk, cells)
- Položky mohou mít různé datové typy
- Random access (přímý přístup):
  - kdykoliv lze přistoupit na libovolnou položku pole
  - přístup do jednotlivých buněk přes `[]`
- Položky pole jsou přístupné jak pro zápis, tak pro čtení




```
1 a = ["a","b","c",2,23]
2 print(a)
3 a[0] = "ahoj"
4 print(a)
```

```
['a', 'b', 'c', 2, 23]
['ahoj', 'b', 'c', 2, 23]
```

Pole: len()

- Délka pole (počet položek): funkce `len()`

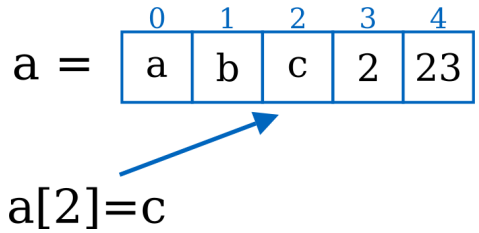
```
1 a = [1,2,3]
2 print( len(a) )
3 a = []
4 print( len(a) )
```

```
3
0
```

Indexace pole

- Buňky jsou číslovány od 0
- První buňka: `a[0]`
- Druhá buňka: `a[1]`

  ...
- Předposlední buňka: `a[len(a)-2]` nebo `a[-2]`
- Poslední buňka: `a[len(a)-1]` nebo `a[-1]`
- `a[-k]` je *k*-tá buňka od konce
- Indexovat lze jen neprázdné pole!




```
1 a = [1,2,3]
2 print("Prvni", a[0] )
3 print("Posledni", a[ len(a)-1] )
4 print("Posledni", a[-1])
```

```
1
3
3
```

Časté chyby při práci s poli

- Indexy musí ukazovat na existující prvky

```
1 a = []
2 print("Prvni", a[0] )
```

```
IndexError: list index out of range
```

```
1 a = ["a","b","c","d"]
2 print(a[4])
```

```
IndexError: list index out of range
```

Časté chyby při práci s poli

- Pozor na indexování proměnné, která není pole

```
1 a = 30
2 print(a[0])
```

```
TypeError: 'int' object is not subscriptable
```

- Obdobně, pokud voláme funkci (vyžadující pole) na proměnnou, která není pole

```
1 a = [1,2,3]
2 a = len(a)  #pozor, prepsani pole!
3 print("Len:", len(a))
```

```
    print("Len:", len(a))
TypeError: object of type 'int' has no len()
```

- Kontrola této chyby nastává až za běhu, ne při/před spuštěním programu
- Může se objevit až po dlouhé době, těžko se ladí

Procházení pole: for+range

- For cyklus + range: do řídicí proměnné se ukládá index buňky

```
1 a = [1,"*", 1/2 ]
2 for i in range( len(a) ):
3     print(a[i])
```

```
1
*
0.5
```

- Pokud je pole prázdné, cyklus neproběhne
- Vhodné pokud je potřeba přistupovat současně k hodnotě i k indexu
- Vhodné pokud je třeba měnit prvky pole
  - např. prohledávání pole, řazení atd.

Procházení pole: for+range

- For cyklus + range: do řídicí proměnné se ukládá index buňky
- Vhodné pro změnu prvků pole

```
1 a = [1,2,3,4]
2 print(a)
3 for i in range( len(a) ):
4     a[i] *= 10    #change of i-th item of a
5 print(a)
```

```
[1,2,3,4]
[10,20,30,40]
```

Procházení pole: for+in

- For cyklus + in: do řídicí proměnné se ukládají hodnoty buněk

```
1 a = [1, 10/2, "word", "last item" ]
2 for item in a:
3     print(item)
```

```
1
5.0
word
last item
```

- Pokud je pole prázdné, cyklus neproběhne
- Hodí se, pokud stačí pracovat s hodnotami (index prvků není důležitý)
  - výpis, hledání prvku, suma, průměry, atd..
- Pozor: změna řídicí proměnné **nemění** prvky v poli

Procházení pole: for+in

- For cyklus + in: do řídicí proměnné se ukládají hodnoty buněk
- Výpočet součtu hodnot pole

```
1 x = [10,20,1]
2 s = 0
3 for value in x:
4     s += value
5 print("Sum", s)
```

```
Sum 31
```

Procházení pole: for+in

- For cyklus + in: do řídicí proměnné se ukládají hodnoty buněk
- Pozor: změna řídicí proměnné **nemění** prvky v poli

```
1 a = [1,2,3,4]
2 print(a)
3 for i in a:
4     i *= 10    #change of i, not of a[i] !!
5 print(a)
```

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

Procházení pole: while

- While cyklus, řídicí proměnná je index pole

```
1 a = [1,2,3]
2 i = 0
3 sum = 0
4 while i < len(a):
5     sum += a[i]
6     i += 1
7 print(sum)
```

```
6
```

Příklad: hledání prvku v poli

- Vstupem je pole a hledaný prvek, máme ho najít a napsat jeho pozici

```
1 a = [0,1,2,3,4,5,6,7,8,9,10]
2 toBeFound = 3
3 for i in range(len(a)):
4     if a[i] == toBeFound:
5         print("We␣found", toBeFound, "at␣position", i)
6 print("end")
```

```
We found 3 at position 3
end
```

- Co se stane, pokud bude hledané číslo v poli několikrát?
- Co se stane, pokud hledané číslo neexistuje?

Příklad: hledání prvku v poli

- Vstupem je pole a hledaný prvek, máme ho najít a napsat jeho pozici

```
1  a = [0,1,2,3,4,5,6,7,8,9,10]
2  toBeFound = 9
3  i = 0
4  while i < len(a) and a[i] != toBeFound:
5      i += 1
6
7  if i != len(a):
8      print("We␣found", toBeFound, "at␣position", i)
9  else:
10     print(toBeFound, "not␣found")
```

```
We found 9 at position 9
```

Příklad: hledání minima v poli

- Vstupem je pole (čísel), chceme najít nejmenší prvek

```
1  def findMin(x): #assume x is array of numbers
2      if len(x) == 0:
3          return None
4      bestMin = x[0]
5      for i in range(len(x)):
6          if x[i] < bestMin:
7              bestMin = x[i]
8      return bestMin
9
10 print( findMin([1,6,1,-1,0]) )
11 print( findMin([0]) )
12 print( findMin([]) )
```

```
-1
0
None
```

Příklad: hledání minima v poli

- Python poskytuje základní funkce: `min()`, `max()`, `sum()`
- Fungují jak pro více argumentů, tak i pro pole
- Předpokládají neprázdné pole
- Předpokládají, že prvky v poli lze vzájemně porovnat

```
1 print( min( [1,2,4,-1] ) )
2 print( min( [1e-3, 2e-4, -4e1] ) )
```

```
-1
-40.0
```

```
1 print( min( [1,2,"ad"] ) )
```

```
    print( min( [1,2,"ad"] ) )
TypeError: '<' not supported between instances of 'str
    ' and 'int'
```

Příklad: hledání minima v poli

- Vstupem je pole (čísel), chceme najít nejmenší **kladný prvek**
- V tomto případě nemůžeme použít Pythoní `min()`

```
1  def findMinPositive(x): #assume x is array of numbers
2      if len(x) == 0:
3          return None
4      bestMin = None
5      for i in range(len(x)):
6          if x[i] > 0 and (bestMin == None or x[i] < bestMin):
7              bestMin = x[i]
8      return bestMin
9
10 print( findMinPositive([ 10,6,-10,-1,0,]) )
11 print( findMinPositive([ 0 ]) )
12 print( findMinPositive([   ]) )
```

```
6
None
None
```

Vytváření, mazání a úprava polí

- Vytvoření prázdného pole: `a = []`
- Vytvoření neprázdného pole: `a = [1,2,3,4,5]`
  - Použití `[]` určuje, že datový typ proměnné `a` je pole
- Přidání prvku do pole: `a.append( prvek )`
- `append()` přidává na konec pole (tj. zprava)

```
1 a = []   #a is array
2 for i in range(5):
3     a.append(i)
4 print(a)
```

```
[0,1,2,3,4]
```

Vytváření, mazání a úprava polí

- Spojování polí: operátor +
- Oba operandy musí být pole

```
1 a = [1,2,10]
2 b = ["one", "two" ]
3 c = a+b
4 print(a)
5 print(b)
6 print(c)
7 b += a
8 print(b)
```

```
[1, 2, 10]
['one', 'two']
[1, 2, 10, 'one', 'two']
['one', 'two', 1, 2, 10]
```

Vytváření, mazání a úprava polí

- Spojování polí: operátor +
- Oba operandy musí být pole

```
1 a = [1,2,10]
2 b = a + 10
3 print(a)
4 print(b)
```

```
    b = a + 10
TypeError: can only concatenate list (not "int") to
   list
```

Vytváření, mazání a úprava polí

- Spojování polí: operátor +
- Oba operandy musí být pole
- Přidání jednoho prvku do pole: `+[prvek]`

```
1 a = [1,2,10]
2 b = a + [10]
3 print(a)
4 print(b)
```

```
[1, 2, 10]
[1, 2, 10, 10]
```

Převod měsíců

- Vstupem je číslo měsíce (1–12)
- Výstupem je jeho jméno (january, . . . , december)
- Řešení přes řadu podmínek
- Uveďte nevýhody tohoto řešení

```
1  def monthName(i):
2      if i == 1:
3          return "january"
4      elif i == 2:
5          return "february"
6      elif i == 3:
7          return "march"
8      ...
9      elif i == 11:
10         return "november"
11     elif i == 12:
12         return "december"
13     else:
14         return "ERROR"
15
16 print( monthName( 3 ) )
17 print( monthName( -3 ) )
```

```
march
ERROR
```

Převod měsíců

- Vstupem je číslo měsíce (1–12)
- Výstupem je jeho jméno (january, . . . , december)
- Dobré řešení je s využitím pole

```
1 months = ["january", "february", "march", "april", "may", "june", "
      july", "august", "september", "october", "november", "december"
       ]
2
3 m = 3
4 print( "Month",m, "is", months[m-1] )  #note m-1 !
```

```
Month 3 is march
```

- Jaké má program výhody a nevýhody?
- (najděte příklad, kdy selže a příklad, kdy bude fungovat nesprávně)

Převod měsíců

- Předchozí program můžeme upravit i pro opačnou úlohu
- Vstup je jméno měsíce, najděme jeho číslo

```
1 months = ["january", "february", "march", "april", "may", "june", "
      july", "august", "september", "october", "november", "december"
       ]
2
3 name = "october"
4
5 for i in range(len(months)):
6     if months[i] == name:
7         print(name,"is", i+1) ## why +1 ????
```

```
october is 10
```

Řezy pole (slices)

Řez: `a[i:j] = [ a[i], a[i+1] ... a[j-1] ]`

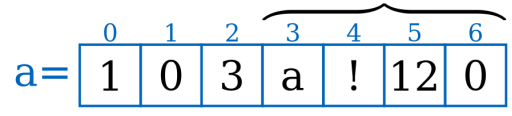
- `a[i:j]` vrací pole od pozice i do pozice j **(kromě j)**
- `a[i:]` pole od pozice i do konce `=a[i:len(a)]`
- `a[:j]` pole od začátku do pozice j **(kromě j)** `=a[0:j]`
- `a[:]` kopie pole `=a[0:len(a)]`
- Výsledek řezu pole je pole
- Řezy polí fungují podobně jako u řetězců
- Výsledkem řezu pole je **vždy pole**

Řezy pole (slices)

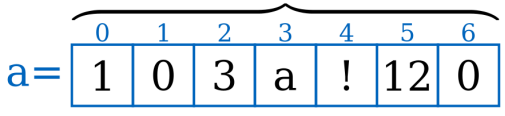

```
1  a = [1,0,3,"a","!",12,0]
2  print(a[3:])
3  print(a[:])
4  print(a[:-3])
5  print(a[3:-3])
6  print(a[3:4])
7  print(a[2:5])
8  print(a[1:-1])
9  print(a[4:4])   #!
10 print("*")
```

```
['a', '!', 12, 0]
[1, 0, 3, 'a', '!', 12, 0]
[1, 0, 3, 'a']
['a']
['a']
[3, 'a', '!']
[0, 3, 'a', '!', 12]
[]
*
```

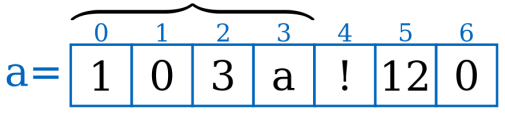
a[3:]=['a', '!', 12, 0]



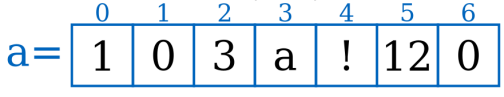
a[:]=[1, 0, 3, 'a', '!', 12, 0]



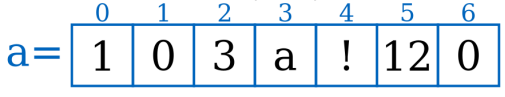
a[:-3]=[1, 0, 3, 'a']



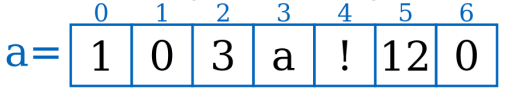
a[3:-3]=['a']



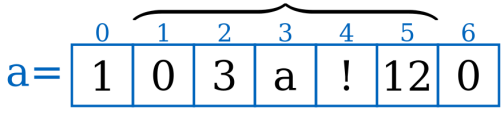
a[3:4]=['a']



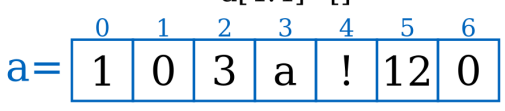
a[2:5]=[3, 'a', '!']



a[1:-1]=[0, 3, 'a', '!', 12]



a[4:4]=[]



Řezy (slices)

- Výsledkem řezu pole je **vždy pole**

```
1 a = ["a","b","c","d","e"]   #list
2 b = a[2:3]                  #list with one element
3 c = a[2]                    #one item of the list
4 print(a,type(a))
5 print(b,type(b))
6 print(c,type(c))
```

```
['a', 'b', 'c', 'd', 'e'] <class 'list'>
['c'] <class 'list'>
c <class 'str'>
```

Pole vs řetězce


```
1 a = "ahoj"
2 b = [1,2,3,4]
```

Řetězce a pole mají v Pythonu společné rysy

- Indexace: `a[i]`
- Řezy: `a[i:j]`
- Délka pole/řetězce: funkce `len()`

Rozdíl mezi řetězci a poli

- Pole obsahuje položky různého datového typu
- Řetězec obsahuje pouze položky typu řetězec
  - i jedno písmeno je v Pythonu uloženo jako string
- Položky pole lze měnit (pole mutable)
- Položky řetězce nelze měnit (řetězec je immutable)

Hodnota vs reference

pythontutor.com/

- Hodnoty immutable datových typů (int,float,string, None, bool,tupple) jsou uloženy přímo v proměnné
- Pokud je proměnná mutable (list (pole),dictionary, object), pak neobsahuje přímo hodnotu, ale referenci do paměti
- Rychlejší předávání argumentů funkcím
  - Nedochází ke kopírování dat, pouze se kopírují reference
- Je třeba znát mechanismus práce s referencemi (důležité při práci s poli,objekty, dictionary, . . .)

Paměťové schéma: immutable typy

```
1 a = "some string"
2 b = 3
3 half = 13/2
4 false = False
```

Global frame	
a	'some string'
b	3
half	6.5
false	False

Paměťové schéma: immutable typy

- Hodnoty immutable datových typů (int,float,string, None, bool,tupple) jsou uloženy přímo v proměnné

```
1 a = "ahoj"
2 b = 10/2
```

Global frame	
a	'ahoj'
b	5.0

Paměťové schéma: immutable typy

- Hodnoty immutable datových typů (int,float,string, None, bool,tupple) jsou uloženy přímo v proměnné

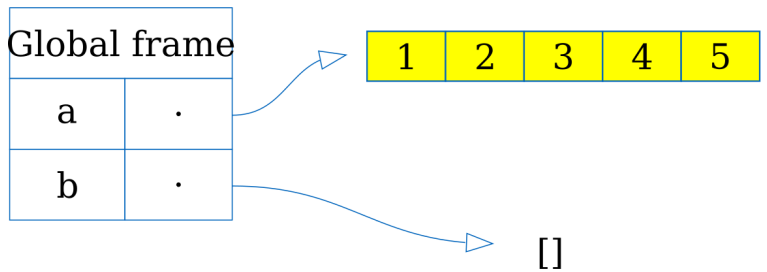
```
1 a = "ahoj"
2 b = 10/2
3
4 c = a
5 d = a
6 e = b
```

Global frame	
a	'ahoj'
b	5.0
c	'ahoj'
d	'ahoj'
e	5.0

Paměťové schéma: mutable typy

- `[]` vytváří nové pole v paměti
- Proměnné a,b jsou typu pole (mutable)
- Hodnota proměnné je reference

```
1 a = [1,2,3,4,5]
2 b = []
```

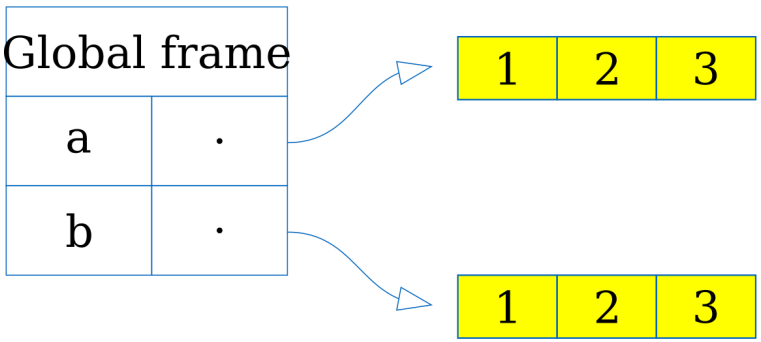


Paměťové schéma: mutable typy

- `[]` vytváří nové pole v paměti
- Proměnné a,b jsou typu pole (mutable)
- Hodnota proměnné je reference

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

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

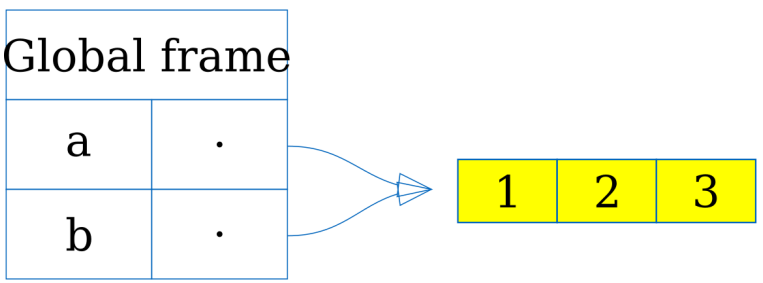


Paměťové schéma: mutable typy

- Přirazení kopíruje reference
- `b=a` znamená, že hodnota (tj. reference) z proměnné `a` se nahraje do proměnné `b`
- Obě proměnné ukazují na stejné pole
- Změna obsahu pole v jedné proměnné se projeví i v druhé proměnné

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

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

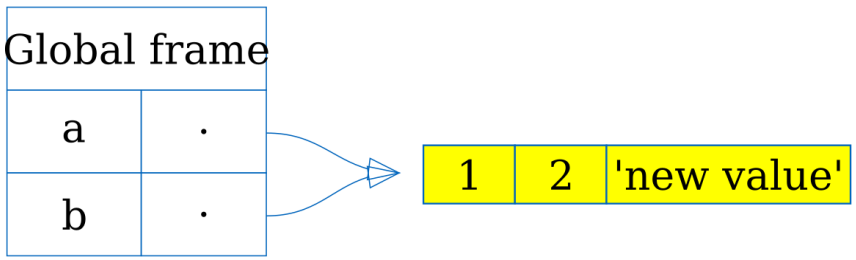


Paměťové schéma: mutable typy

- Přirazení kopíruje reference
- `b=a` znamená, že hodnota (tj. reference) z proměnné `a` se nahraje do proměnné `b`
- Obě proměnné ukazují na stejné pole
- Změna obsahu pole v jedné proměnné se projeví i v druhé proměnné

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

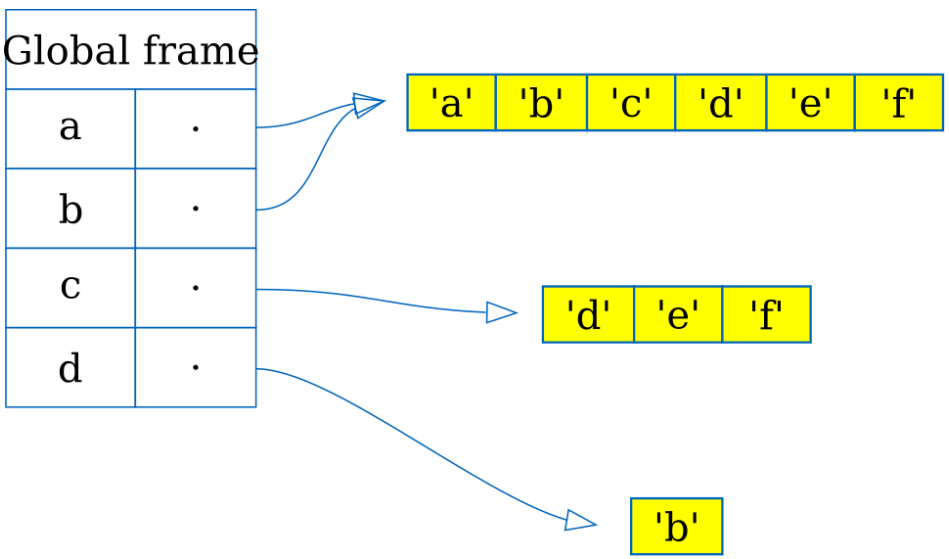
```
[1, 2, 3]
[1, 2, 3]
[1, 2, 'new value']
[1, 2, 'new value']
```



Paměťové schéma: mutable typy

- Řezy pole vytvářejí nové pole

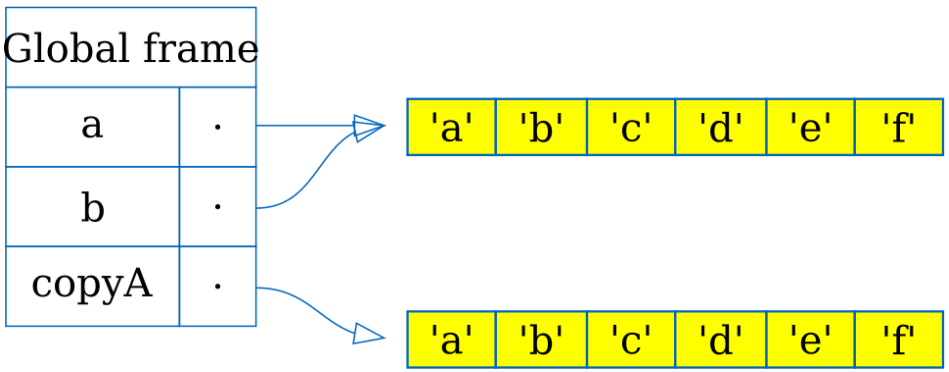
```
1 a = ['a','b','c','d','e','f']
2 b = a
3 c = a[3:]
4 d = a[1:2]
```



Paměťové schéma: mutable typy

- Řezy pole vytvářejí nové pole

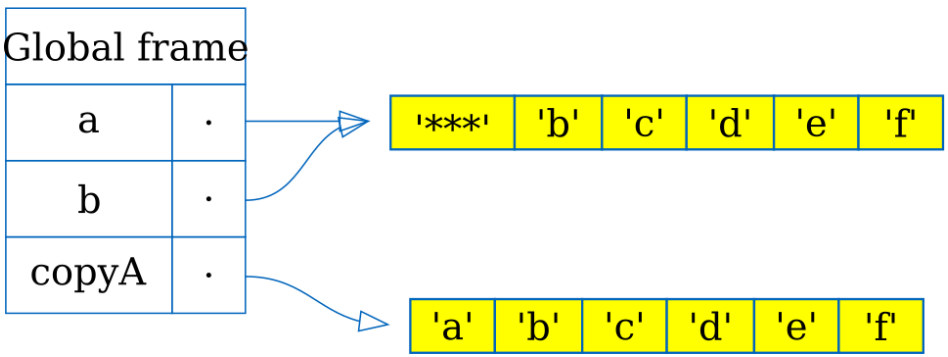
```
1 a = ['a','b','c','d','e','f']
2 b = a
3 copyA = a[:]
```



Paměťové schéma: mutable typy

- Řezy pole vytvářejí nové pole

```
1 a = ['a','b','c','d','e','f']
2 b = a
3 copyA = a[:]
4 b[0] = "***"
5 print(a)
6 print(b)
7 print(copyA)
```




```
['***', 'b', 'c', 'd', 'e', 'f']
['***', 'b', 'c', 'd', 'e', 'f']
['a', 'b', 'c', 'd', 'e', 'f']
```

Pole jako argument funkce

- Pole se do funkce předává jako reference
- Dokud se (ve funkci) reference nezmění, ukazuje na původní pole
- Pokud na odkazované pole přistupujeme přes `[]` nebo přes metody (např. `.append()`, `.pop()` atd.), nejedná se o změnu reference

```
1 def someFunction(a,item): #a is ref to an array
2     a.append(item)        #ref is not changed
3     print("after add:", a)
4
5 b = [1,2]
6 print(b)
7 someFunction(b,3)         #b is ref to array
8 someFunction(b,"last")    #b is ref to array
9 print(b)
```

```
[1, 2]
after add: [1, 2, 3]
after add: [1, 2, 3, 'last']
[1, 2, 3, 'last']
```

Pole jako argument funkce

- Pole se do funkce předává jako reference
- Pokud změníme hodnotu proměnné (operátorem =), ‘ztratí’ spojení s původním polem

```
1 def someFunction(a,item): #a is ref to an array
2     a = a + [item]        #a points to a new array
3     print("after add:", a)
4
5 b = [1,2]
6 print(b)
7 someFunction(b,3)
8 someFunction(b,"last")
9 print(b)
```

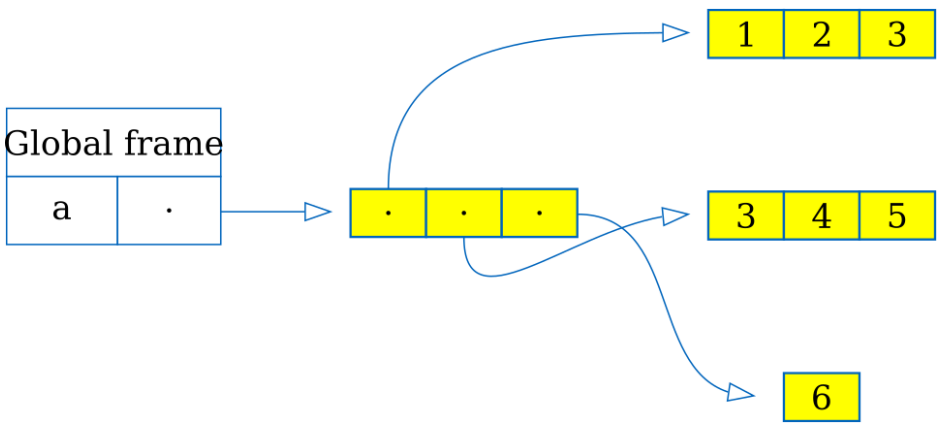
```
[1, 2]
after add: [1, 2, 3]
after add: [1, 2, 'last']
[1, 2]
```

2D pole

- Prvky pole mohou být i další pole
- Vznikají tak více-rozměrná pole (n-dimensional arrays)
- 2D pole: je jednorozměrné pole (řádky), každý řádek odkazuje na další pole, kde jsou uloženy sloupce příslušného řádku

```
1 a = [ [1,2,3], [3,4,5], [6] ]
2 print(a)
3 print(a[1])
4 print(a[2])
```

```
[[1, 2, 3], [3, 4, 5], [6]]
[3, 4, 5]
[6]
```



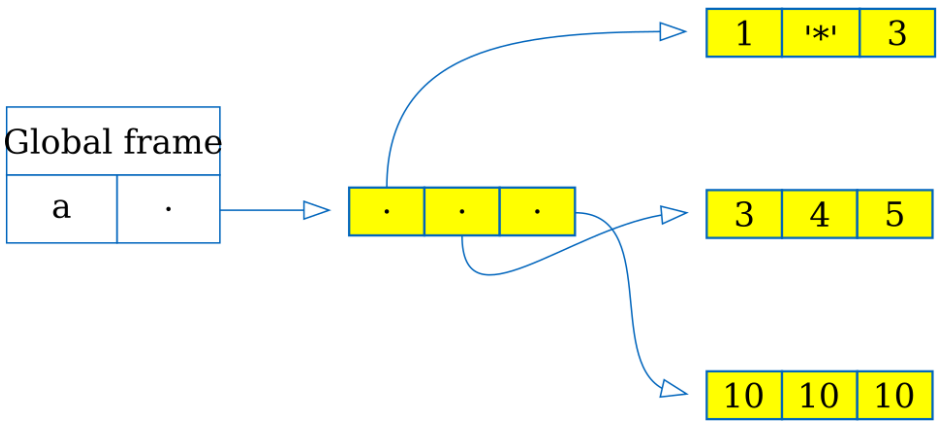
Indexace 2D polí

- "nejdříve řádek, pak sloupec": `a[row][col] = 3`
- toto schéma používáme v ALP

2D pole

- Prvky pole mohou být i další pole
- Vznikají tak více-rozměrná pole (n-dimensional arrays)
- 2D pole: je jednorozměrné pole (řádky), každý řádek odkazuje na další pole, kde jsou uloženy sloupce příslušného řádku

```
1 a = [ [1,2,3], [3,4,5], [6] ]
2 print(a)
3 a[0][1] = "*"
4 a[2] = [10,10,10]
5 print(a)
```




```
[[1, 2, 3], [3, 4, 5], [6]]
[[1, '*', 3], [3, 4, 5], [10, 10, 10]]
```

2D pole: výpis

- `print()` umí vypsat 2D pole v kompaktním zápisu
- Pro práci s 2D poli je vhodné vypisovat pole jako 2D matici

printMatrix.py

```
1 def PM(x): #x is 2D array
2     for row in range(len(x)):
3         for col in range(len(x[row])):
4             print(x[row][col], end="␣")
5         print()
```

```
1 from printMatrix import PM
2
3 a = [ [1,2,3], [4,5,6], [0,0,0] ]
4 PM(a)
```

```
1 2 3
4 5 6
0 0 0
```

2D pole: vytvoření

- Vytvoříme 1D pole: `m=[]`
- Pak do něj přidáme tolik polí, kolik chceme řádků

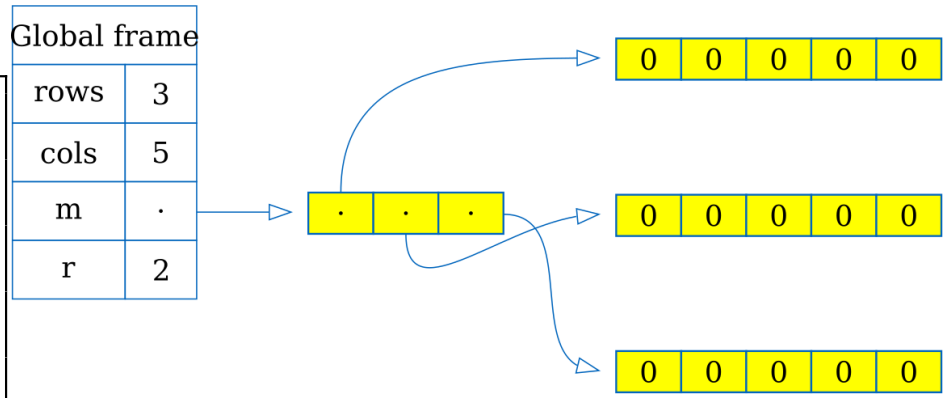
```
1  from printMatrix import PM
2  rows = 3
3  cols = 5
4  m = []
5  for r in range(rows):
6      m.append( [0]*cols )
7  PM(m)
8  print()
9  m[1][2] = "*"
10 PM(m)
```

```
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
```

2D pole: paměť

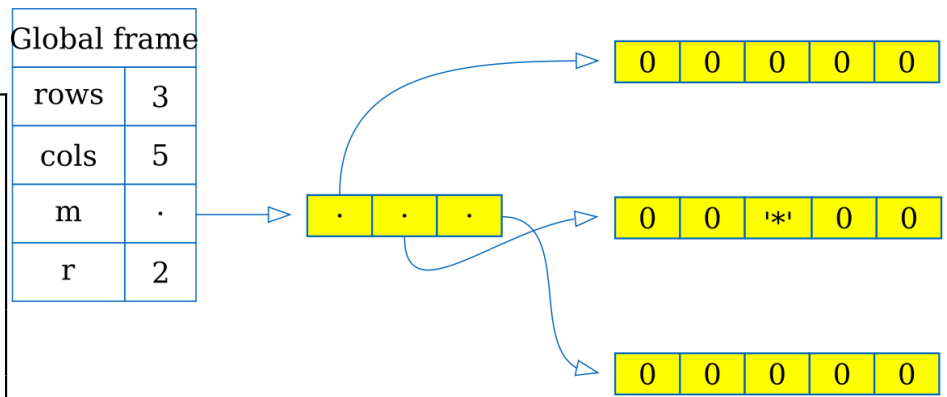
- Vytvoření pole

```
1 rows = 3
2 cols = 5
3 m = []
4 for r in range(rows):
5     m.append( [0]*cols)
```



- Práce s polem

```
1 rows = 3
2 cols = 5
3 m = []
4 for r in range(rows):
5     m.append( [0]*cols)
6 m[1][2] = "*"
```



Pozor 2D pole

- Špatné vytváření 2D pole

```
1 from printMatrix import PM
2 rows = 5
3 cols = 3
4 m = [ [0]*cols ]*rows
5 PM(m)
6
7 m[0][2] = "#"
8 print()
9 PM(m)
```

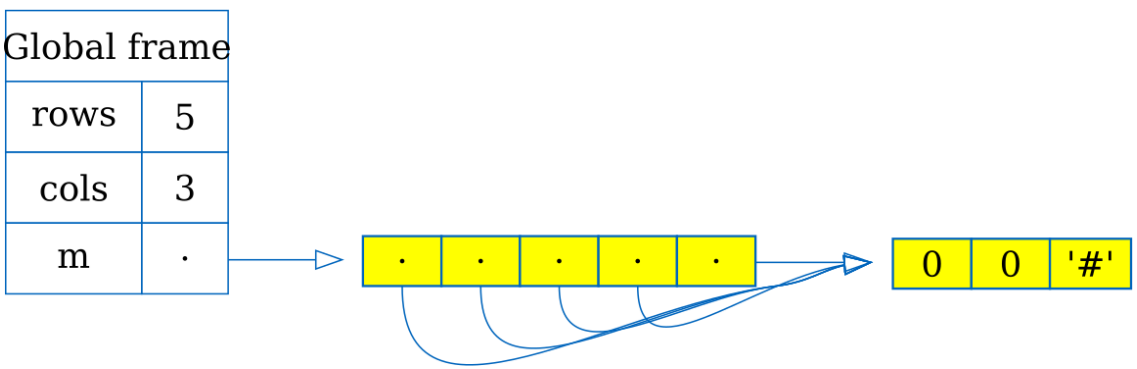
```
0 0 0
0 0 0
0 0 0
0 0 0
0 0 0

0 0 #
0 0 #
0 0 #
0 0 #
0 0 #
```

Pozor 2D pole

- Špatné vytváření 2D pole

```
1 from printMatrix import PM
2 rows = 5
3 cols = 3
4 m = [ [0]*cols ]*rows
5 PM(m)
6
7 m[0][2] = "#"
8 print()
9 PM(m)
```

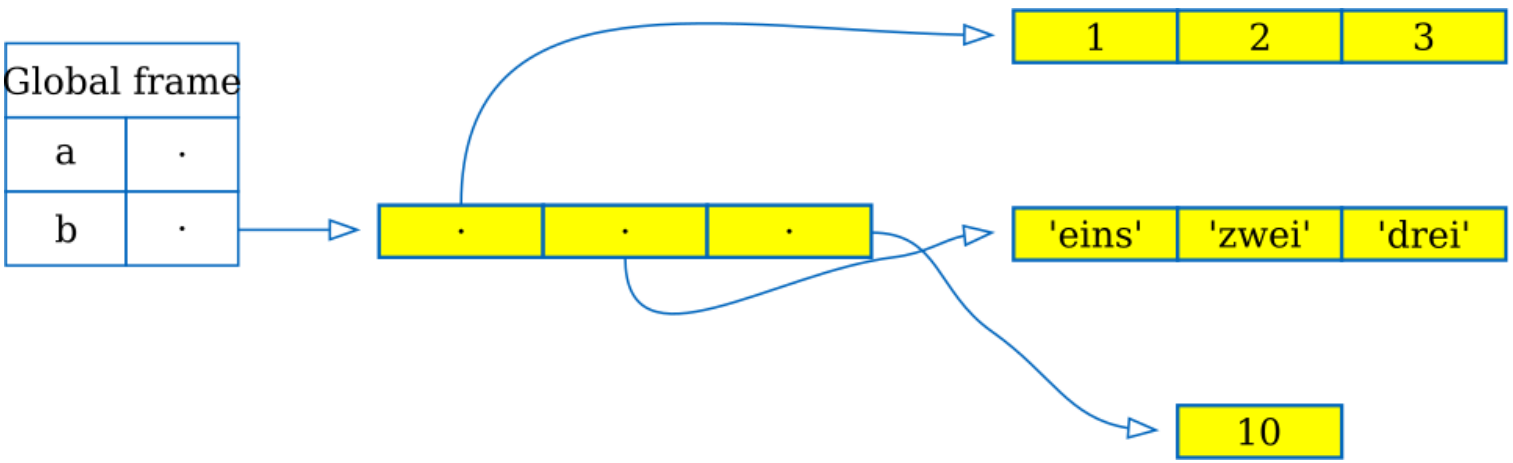
```
0  0  0
0  0  0
0  0  0
0  0  0
0  0  0

0  0  #
0  0  #
0  0  #
0  0  #
0  0  #
```



Kopírování polí

- Proměnná typu pole je reference
- Pokud `a` je pole, pak `b = a` zkopíruje hodnotu (tj referenci) na původní pole
- V tomto případě ukazují `a` i `b` na stejné pole!

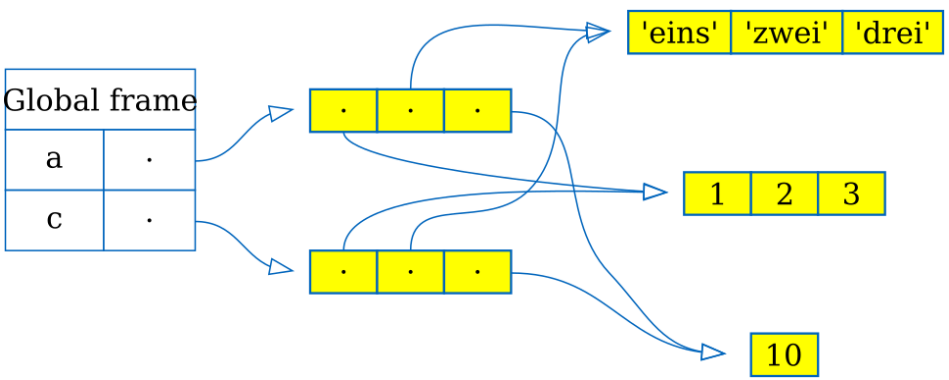
```
1 a=[[1,2,3],["eins","zwei","drei"],[10] ]
2 b = a
```



Kopírování polí: shallow copy

- Proměnná typu pole je reference
- `a[:]` vytvoří nové pole, do kterého zkopíruje hodnoty z původního pole
- Pokud je `a` 1D pole, je výsledkem úplná kopie
- Pokud je `a` vícerozměrné pole, nedojde ke kopii dalších položek
- `a[:]` je tzv. shallow copy

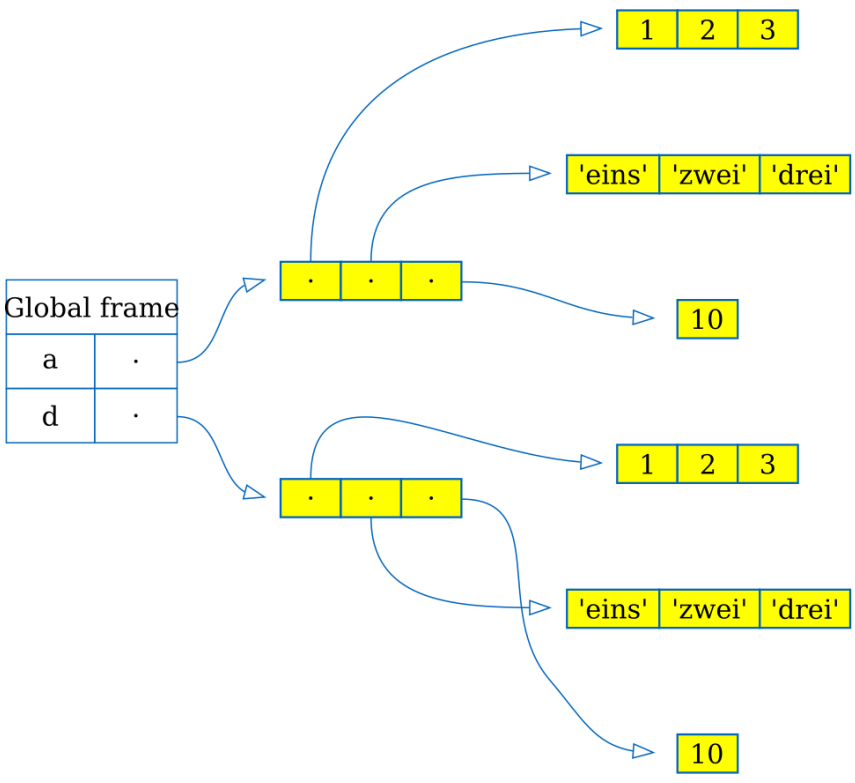
```
1 a=[[1,2,3],["eins","zwei","drei"],[10] ]
2 c = a[:]
```



Kopírování polí: deep copy

- Proměnná typu pole je reference
- Funkce `deepcopy()` z modulu `copy` zajistí úplnou kopii

```
1 import copy
2 a=[[1,2,3],["eins","zwei","drei"],[10] ]
3 d = copy.deepcopy(a)
```



Viditelnost proměnných

- Proměnná vytvořená ve funkci je viditelná pouze v této funkci
- Proměnná vytvořená mimo funkce je globální (a viditelná všude pro čtení)
- Pokud chceme ve funkci změnit globální proměnnou, je třeba `global`

```
1 def fun1():
2     x = 2               #create new local variable x
3     print("1: x",x)
4
5 x = 10
6 print("global: x", x)
7 fun1()
8 print("global: x", x)
```

```
global: x 10
1: x 2
global: x 10
```

Viditelnost proměnných

- Proměnná vytvořená ve funkci je viditelná pouze v této funkci
- Proměnná vytvořená mimo funkce je globální (a viditelná všude pro čtení)
- Pokud chceme ve funkci změnit globální proměnnou, je třeba `global`

```
1  def fun1():
2      global x
3      print("1: x",x)
4      x = 2              #change of global x
5      print("2: x",x)
6
7  x = 10
8  print("global: x", x)
9  fun1()
10 print("global: x", x)
```

```
global: x 10
1: x 10
2: x 2
global: x 2
```