JEŠTĚ K FUNKCÍM...

Funkce je objekt

...takže se dá dosadit do proměnné

Funkce CHYBA

```
tisk = print
tisk(f"{2}*{5}={2*5}")
```

...nebo předat jako parametr

```
def tabulka(f, jmeno, od, do):
    print( f"Tabulka hodnot funkce {jmeno} pro hodnoty {od}..{do}")
    print("---------------")
    for x in range(od,do+1):
        print(f"{x}:{f(x):5.2f}")
    print("---------------")

import math
tabulka( math.sin, "sin", 0,10 )
tabulka( math.cos, "cos", 0,10 )
tabulka( math.sqrt, "sqrt", 0,10 )
```

Funkce je objekt

...nebo dosadit do seznamu

```
for funkce ,jmeno in [(math.sin,"sin"), (math.cos,"cos"),
                      (math.sqrt,"odmocnina")]:
    tabulka( funkce, jmeno, 0,10 )
```

...nebo do slovníku

```
slovnik = {"sin":math.sin, "cos":math.cos, "odmocnina":math.sqrt}

jmeno = input()
tabulka( slovnik[jmeno], jmeno, 0,10 )
```

Lambda funkce

lambda calculus, Alonzo Church, 1930

je to jediný výraz
nemůžou obsahovat příkazy*)

známy též jako: anonymní funkce nebo lambda-výrazy

```
tabulka( lambda x: 1/x, "prevracena hodnota", 1,10 )
tabulka( lambda x: x*x, "ctverec", 1,10 )

pricti1 = lambda x:x+1
print( pricti1(27) )
```

Lambda funkce [2]

použití: funkce vyžadované jako parametr:

```
lidi = [ ['Bořík','Weber'],['Anna','Zelená'],
           ['Cyril','Sláma'],['David','Hroch']]

sorted( lidi )
sorted( lidi, key = lambda x: x[1]+x[0] )
```

Closure

uvnitř funkce můžeme definovat jinou funkci...

```
def f(x):

    def g():
        return 7

    return x+g()
```

...ta (vnitřní) funkce (g) není zvenku viditelná…

Closure

...ale můžeme ji dosadit do globální proměnné nebo vrátit!

```
def f(x):
    def g():
        return 7
    global funkce_g
    funkce_g = g  # varianta A)
    return g      # varianta B)

f(5)
print( funkce_g() ) # varianta A)
print( f(1)() )     # varianta B)
```

Closure

Problém: Ta vnitřní funkce vidí
proměnné a parametry té vnější funkce.
Uvidí na ně, i když ta vnější funkce už skončila?

```
def Pricti(kolik):
    def g(x):
        return x+kolik
    return g

pricti_1 = Pricti(1)
pricti_5 = Pricti(5)

print( pricti_1(10) )
print( pricti_5(10) )
```

Generátor

Fukce, která „postupně vrací výsledek".

Například kdybychom chtěli zpracovat veliký soubor:

(zdroj: https://realpython.com/introduction-to-python-generators/)

```
radky = PrectiSoubor("...")
for r in radky:
    print( r ) # ZpracujRadek( r )
```

V čem je problém?

Generátor

Obyčejná funkce:

```
def PrectiSoubor(jmeno):
    f = open(jmeno,'r')
    s = f.read().split('\n')
    f.close()
    return s
```

Generátor:

```
def PrectiSoubor(jmeno):
    for s in open(jmeno,'r'):
        yield s[:-1]
```

Generátor

Jak to funguje:

- první yield vrátí objekt třídy generator
- ten má metodu __next__(),
  která na každé zavolání vrátí další hodnotu
  (dá se volat via next(g))
- stav funkce se uloží, včetně jejích proměnných,
  ukazatele instrukcí, zásobníku a obsluhy výjimek
- ...takže při zavolání další metody
  může být funkce obnovená

Generátor

```
def gen():
    for i in range(10):
        yield i


it = gen()
print( it, type(it) )
while True:
    # print( it.__next__() )
    print( next( it ) )
```

Generátor

Jak ukončit generátor: metoda .close()

```
def vsechna_cisla():
    i = 0
    while True:
        yield i
        i += 1

vc = vsechna_cisla()
for i in vc:
    print(i)
    if i==17:
        vc.close()
```

Generator expression

Způsob, jak rychle vyrobit generátor:

```
radky = (line for line in open("..."))
```

a potom:

```
while True:
    print( next(radky) )
```

nebo:

```
for r in radky:
    print( r )
```

Jak funguje for-cyklus

Pro daný objekt (iterable) vyrobí iterátor.

```
it1 = iter("abcd")
it2 = iter([1,2,3,4])

print( it1 )
   <str_iterator object at 0x0000025D14662380>
print( it2 )
   <list_iterator object at 0x0000025D14662320>
print( next( it1 ) )
   'a'
print( next( it1 ) )
   'b'
```

...a pak na něj volá funkci next().

Generátor vytváří instanci iterátoru, je to odvozený typ.

CONTEXT MANAGER

Obvyklý problém programování...

...správa externích zdrojů: databáze, soubory…

Dvě možnosti řešení:

1) try .. finally

2) with

Příkaz with
Context management protocol

```
with open("soubor.txt",'r') as f:
    while True:
        print( f.read(1),end='' )
```

Metoda .__enter__() se zavolá na začátku with-bloku.

Metoda .__exit__() se zavolá při opuštění with-bloku
a to i když blok opouštíme výjimkou!

=> U souboru ve with se takhle vždycky zavolá .close().

Více context managerů

```
with open("220102_funkce.py",'r') as f,\ # pokračovat
     open("kopie.txt",'w') as g:
    g.write( f.read() )
```

Díky with se na konci oba soubory zavřou.

Vlastní context manager

```
class MujCM:
    def __enter__(self):
        print("Začátek")
        return "Hello, World!"
    def __exit__(self, exc_type, exc_value, exc_tb):
        print("Konec")
        print(exc_type, exc_value, exc_tb, sep="\n")

with MujCM():
    print("...prikaz")
with MujCM():
    i = 1/0
```

Vlastní context manager – výjimky

```
    def __exit__(self, exc_type, exc_value, exc_tb):
        print("Konec")
        if isinstance(exc_value, ZeroDivisionError):
            # Handle IndexError here...
            print(f"Vyjimka: {exc_type}")
            print(f"Zprava: {exc_value}")
            return True
```

Když vrátí True, tak skryje výjimku
a program pokračuje za with-blokem.

PŘÍKLADY...