Práce se soubory
Správa paměti

IB111

Práce se soubory v Pythonu

- Soubor musíme „otevřít“
- Poté s ním pracujeme
  - Čteme a/nebo zapisujeme
- Nakonec musíme soubor „zavřít“

```
>>> f = open('/tmp/workfile', 'w')
>>> print f
<open file '/tmp/workfile', mode 'w' at 80a0960>
```

```
>>> f.write('0123456789abcdef')
```

```
>>> f.close()
```

Otevírání souboru

- Příkaz open(soubor, způsob)
  - Jméno souboru
  - Způsob otevření
    - Čtení (r)
    - Zápis (w)
    - Čtení i zápis (r+)
    - Přidání dat nakonec (a)
    - Binární režim (b)

Práce se souborem

- Čtení: read()
- Čtení jednoho řádku: readline()
- Čtení pro vytvoření *seznamu* řádků: readlines()
- Zápis: write(string)
- Zjištění aktuální pozice (offsetu) v souboru: tell()
- Přesun aktuální pozice souboru: seek()

Práce se soubory: příklad

```
>>> value = ('the answer', 42)
>>> s = str(value)
>>> f.write(s)
```

```
>>> f = open('/tmp/workfile', 'r+')
>>> f.write('0123456789abcdef')
>>> f.seek(5)      # Go to the 6th byte in the file
>>> f.read(1)
'5'
>>> f.seek(-3, 2) # Go to the 3rd byte before the end
>>> f.read(1)
'd'
```

Zkratka přes with

- Při použití „with“ není třeba soubor explicitně zavírat

```
>>> with open('/tmp/workfile', 'r') as f:
...     read_data = f.read()
>>> f.closed
True
```

Jak se pracuje s proměnnými a pamětí v jazyce Python

IB111

Globální a lokální proměnné

Příklad 1: Rozdíl mezi lokální a globální proměnnou

```
gl_a = 'neco'
def priklad1():
    lc_a = 'jineho'
    print gl_a
    print lc_a

priklad1()
print gl_a
print lc_a
```

```
>>>
neco
jineho
neco

Traceback (most recent call last):
  File "C:/Documents and Settings/Administrat
or/Plocha/priklad1.py", line 9, in <module>
    print lc_a
NameError: name 'lc_a' is not defined
>>>
```

Příklad 2: Jak nelze nastavit globální proměnnnou…

```
a = ' neco'
def priklad2():
    a = 'jineho'
    print 'Uvnitr funkce je hodnota a: ', a

priklad2()
print 'Mimo funkci je hodnota a:', a
```

```
>>>
Uvnitr funkce je hodnota a:  jineho
Mimo funkci je hodnota a:  neco
>>>
```

Příklad 3: Jak správně změnit globální proměnnou…

```
a = ' neco'
def priklad3():
    global a
    a = 'jineho'
    print 'Uvnitr funkce je hodnota a: ', a

priklad3()
print 'Mimo funkci je hodnota a:', a
```

```
>>>
Uvnitr funkce je hodnota a:  jineho
Mimo funkci je hodnota a: jineho
>>>
```

Příklad 4: Vnořené funkce

```
def priklad4():
    vnejsi_a = 'neco'
    def priklad4_vnitrni():
        vnitrni_a = 'jineho'
        print vnejsi_a
        print vnitrni_a
    priklad4_vnitrni()
    print vnejsi_a
    print vnitrni_a

priklad4()
```

```
>>>
neco
jineho
neco

Traceback (most recent call last):
  File "C:/Documents and Settings/Administrat
or/Plocha/priklad4.py", line 11, in <module>
    priklad4()
  File "C:/Documents and Settings/Administrat
or/Plocha/priklad4.py", line 9, in priklad4
    print vnitrni_a
NameError: global name 'vnitrni_a' is not def
ined
>>>
```

Příklad 5: Vnořené funkce – jak nezměníte vnější proměnnou…

```
def priklad5():
    a = 'neco'
    def priklad5_vnitrni():
        a = 'jineho'
        print a
    priklad5_vnitrni()
    print a

priklad5()
```

```
>>>
jineho
neco
>>>
```

Příklad 6: Vnořené funkce

```
def priklad6():
    priklad6.a = 'neco'
    def priklad6_vnitrni():
        priklad6.a = 'jineho'
        print priklad6.a
    priklad6_vnitrni()
    print priklad6.a

priklad6()
#takto vlastně vzniká globální proměnná…
```

```
>>>
jineho
jineho
>>> print priklad6.a
jineho
>>>
```

Viditelnost proměnných

- Proměnné jsou viditelné v rámci bloku
- Blokem jsou:
  - Moduly
  - Funkce
  - Třídy
  - Interaktivní příkaz
- Každé využití jména (proměnné) se odkazuje na nejvnitřnější blok, který jméno obsahuje

Příklad využití globálních proměnných

```
from Tkinter import *

root = Tk()

body=0

def callback(event):
    print "Kliknuto na pozici", event.x, "x", event.y
    global body
    if event.x==50 and event.y==50:
        body+=100
        print "Ziskali jste bonus 100 bodu. Celkem mate:",body,"bodu."

frame = Frame(root, width=100, height=100)
frame.bind("<Button-1>", callback)
frame.pack()

root.mainloop()
```

Přiřazování proměnných

Názvy proměnných

- Posloupnost písmen, číslic a znaků '_'
- Nelze použít 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
```

- Názvy nesmí obsahovat mezery
  - v praxi se místo mezer používá podtržítko nebo střídání velikosti písmen, např.
    - dlouhy_nazev_promenne
    - DlouhyNazevPromenne

Přiřazování hodnot proměnným

```
int a = 1;
```


```
a = 2;
```


```
int b = a;
```

Jazyky C/C++

```
a = 1
```


```
a = 2
```


```
b = a
```

Jazyk Python

Příklad (intuitivní)

```
>>> import sys
>>> a=1001
>>> b=a
>>> print a,b
1001 1001
>>> print id(a),id(b)
13090056 13090056
>>> print sys.getrefcount(a), sys.getrefcount(b)
3 3
>>> b+=1
>>> print a,b
1001 1002
>>> print id(a),id(b)
13090056 13837004
>>> print sys.getrefcount(a), sys.getrefcount(b)
2 2
>>>
```

Příklad (méně intuitivní)

```
>>> import sys
>>> a=[1]
>>> b=a
>>> print a,b
[1] [1]
>>> print id(a),id(b)
20027712 20027712
>>> print sys.getrefcount(a), sys.getrefcount(b)
3 3
>>> b.append(2)
>>> print a,b
[1, 2] [1, 2]
>>> print id(a), id(b)
20027712 20027712
>>> print sys.getrefcount(a), sys.getrefcount(b)
3 3
>>>
```

Předávání parametrů funkci

- Způsoby předávání parametrů
  - Hodnotou (call-by-value)
    - Používáno standardně například v C/C++ a Pascalu
  - Odkazem (call-by-reference)
    - popř. jménem (call-by-name) – Algol/Cobol
    - Používáno standardně v Perlu
    - Možné použít i v C/C++ a Pascalu
  - Něco mezitím :-)
    - A to je případ jazyka Python

Předávání parametrů hodnotou

- Při volání funkce se předávaný výraz vyhodnotí a ve volané funkci pak „žije vlastním životem“
- Parametr je tak vlastně lokální proměnná inicializovaná na hodnotu předanou volajícím
- Funkce tak pracuje nad svou lokální kopií hodnoty
- Funkce nemůže změnit hodnotu proměnné, předávané jako parametr, tak aby změna byla viditelná ve volající funkci

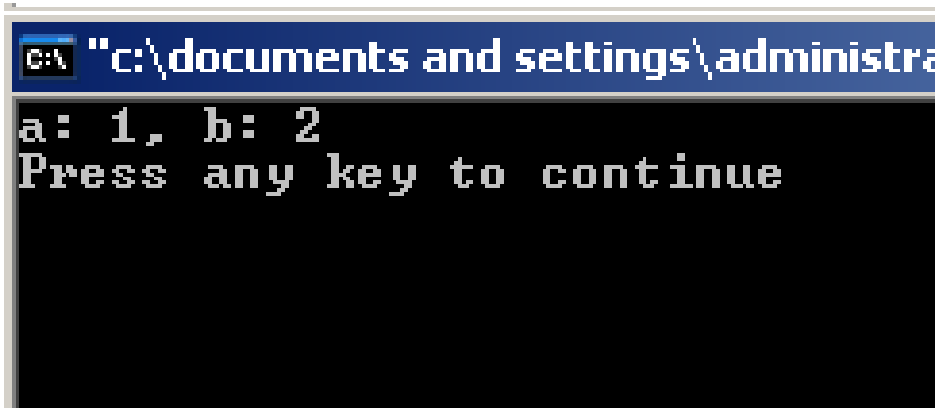
Předávání parametrů odkazem

- Při volání se nepředává aktuální hodnota proměnné, ale pouze odkaz (ukazatel, pointer) na proměnnou
- Jakékoliv změny této proměnné uvnitř volané funkce se projeví i ve funkci volající
- Nevzniká lokální kopie proměnné, pracuje se rovnou s původní proměnnou

Příklad v jazyce C++

```
void funkce(int a, int &b)
{
    a++;
    b++;
}

int main()
{
    int a=1;
    int b=1;
    funkce(a,b);
    printf("a: %i, b: %i\n",a,b);
    return 0;
}
```



Předávání parametrů v jazyce Python

- U proměnných, které jsou nezměnitelné, se používá předávání parametrů hodnotou
  - resp. zdá se, že je použito volání odkazem, jakmile ale dojde ke změně hodnoty, jedná se o volání hodnotou
  - Řetězce, čísla, n-tice (tuples)
- V jiných případech se používá předávání odkazem
  - Funkce může měnit hodnotu parametrů a změna se tyká i volající funkce

Příklad v Pythonu – nezměnitelné typy

```
>>> def funkce(x):
        print x,'(',id(x),')'
        x+=1
        print x,'(',id(x),')'
```

„Volání hodnotou“

```
>>> promenna=1001
>>> id(promenna)
13541700
>>> funkce(promenna)
1001 ( 13541700 )
1002 ( 13541652 )
>>>
```

Příklad v Pythonu – měnitelné typy

```
>>> def funkce1(seznam):
        print seznam, id(seznam)
        seznam=[1001, 1002]
        print seznam, id(seznam)
```

```
>>> def funkce2(seznam):
        print seznam,id(seznam)
        seznam+=[1001,1002]
        print seznam,id(seznam)
```

```
>>> s=[1,2,3]
>>> funkce1(s)
[1, 2, 3] 20201344
[1001, 1002] 20181320
>>>
>>> s=[1,2,3]
>>> funkce2(s)
[1, 2, 3] 20092328
[1, 2, 3, 1001, 1002] 20092328
>>> print s
[1, 2, 3, 1001, 1002]
>>>
```

```
>>> s=[1,2,3]
>>> funkce2(s[:])
[1, 2, 3] 20200824
[1, 2, 3, 1001, 1002] 20200824
>>> print s
[1, 2, 3]
>>> print id(s)
20199384
>>>
>>>
```

Správa paměti

- Automatická správa paměti
  - Garbage collection
- Python si sám sleduje využívání paměťových míst
  - Jakmile se něco přestane používat, paměť je automaticky uvolněna
- Programátor se o správu paměti nemusí moc starat
  - Přesto je dobré vědět, jak to funguje
- Python si sleduje počet odkazů/referencí/využití
  - Jakmile klesne na nulu, paměť uvolní

Počet odkazů

- Zvýšení
  - Vytvoření — a = ‘Ahoj’
  - Vytvoření aliasu — b = a
  - Předání funkci — zpracuj(a)
  - Stane se součástí složeného prvku — x = [a, ‘B’, ‘C’]
- Snížení
  - Ukončení „scope“ proměnné — konec funkce
  - Smazání proměnné — del a
  - Aliasu je přiřazena jiná hodnota — b = ‘Nazdar’
  - Odstranění ze složeného prvku — x.remove(a)
  - Odstranění složeného prvku — del x

Příklad

```
>>> import sys
>>> a=1001
>>> print sys.getrefcount(a)
2
>>> b=a
>>> print sys.getrefcount(a)
3
>>> c=[a,1]
>>> print sys.getrefcount(a)
4
>>>
```

Často používané hodnoty

- Často používané hodnoty Python vytváří automaticky a udržuje je stále v paměti
  - Čísla mezi -5 a +256
  - Jednotlivé znaky
  - Prázdný řetězec
- Důvodem je vyšší rychlost/výkon

```
>>> import sys
>>> a=1
>>> print sys.getrefcount(a)
531
>>> a=0
>>> print sys.getrefcount(a)
529
>>> a=2
>>> print sys.getrefcount(a)
232
```

Kopírování objektů

1. Vytvoření aliasu
   - Přímo odkazujeme na původní proměnnou
   - Příklad: b = a
2. Mělká kopie
   - Vytvoříme novou proměnou, ale odkazujeme původní prvky jako aliasy
   - Příklad: b=a[:]
- Hluboká kopie
   - Vytváříme kompletní kopii dat
   - Příklad: b=copy.deepcopy(a)

Kopírování objektů – příklad

```
>>> a=[1,2,3]
>>> b=a
>>> b.append(0)
>>> print a,b
[1, 2, 3, 0] [1, 2, 3, 0]
>>>
>>> x=[1,[0,0,0],2]
>>> y=x[:]
>>> y[1][0]=9
>>> print x,y
[1, [9, 0, 0], 2] [1, [9, 0, 0], 2]
>>> y[0]=8
>>> print x,y
[1, [9, 0, 0], 2] [8, [9, 0, 0], 2]
>>>
```

```
>>> from copy import deepcopy
>>> r=[1,[2,3,4],5]
>>> s=deepcopy(r)
>>> s[0]=99
>>> s[1][0]=88
>>> print r,s
[1, [2, 3, 4], 5] [99, [88, 3, 4], 5]
>>>
```

Kopírování objektů – příklad druhý

```
a = range(5)
b = [ a for i in range(5) ]
print b
b[1][1] = 88
print b
```

```
>>>
[[0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4]]
[[0, 88, 2, 3, 4], [0, 88, 2, 3, 4], [0, 88, 2, 3, 4], [0, 88, 2, 3, 4], [0, 88, 2, 3, 4]]
>>>
```

Python Tutor

PythonTutor

- www.pythontutor.com
- Vizualizace běhu programu v pythonu
- Krokování
- Vhodné pro:
  - Pochopení práce s proměnnými/pamětí
  - Ladění programu

PythonTutor - ukázka

```
 1  x = [1, 2, 3]
 2  y = [4, 5, 6]
 3  z = y
 4  y = x
 5  x = z
 6
 7  x = [1, 2, 3] # a different [1, 2, 3] list!
 8  y = x
 9  x.append(4)
10  y.append(5)
11  z = [1, 2, 3, 4, 5] # a different list!
12  x.append(6)
13  y.append(7)
14  y = "hello"
15
16
17  def foo(lst):
18      lst.append("hello")
19      bar(lst)
```

Edit code

<< First | < Back | Step 7 of 27 | Forward > | Last >>