Proměnné, paměť, typy

IB111 Základy programování
Radek Pelánek

2018

Rozcvička I

```
a = [3, 1, 7]
print(sorted(a))
print(a)
b = [4, 3, 1]
print(b.sort())
print(b)
```

Rozcvička II

```
a = ["magic"]
a.append(a)
print(a[1][1][1][1][1][0][1][0][0][0])
```

Dnes

„důležité technické detaily“

- globální a lokální proměnné
- reprezentace dat v paměti, kopírování
- předávání parametrů funkcím
- typy

základní témata obecně důležitá
detaily specifické pro Python

Dnešní programy

Varování: Dnešní programy jsou vesměs „úchylné“.

- minimalistické ukázky důležitých jevů
- nikoliv pěkný, prakticky používaný kód

Python Tutor – vizualizace

Vizualizace použité ve slidech:

`http://www.pythontutor.com`

Odkazy na dnešní ukázky:

`https://www.fi.muni.cz/~xpelanek/IB111/tutor-codes.html`

Doporučeno projít si interaktivně.

Názvy proměnných – konvence

- konstanty – velkými písmeny
- běžné proměnné:
  - smysluplná slova
  - víceslovné názvy: `lower_case_with_underscores`
- krátké (jednopísmenné) názvy:
  - indexy
  - souřadnice: `x, y`
  - pomocné proměnné s velmi lokálním využitím

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 mohou být:
  - funkce
  - moduly (soubory se zdrojovým kódem)
  - třídy (o těch se dozvíme později)
  - a jiné (závisí na konkrétním jazyce)

relevantní terminologie: „namespace“, „scope“

Globální a lokální proměnné

```
a = "This is global."

def example1():
    b = "This is local."
    print(a)
    print(b)

example1()    # This is global.
              # This is local.
print(a)      # This is global.
print(b)      # ERROR!
# NameError: name 'b' is not defined
```

Globální a lokální proměnné

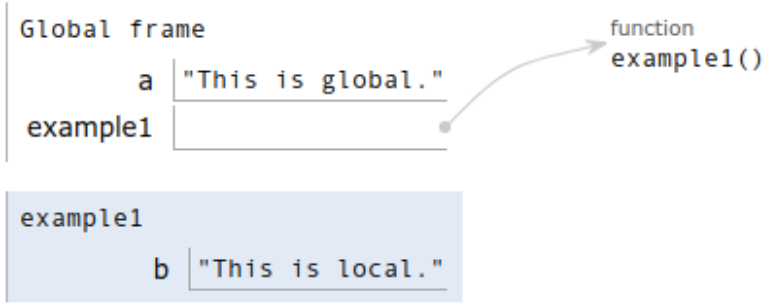
Python 3.6

```
 1  a = "This is global."
 2
 3  def example1():
 4      b = "This is local."
 5      print(a)
 6      print(b)
 7
 8  example1()   # This is global.
 9               # This is local.
10  print(a)     # This is global.
11  print(b)     # ERROR!
```

Edit code | Live programming

Print output (drag lower right corner to resize)

```
This is global.
```

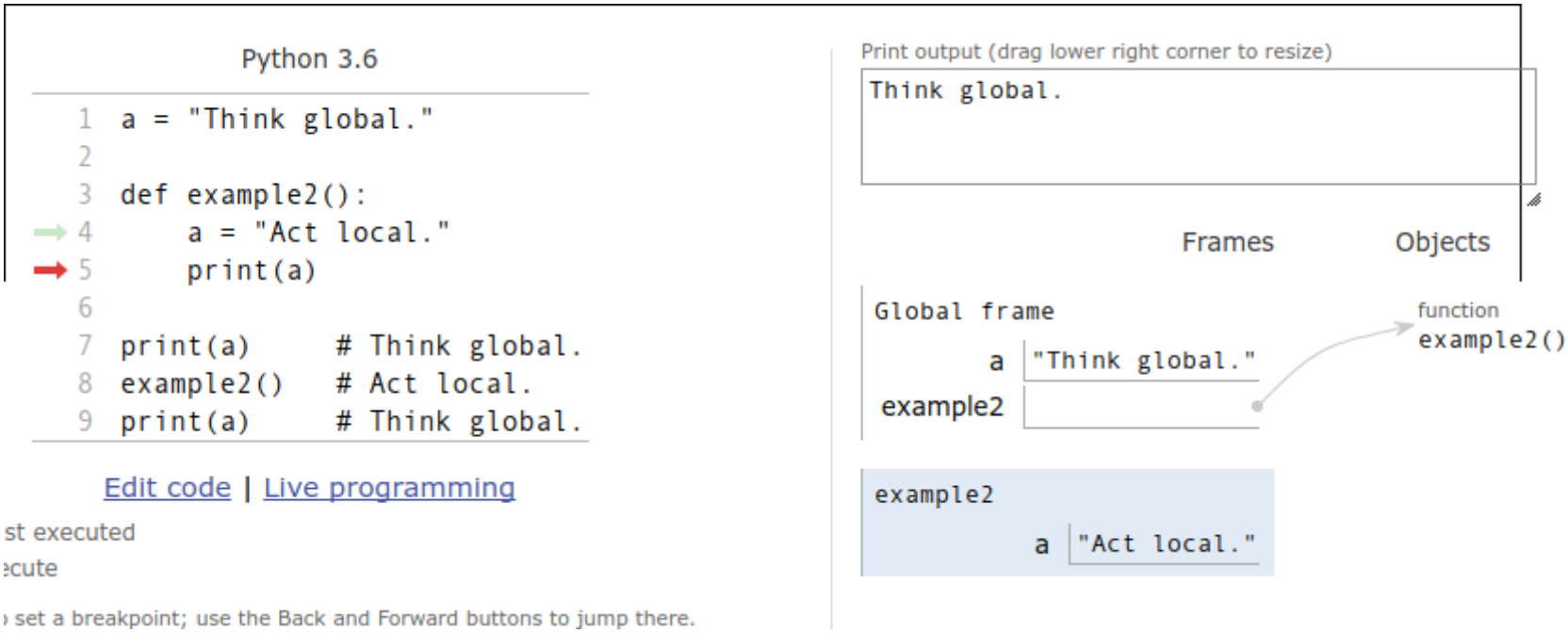


Globální a lokální proměnné

vytváříme novou lokální proměnnou, neměníme tu globální

```
a = "Think global."

def example2():
    a = "Act local."
    print(a)

print(a)      # Think global.
example2()    # Act local.
print(a)      # Think global.
```

Globální a lokální proměnné



Globální a lokální proměnné

Jak měnit globální proměnné?

```
a = "Think global."

def example3():
    global a
    a = "Act local."
    print(a)

print(a)      # Think global.
example3()    # Act local.
print(a)      # Act local.
```

Lokální proměnné: deklarace

lokální proměnná vzniká, pokud je přiřazení kdekoli uvnitř funkce

---

```
a = "Think global."
def example4(change_opinion=False):
    print(a)
    if change_opinion:
        a = "Act local."
        print("Changed opinion:", a)

print(a)        # Think global.
example4()      # ERROR!

# UnboundLocalError: local variable 'a' referenced be
# assignment
```

Globální a lokální proměnné: příklad

```
a = 5

def test1():
    print(a)

def test2():
    print(a)
    a = 8

test1()  # 5
test2()  # UnboundLocalError
```

Rozsah proměnných: for cyklus

- rozsah proměnné v Pythonu není pro „dílčí blok kódu“, ale pro celou funkci (resp. globální kód)
- častá chyba (záludný překlep): proměnná for cyklu použita po ukončení cyklu

---

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

Globální a lokální proměnné

- proměnné interně uloženy ve slovníku
- výpis: `globals()`, `locals()`

```
def function():
    x = 100
    s = "dog"
    print(locals())

a = [1, 2, 3]
x = 200
function()
print(globals())
```

Globální a lokální proměnné

Doporučení:

- vyhýbat se globálním proměnným
- pouze ve specifických případech, např. globální konstanty

Proč?

- horší čitelnost kódu
- náročnější testování
- zdroj chyb

obecně: „lokalita kódu“ je užitečná

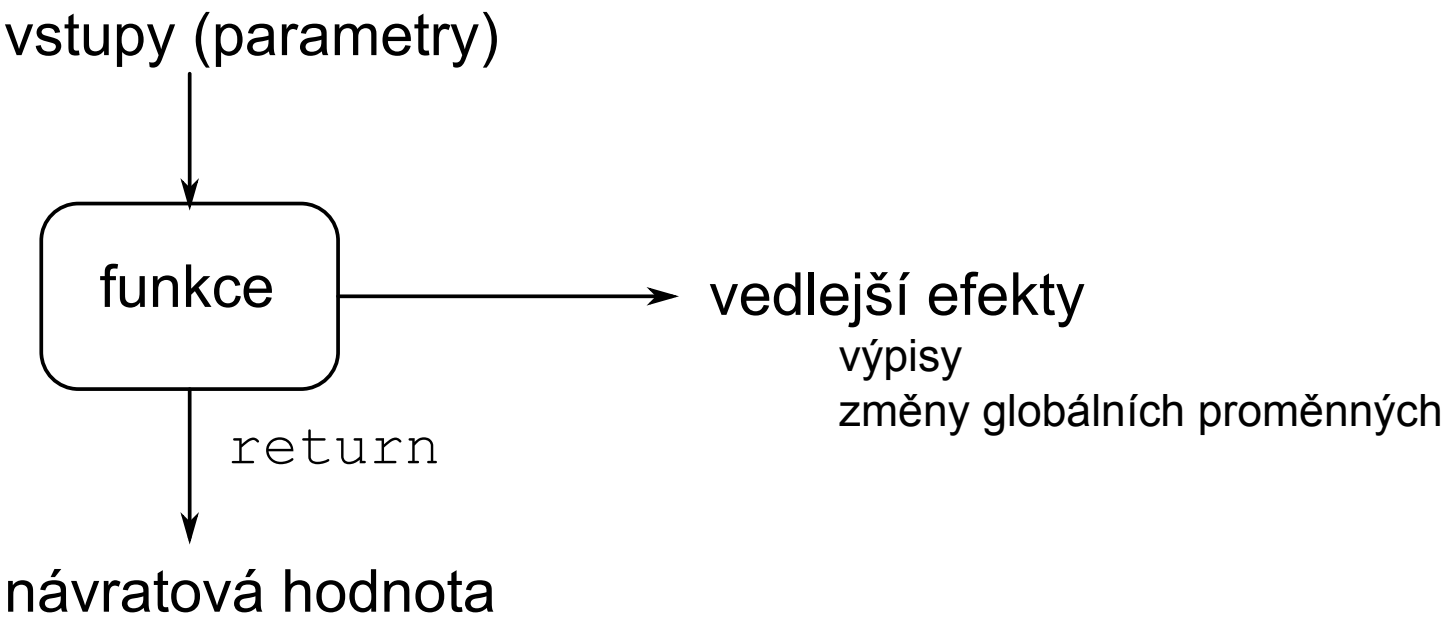
Globální proměnné: alternativy

- předávání parametrů funkcím a vracení hodnot z funkcí
- objekty (probereme později)
- a další (nad rámec předmětu): statické proměnné (C, C++, Java, ...), návrhový vzor Singleton, ...

příklad: herní plán

Funkce

Připomenutí z dřívější přednášky



Funkce bez vedlejších efektů

„čistá funkce“ = funkce bez vedlejších efektů

Reklama

Čisté funkce jsou vaši přátelé!

- ladění
- modularita
- přemýšlení o problému
- čitelnost kódu

Funkce: 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, odeslání na tiskárnu, . . .)
  - nutnost, ale nemíchat chaoticky s výpočty

Proměnné a paměť

Proměnné v různých jazycích

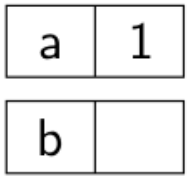
- pojmenované místo v paměti
- odkaz na místo v paměti *(Python)*
- kombinace obou možností

Přiřazení

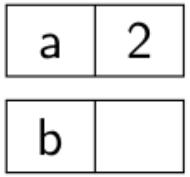
- proměnné ve stylu C: změna obsahu paměti
- proměnné ve stylu Pythonu: změna odkazu na jiné místo v paměti

Proměnné a paměť

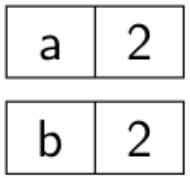
```
int a, b;
a = 1;
```




```
a = 2;
```



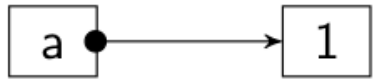

```
b = a;
```



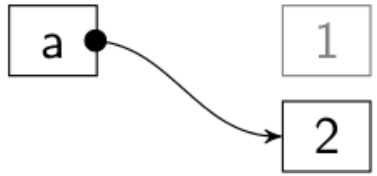
Jazyk C

Proměnné jako hodnoty

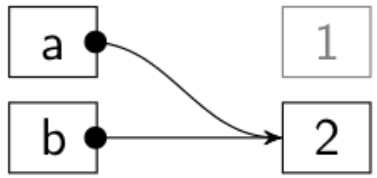
```
a = 1;
```




```
a = 2;
```




```
b = a;
```



Jazyk Python

Proměnné jako odkazy

Proměnné a paměť

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))
```

Rovnost vs. „stejná identita“

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

---

operátor `is` – stejná identita

Předávání parametrů funkcím

- 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

Předávání parametrů hodnotou

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

Předávání parametrů odkazem

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

Předávání parametrů: příklad v C++

```
#include <iostream>

void test(int a, int& b) {
    a = a + 1;
    b = b + 1;
}
int main() {
    int a = 1;
    int b = 1;
    std::cout << "a: " << a << ", b: " << b << "\n";
    test(a, b);
    std::cout << "a: " << a << ", b: " << b << "\n";
}
```

Předávání parametrů v Pythonu

Předávání parametrů v Pythonu

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

Připomenutí:

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

Předávání parametrů: ukázky

číselný parametr je neměnitelný, toto nic neprovede

```
def update_param_int(x):
    x = x + 1

a = 1
print(a)  # 1
update_param_int(a)
print(a)  # 1
```

Předávání parametrů: ukázky

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: ukázky

odkaz se změní na nový seznam, původní je nezměněn – změna se neprojeví mimo funkci

---

```
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: kvíz

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

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

Předávání parametrů: vizualizace

Python 3.6

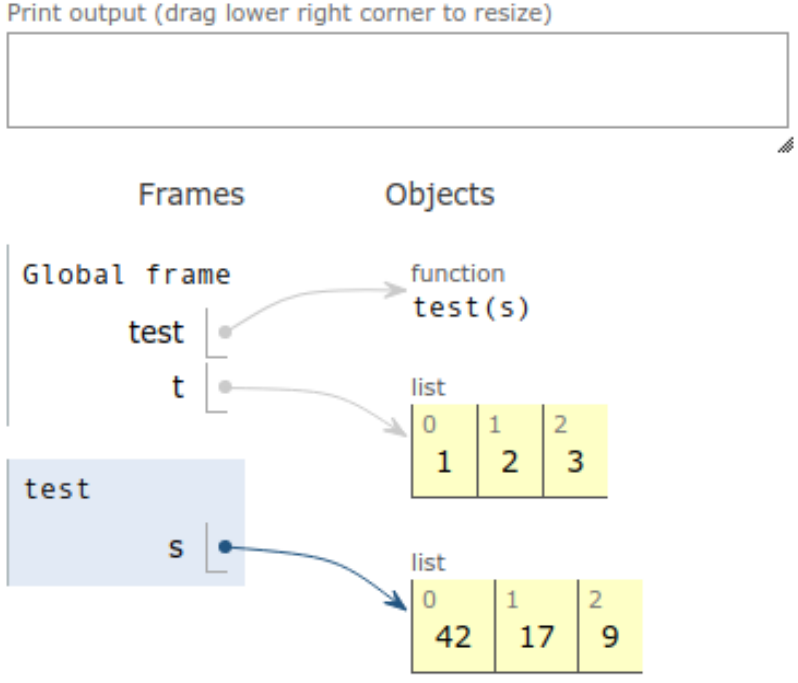
```
1  def test(s):
2      s.append(3)
3      s = [42, 17]
4      s.append(9)
5      print(s)
6
7  t = [1, 2]
8  test(t)
9  print(t)
```

Edit code | Live programming

ted

eakpoint; use the Back and Forward buttons to jump there.

Print output (drag lower right corner to resize)



Práce s parametry: příklad

```
def change_list(alist, value):
    alist.append(value)


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

Předávání parametrů: příklad +=

Operátor +=
různé chování pro neměnné typy a pro seznamy

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

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

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

t = [1, 2, 3]
add_to_list(t)
print(t, id(t))
```

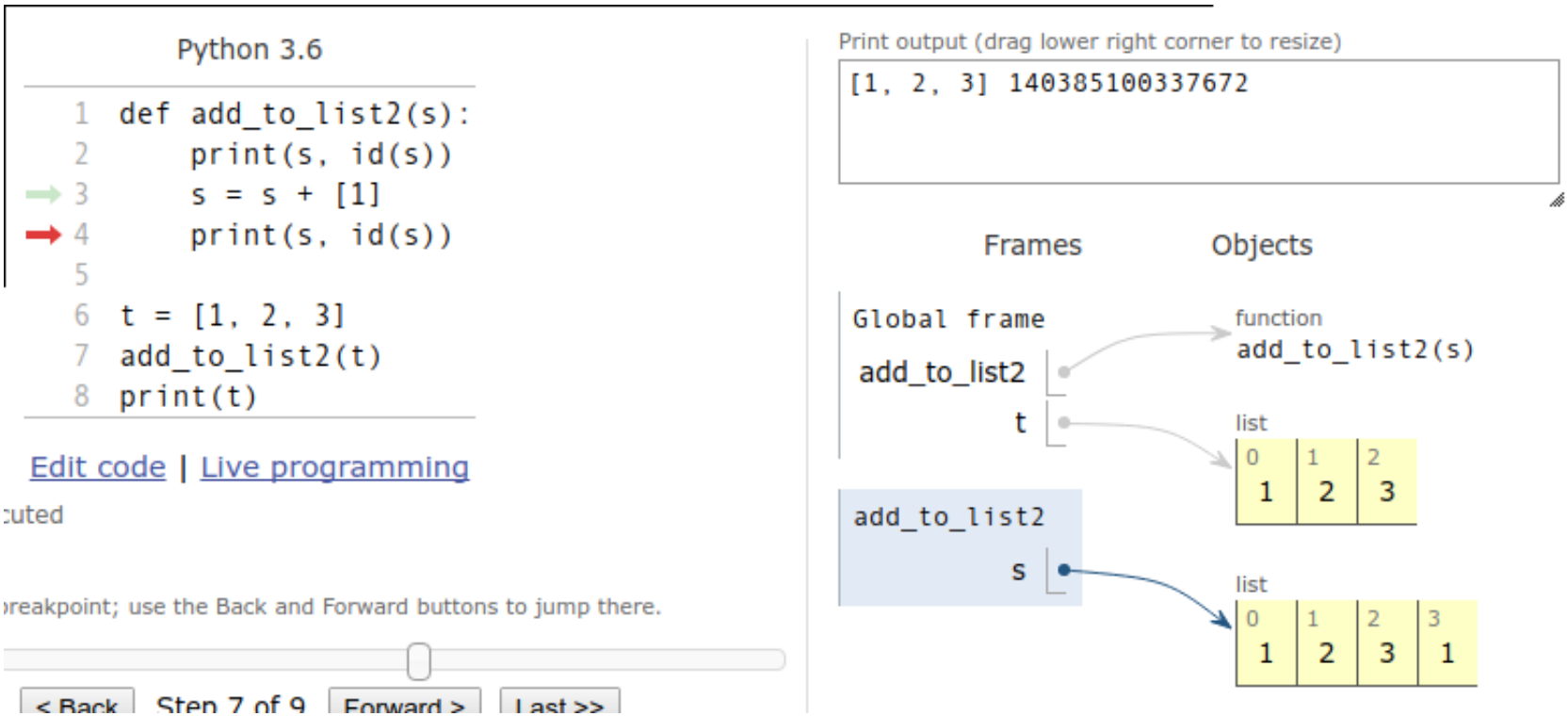
Předávání parametrů: += a =

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, 3]
add_to_list1(t)
print(t)

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

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

t = [1, 2, 3]
add_to_list2(t)
print(t)

# ???
```

Předávání parametrů: vizualizace



Kahoot

Kahoot: Program A

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

def test():
    a = 5
    b[0] = 5

test()
print(a, b[0])
```

Kahoot: Program B

```
def test(b):
    b.append(3)
    b = [4, 5]
    b.append(6)

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

Předávání výsledků funkcí

funkce mají:

- libovolný počet parametrů
- právě jeden výstup (`return x`)

Co když chceme z funkce vrátit více hodnot?

Elementární příklad: dělení se zbytkem

Předávání více výsledků

Jak předat více výsledků?

- n-tice (možno zapisovat i bez závorek)

```
def division_with_remainder(a, b):
    return a // b, a % b

div, mod = division_with_remainder(23, 4)
```

Předávání více výsledků

Jak předat více výsledků?

- slovník („pojmenované“ výstupy)

```
def division_with_remainder(a, b):
    return {"div": a // b, "mod": a % b}

result = division_with_remainder(23, 4)
print(result["div"], result["mod"])
```

Kopírování objektů

Vytvoření aliasu `b = a`

- odkaz na stejnou věc

Mělká kopie `b = a[:]` nebo `b = list(a)`

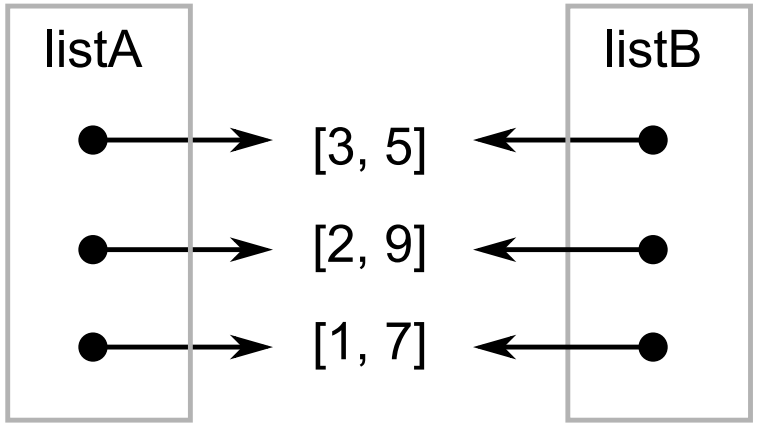
- vytváříme nový seznam, ale prvky tohoto seznamu jsou aliasy
- obecně i pro jiné typy než seznamy (knihovna `copy`)
  - `b = copy.copy(a)`

Hluboká kopie

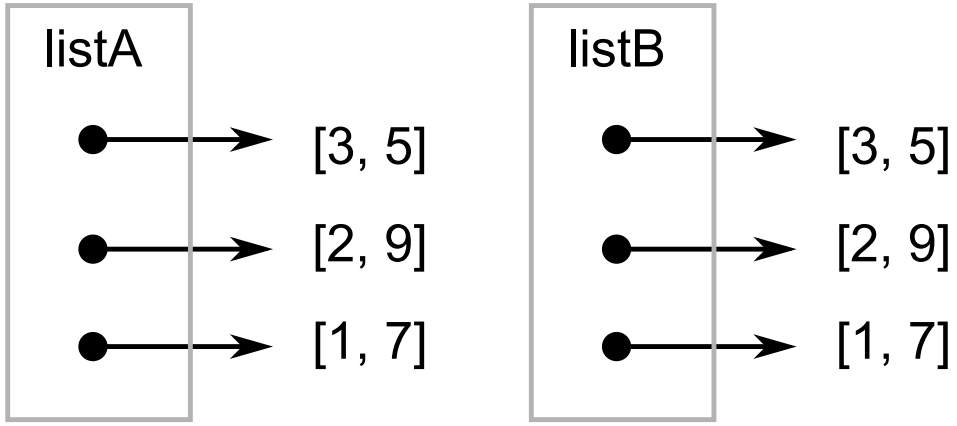
- kompletní kopie všech dat
- obecné řešení (opět knihovna `copy`)
  - `b = copy.deepcopy(a)`

Kopírování objektů

mělká kopie



hluboká kopie



Příklad

```
import copy
a = [[3, 5], [2, 9], [1, 7]]
b = a
c = a[:]
d = copy.deepcopy(a)
a[0][0] = 100
print(b[0][0], c[0][0], d[0][0])
a[0] = [200, 200]
print(b[0][0], c[0][0], d[0][0])
```

Datové typy

Datové typy určují:

- význam dat
- operace, které lze s daty provádět
- hodnoty, kterých mohou data nabývat

Typy v Pythonu

- `bool`
- `int`, `float`, `complex` – číselné typy
- `str` – řetězec
- `list` – seznam
- `tuple` – n-tice
- `dict` – slovník
- `set` – množina

(výběr nejdůležitějších)

Typy: kvíz

```
print(type(3))
print(type(3.0))
print(type(3==0))
print(type("3"))
print(type([3]))
print(type((3,0)))
print(type({3:0}))
print(type({3}))
```

Typy: kvíz

```
type(3)          # <class 'int'>
type(3.0)        # <class 'float'>
type(3==0)       # <class 'bool'>
type("3")        # <class 'str'>
type([3])        # <class 'list'>
type((3,0))      # <class 'tuple'>
type({3:0})      # <class 'dict'>
type({3})        # <class 'set'>
```

Typy: měnitelnost

- neměnitelné (immutable):
  `bool, int, float, str, tuple`
- měnitelné (mutable):
  `list, dict, set`

Příklady, kde důležité:

- změna indexováním
- předávání parametrů funkcím
- indexování slovníku

None

- `None` – jedinečná hodnota typu `NoneType`
- význam: „prázdné“, „žádná hodnota“
- využití: např. defaultní hodnota parametrů funkcí
- implicitní návratová hodnota z funkcí (pokud nepoužijeme `return`)

Pravdivostní hodnota

```
if value:
    print("foo")
```

Pro které z těchto hodnot `value` se vypíše `foo`?

`True`, `False`, `3`, `0`, `3.0`, `-3`, `[3]`, `[]`, `"3"`, `""`, `None`

Pravdivostní hodnota

test je úspěšný (”true”) vždy, kromě následujících případů:

- konstanty: `None`, `False`
- nulové hodnoty u numerických typů: `0`, `0.0`, `0j`
- prázdné sekvence (nulová délka měřená pomocí `len`): ””, [], (),

(mírně zjednodušeno, např. u objektů může být komplikovanější)

Dynamická kontrola typů

- `type(x)` – zjištění typu
- `isinstance(x, t)` – test, zda je proměnná určitého typu

```
values = [3, 8, "deset", 4, "dva", "sedm", 6]
s = 0
for value in values:
    if isinstance(value, int):
        s += value
    else:
        print("Not int:", value)
print("Sum of ints:", s)
```

Typová anotace

- Python používá dynamické typování
  - výhody: stručný, flexibilní kód
  - nevýhody: náročnější ladění, uvažování o problémech, čitelnost
- typové anotace (type hints):
  - volitelné, neovlivňují běh programu
  - možnost statické kontroly (např. `mypy`)
  - integrace v některých IDE

```
def greeting(name: str) -> str:
    return "Hello " + name
```

Načítání vstupu od uživatele: input

```
x = input("Give me a large number:")
x = int(x)
print("My number is larger:", x+1)
```

- `input` vrací řetězec
- typicky nutno přetypovat
- Python2: odlišné chování (`input`, `raw_input`)

Výjimky

- Co když uživatel zadá místo čísla `"deset"`?

- častý přístup: doufat, že se to nestane
- základní přístup: důsledně před každou operací kontrolovat vstupy
  - u složitějších programů nepřehledné
- sofistikovanější přístup: výjimky

Výjimky

```
try:
    x = input("Give me a large number:")
    x = int(x)
    print("My number is larger:", x+1)
except ValueError:
    print("Sorry, that is not a valid number")
```

Nad rámec tohoto kurzu.

Shrnutí

- představa o reprezentaci v paměti je potřeba
- globální, lokální proměnné
- předávání parametrů funkcím
- mělká vs. hluboká kopie
- typy, měnitelné, neměnitelné