

# IB111 Úvod do programování skrze Python

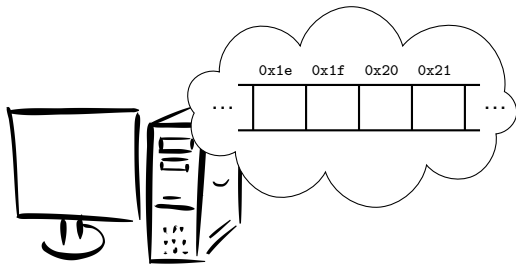
## Přednáška 7

Správa paměti (proměnné podrobněji)  
Práce se soubory

Nikola Beneš

2. listopad 2016

## Proměnné a paměť



## Proměnné

- něco, co drží hodnotu
- jejich hodnota se může během výpočtu měnit

## Názvy proměnných (v Pythonu)

- posloupnost písmen, číslic a znaků '\_'
- bez mezer, více slov pomocí:
  - podtržíték: dlouhy\_nazev\_promenne
  - střídání velikosti písmen: dlouhyNazevPromenne
- nelze používat rezervovaná klíčová slova jazyka

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

## Globální proměnné

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

## Lokální proměnné

- definovány uvnitř funkce
- jsou viditelné jen ve své funkci

## Obecněji:

- proměnné jsou viditelné v rámci svého bloku
- blokem mohou být:
  - moduly (soubory se zdrojovým kódem)
  - funkce
  - třídy (o těch se dozvíme později)
  - a jiné (závisí na konkrétním jazyce)

# Globální a lokální proměnné v Pythonu

## Příklad 1

```
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)    # CHYBA!
```

```
# NameError: name 'b' is not defined
```

## Příklad 2

```
a = "Think global."
```

```
def example2():  
    a = "Eat local."  
    print(a)
```

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

- vytváříme novou lokální proměnnou, neměníme tu globální

## Příklad 3

- jak měnit globální proměnné?

```
a = "Think global."
```

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

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

# Globální a lokální proměnné v Pythonu

## Příklad 4

- lokální proměnná vzniká přiřazením **kdekoli uvnitř funkce**

```
a = "Think global."
```

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

```
print(a)      # Think global.  
example4()   # CHYBA!
```

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



# Globální a lokální proměnné

## Doporučení

- pokud možno nepoužívat globální proměnné
- globální *konstanty* jsou v pořádku

## Proč?

- porušují lokalitu kódu
- potenciální chyby
- nečitelnost

## Alternativy

- předávání parametrů funkcím a vracení hodnot z funkcí
- objekty (o těch si něco málo řekneme později)
- a další (nad rámec předmětu)
  - statické proměnné (C, C++, Java, ...)
  - návrhový vzor Singleton
  - ...

## Proměnné v různých jazycích

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

## Přířazení

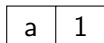
- proměnné ve stylu C: změna obsahu paměti
- proměnné ve stylu Pythonu: změna odkazu na jiné místo v paměti

# Proměnné podrobněji

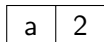
## Ilustrace přiřazení

```
int a, b;
```

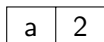
```
a = 1;
```



```
a = 2;
```



```
b = a;
```



Jazyk C

Proměnné jako hodnoty

```
a = 1;
```



```
a = 2;
```



```
b = a;
```



Jazyk Python

Proměnné jako odkazy

# Proměnné podrobněji

## Příklad

```
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))
```

[další ukázky, použití <http://pythontutor.com>]

## Způsoby předávání parametrů

- 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 referencí
  - podobně funguje např. Java
  - někdy nazýváno *call by object sharing*

## 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

## Příklad předávání v C++

```
#include <iostream>

void fun(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";

    fun(a, b);

    std::cout << "a: " << a << ", b: " << b << "\n";
}
```

# Předávání parametrů funkcím

## Předávání parametrů v Pythonu

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

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

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



# Předávání parametrů funkcím

## Operátor +=

- různé chování pro neměnné typy a pro seznamy

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

```
p = 2015  
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 funkcím

**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)
```

*# výsledek je [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)
```

*# jaký je výsledek?*

## Různé přístupy ke správě paměti

- manuální – funkce pro přidělení/uvolnění paměti
- automatická – paměť je uvolněna na konci života proměnné
- automatická – počítání referencí
  - kolik částí programu ještě s danou pamětí pracuje
  - pokud už nikdo, paměť je uvolněna
- automatická – garbage collection
  - jednou za čas se uklidí nepoužívaná paměť

## Správa paměti v Pythonu

- automatická – počítání referencí + někdy i větší úklid
- počet referencí `sys.getrefcount(object)`

## Zvýšení počtu odkazů

- vytvoření
- vytvoření aliasu
- předání funkci
- vložení do složeného prvku

```
a = "Hello!"  
b = a  
fun(a)  
s = [42, a, -7]
```

## Snížení počtu odkazů

- ukončení platnosti lokální proměnné
- smazání proměnné
- přiřazení jiné hodnoty aliasu
- odstranění ze složeného prvku
- odstranění složeného prvku

```
konec funkce  
del a  
b = "Aloha!"  
s.remove(a)  
del s
```

## Pamatování některých hodnot

- některé hodnoty si Python automaticky udržuje v paměti a nevytváří je znovu
- kvůli vyšší rychlosti / výkonu
- často používané hodnoty
- čísla mezi -5 a 256
- prázdný řetězec, jednotlivé znaky
- krátké řetězce se vytváří jen jednou

```
import sys
a = 1
print(sys.getrefcount(a))
a = ""
print(sys.getrefcount(a))
```

# 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 – jak?
- obecné řešení (opět knihovna `copy`)
  - `b = copy.deepcopy(a)`

## Práce se soubory



## Způsob práce

- otevření souboru
- práce se souborem (čtení / zápis)
- zavření souboru

```
>>> my_file = open("/tmp/my_file", "w")
>>> print(my_file)
<_io.TextIOWrapper name='/tmp/my_file' mode='w' encoding='UTF-8'>
>>> my_file.write("Hasta la vista!\n")
>>> my_file.close()
```



## Otevření souboru

- `open(jmeno, zpusob)`
- jméno souboru: řetězec (pozor na Windows a '\\')
- způsob otevření:
  - čtení ("r")
  - zápis ("w") – přepíše soubor, pokud není, vytvoří jej
  - přidání na konec ("a")
  - čtení i zápis ("r+", "w+" nebo "a+")
  - binární režim (přidat "b" k některému způsobu)

## Čtení ze souboru

- `read(pocet)` – přečte daný počet znaků
- `read()` – přečte celý soubor, vrací řetězec
- `readline()` – přečte celý jeden řádek
- `readlines()` – přečte všechny řádky, vrací seznam řádků

## Zápis do souboru

- `write(text)` – zapíše řetězec do souboru
  - neukončuje řádky, je třeba explicitně použít `'\n'`

## Jiné

- `tell()` – aktuální pozice v souboru
- `seek(pozice)` – přesun pozice v souboru

## Iterace po řádcích

```
for line in my_file:  
    print(line)
```

## Speciální blok `with`

- při jeho použití není třeba soubor zavírat
- zavře se sám při ukončení bloku

```
with open("/tmp/my_file", "r") as my_file:  
    lines = my_file.readlines()  
  
print(lines)
```

## Proměnné

- lokální vs. globální (používejte jen jako konstanty)
- v Pythonu: drží odkaz na hodnotu

## Předávání parametrů

- v Pythonu: něco mezi hodnotou a odkazem (*call by sharing*)

## Správa paměti

- v Pythonu: počítání referencí, automatický úklid

## Kopírování objektů

- mělká vs. hluboká kopie
- užitečná knihovna `copy`

## Práce se soubory

- otevření / zavření souboru; užitečný blok `with`
- zápis, čtení