Obsah dnesnej prednasky

1. Rozsah platnosti premennej

2. Vnorena funkcia

3. Closure

4. Decorator

Closure - Uzaver

- Funkcia, ktora pouziva neglobalnu premennu definovanu mimo svojho tela (vysvetlim)
- Da sa to pouzit napriklad na asynchronne programovanie pomocou callbackov (spatne volanie?).
- prevzane z: Ramalho, Luciano. *Fluent Python.* O'Reilly Media, 2015.

Rozsah platnosti premennych

In [ ]:
```
def f1(a):
    print(a)
    print(b)

f1(3)
```

In [ ]:
```
b = 6
f1(3)
```

Funkcia vie citat premennu definovanu mimo svojho tela

In [ ]:
```
b = 6

def f1(a):
    print(a)
    print(b)

f1(3)
```

In [ ]:
```
b = 6
def f2(a):
    print(a)
    print(b)
    b = 9

f2(3)
```

In [ ]:
```
b = 6
def f2(a):
    print(a)
    print(b) # tuna nevypisujeme obsah premennej z prveho riadku ale tej, ktora je az na dalsom riadku. To logicky nemoze prejst
    b = 9 # akonahle raz vo funkcii priradujete do premennej, tak sa vytvori nova, lokalna pri kompilacii do bitekodu.

f2(3)
```

Python nedovoli menit (len citat) hodnotu premennej, ktora nie je lokalna

Python nevyzaduje deklaraciu premennych ale predpoklada, ze premenna priradena v tele funkcie je lokalna.

JavaScripte napriklad vyzaduje deklaraciu lokalnych premennych pomocou `var`. Ak na to zabudente, tak nic nepredpokalda a pokusi sa hladat premennu medzi globalnymi. Toto casto sposobuje bugy.

premennu medzi globalnymi. Toto casto sposobuje bugy.

Ak chcete vo funkcii priradit hodnotu premennej definovanej mimo nej, musite z nej spravit globalnu premennu

In [ ]:
```
b = 6
def f3(a):
    global b
    print(a)
    print(b)
    b = 9

f3(3)
print(b)
```

Ale my globalne premenne nemame radi. Takze toto nebudeme pouzivat

Co si zapamatat o rozsahu platnosti premennych?

- funkcia vie citat premenne definovane mimo svojho tela
- tieto premenne ale nevie menit.
- ak by ich chcela menit, tak pri kompilacii do bitekodu sa vytvori lokalna premenna a ak ju pouzijeme skor ako jej priradime hodnotu, tak mame problem
- teoreticky mozeme pouzit globalnu premennu, ale my nechceme

Dalsia vec, ktoru si potrebujeme vysvetlit na to aby sme spravili closure su vnorene funkcie

Ano, funkcie sa daju definovat vo vnutri inej funkcie

In [ ]:
```
def outer():
    def inner(a):
        return a + 7
    return inner
```

Vnutorna funkcia nie je z vonka dostupna. Existuje len v ramci tela vonkajsej funkcie. Teda pokial ju nevratime ako navratovu hodnotu.

In [ ]:
```
def outer():
    def inner(a):
        return a + 7
    return inner
```

In [ ]:
```
inner(3) # zvonka funkcia nieje dostupna a je teda chranena (nikto k nej nemoze a nezaspini nam priestor m
ien)
```

In [ ]:
```
pom = outer()
pom
```

Na co su vnorene funkcie dobre?

- Hlavne skryvaju implementaciu funkcie pred okolim
- Umoznuju definovanie komplexnej logiky a mnozstva pomocnych funkcii zatial co zvonka bude dostupna len jedina.
- Nemusite tak definovat zbytocne vela "privatnych" funkcii
- V pythne ani privatne funkcie niesu, takze toto je jediny sposob ako skryt pomocne funkcie pred svetom

Napriklad ak sa vam opakuje rovnaka postupnost riadkov, tak ju vyjmete

do funkcie ale ak je tato zbytocna pre ine funkie, tak zvonka vobec nemusi byt dostupna

tj. kalsicky princip privatnej pomocnej funkcie

In [ ]:
```
def process(file_name):
    def do_stuff(file_process):
        for line in file_process:
            print(line)

    if isinstance(file_name, str):
        with open(file_name, 'r') as f:
            do_stuff(f)
    else:
        do_stuff(file_name)
```

Daju sa pouzit napriklad na kontrolu vstupov odelenu od samotenho vypoctu

In [ ]:
```
# https://realpython.com/blog/python/inner-functions-what-are-they-good-for/
def factorial(number):

    # error handling
    if not isinstance(number, int):
        raise TypeError("Sorry. 'number' must be an integer.")
    if not number >= 0:
        raise ValueError("Sorry. 'number' must be zero or positive.")

    # logika spracovania je pekne sustredena na jednom mieste
    def inner_factorial(number):
        if number <= 1:
            return 1
        return number*inner_factorial(number-1)
    return inner_factorial(number)

# call the outer function
print(factorial(4))
```

Takze co je to teda ten uzaver?

Funkcia, ktora pouziva neglobalnu premennu definovanu mimo svojho tela

Naco?

1. Specializovanie vseobecnej funkcie (Partial function application)
2. Udrziavanie stavu medzi volaniami funkcie

> Ano, bude to spinava funkcia
>
> Obcas sa ale uchovavaniu stavu nevyhneme
>
> Ideme spravit funkcionalny sposob ako si uchovavat stav
>
> Chcem ukazat ako sa daju funkcionalne crty pouzit na vylepsenie imperativneho kodu

Specializovanie vseobecnej funkcie (Partial function application)

Teraz len jeden priklad. Poriadne sa tomu budem venovat zajtra

In [ ]:
```
def make_power(a):
    def power(b):
        return b ** a
    return power
```

In [ ]:
```
square = make_power(2)
square(3)
```

Udrziavanie stavu medzi volaniami funkcie - Ako?

- Funkcia vracajuca vnorenu funkciu, ktora vyuziva lokalnu premennu na udrziavanie stavu.
- Predstavte si takuto ulohu:

> Chceme funkciu, ktora bude pocitat priemer stale rastuceho poctu cisel.
>
> Poziadavka je aby sme to vedeli spravit v jednom prechode cez data nad potencialne nekonecnou sekvenciou dat.
>
> > Jedna moznost je generator, druha je uzaver

Chceme funkciu, ktoru budeme opakovane volat s dalsim a dalsim cislom a vracat nam to bude vzdy aktualny priemer vsetkych doterajsich cisel

In [ ]:
```
# zatial nespustat. avg nieje definovane
avg(10)
# 10.0
avg(11)
# 10.5
avg(12)
# 11
```

Na vypocet potrebujeme uchovavat stav: sumu a pocet hodnot. Kde sa uklada stav?

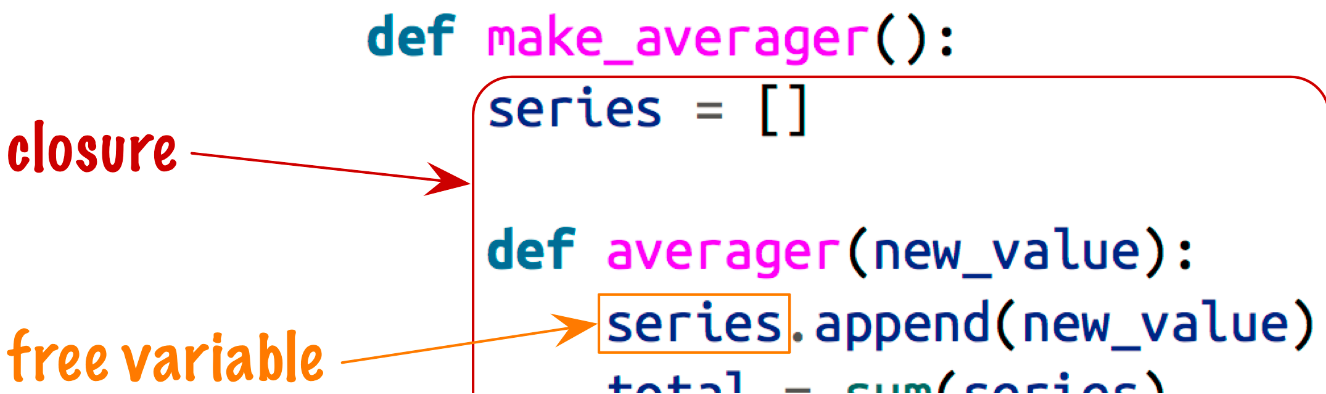
- V globalnej premennej? Nie, nechceme si zapratat priestor mien nejakymi nahodnymi premennymi, ktore by nam mohol hocikto prepisat.
- Potrebuje nieco, co nam tie premenne schova.

Nieco taketo by sa dalo implementovat pomocou uzaveru

In [ ]:
```
def make_averager():
    series = [] # tato premenna je platna len vo funkcii make_averager. Je pre nu lokalna. Mimo nej neexis
tuje.
    def averager(new_value):
        series.append(new_value) # vieme pristupit k premennej definovanej vyssie.
        # Kedze list je mutable, tak ho vieme aj zmenit. Pozor, nemenime premennu, menime objekt!
        # Aby sme menili premennu, tak by tu muselo byt =
        total = sum(series)
        return total/len(series)
    return averager
```

In [ ]:
```
avg = make_averager()
print(avg(10))
print(avg(11))
print(avg(12))
```

Obalujuca funkcia definuje rozsah platnosti lokalnych premennych. Po skonceni vykonavania uz na ne neexistuje referencia. Okrem tej, ktora sa vrati ako navratova hodnota. Co je zhodou okolnosti funkcia, ktora jednu z lokalnych premennych pouziva. Tato premenna sa vola **volna premenna**, kedze na nu neexistuje ziadna ina referencia.



```
    total = sum(series)
    return total/len(series)

return averager
```

Aj ked k volnej premennej sa este da dostat

Kedze v pythone nie je slovicko private a vsetka kontrola pristupu je len na konvecnii, tak by ste sa toho nemali chytat. Na debugovanie a testovanie je to ale celkom dobre vediet.

In [ ]:
```
avg
# je to funkcia, ktora je definovana vo funkcii make_averager ako lokalna premenna s nazvom averager
```

Da sa pristupit k nazvom premennych a aj volnych premennych

In [ ]:
```
print(avg.__code__.co_varnames)
print(avg.__code__.co_freevars)
```

A aj k ich hodnotam

In [ ]:
```
print(avg.__closure__)
print(avg.__closure__[0].cell_contents) # tato hodnota sa da aj zmenit, ale nerobte to.
```

Na spocitanie priemeru nepotrebujeme cely zoznam. Staci nam suma a pocet.

Tento priklad je ale pokazeny. Kto vie preco? Uz som to naznacil viac krat.

In [ ]:
```
def make_averager():
    count = 0
    total = 0
    def averager(new_value):
        count += 1
        total += new_value
        return total / count
    return averager
```

In [ ]:
```
avg = make_averager()
avg(10)
```

+= je vlastne priradenie a teda spravi z premennych count a total lokalne premenne, pricom ich chce hned predtym aj pouzit

Pripominam, ze predchadzajuci priklad nepriradzoval do premennej, len upravoval mutable objekt.

Podobne ako uplne na zaciatku bol riesenim prikaz `global`, teraz to bude `nonlocal`

In [ ]:
```
def make_averager():
    count = 0
    total = 0
    def averager(new_value):
        nonlocal count, total # tieto dve premenne teda nebudu lokalne v ramci funkcie averager
        # ale sa zoberu z funkcie o uroven vyssie
        count += 1
        total += new_value
        return total / count
```

```
    return averager
```

In [ ]:
```
avg = make_averager()
avg(10)
```

Minule som vam slubil vysvetlenie ako opravit jeden hack

Mali sme kod, ktory meral mnozstvo pamati spotrebovanej pri pocitani s generatorom a bez neho Funkcia measure_add mala vedlajsi efekt, ktory vypisoval spotrebu pamati kazdych 200 000 volani. Potrebovala teda pocitadlo, ktore si uchovavalo stav medzi volaniami. Pouzili sme mutable objekt na to aby sme nemuseli pouzit priradovanie a nesnazili sa pristupit k premennej pred jej definicou alebo aby sme nedefinovali globalnu premennu..

In [ ]:
```
from functools import reduce
import gc
import os
import psutil
process = psutil.Process(os.getpid())

def print_memory_usage():
    print(process.memory_info().rss)

counter = [0] # Toto je ta globalny mutable objekt
def measure_add(a, result, counter=counter):
    if counter[0] % 200000 == 0:
        print_memory_usage()
    counter[0] = counter[0] + 1
    return a + result
```

In [ ]:
```
gc.collect()
counter[0] = 0
print_memory_usage()
print(reduce(measure_add, [x*x for x in range(1000000)]))
```

Ako tento hack opravit?

1. zabalime to cele do funkcie
2. zmenime mutable objekt za immutable
3. definujeme nonlokalnu premennu
4. vratime vnutornu funkciu

In [ ]:
```
counter = [0] # Toto je ta globalny mutable objekt
def measure_add(a, result, counter=counter):
    if counter[0] % 200000 == 0:
        print_memory_usage()
    counter[0] = counter[0] + 1
    return a + result
```

In [ ]:
```
# toto tu mam en pre kontrolu, aby som nespravil chybu
def make_adder():
    counter = 0
    def adder(a, result):
        nonlocal counter
        if counter % 2000000 == 0:
            print_memory_usage()
        counter += 1
        return a+result
    return adder
```

A teraz to vyskusame

In [ ]:
```
measure_add = make_adder()
gc.collect()
print_memory_usage()
print(reduce(measure_add, [x*x for x in range(10000000)]))
```

Pomocou Closure sa da vytvorit napriklad aj jednoducha trieda

In [ ]:
```python
class A:
    def __init__(self, x):
        self._x = x

    def incr(self):
        self._x += 1
        return self._x

obj = A(0)
```

In [ ]:
```python
obj.incr()
```

Pomocou closure napriklad takto

In [ ]:
```python
def A(x):
    def incr():
        nonlocal x
        x +=1
        return x
    return incr

obj = A(0)
```

In [ ]:
```python
obj()
```

Ale mohol by som vracat aj viac "metod"

In [ ]:
```python
def A(x):
    def incr():
        nonlocal x
        x +=1
        return x

    def twice():
        incr()
        incr()
        return x
    return {'incr': incr, 'twice':twice}
obj = A(0)
```

In [ ]:
```python
obj['incr']()
# obj['twice']()
```

A aby bolo to volanie krajsie, tak mozem spravit nieco taketo

dalo by sa to aj krajsie, ale bol som lenivy :)

In [ ]:
```python
import pyrsistent as ps
def A(x):
    def incr():
        nonlocal x
        x +=1
        return x

    def twice():
        incr()
        incr()
        return x

    return ps.freeze({'incr': incr, 'twice':twice})
obj = A(0)
```

In [ ]:
```python
# obj.incr()
obj.twice()
```

V skutocnosti su uzaver a objekt vytvoreny z triedy ekvivalentne

[http://c2.com/cgi/wiki?ClosuresAndObjectsAreEquivalent (http://c2.com/cgi/wiki?ClosuresAndObjectsAreEquivalent)](http://c2.com/cgi/wiki?ClosuresAndObjectsAreEquivalent)

The venerable master Qc Na was walking with his student, Anton. Hoping to prompt the master into a discussion, Anton said "Master, I have heard that objects are a very good thing - is this true?" Qc Na looked pityingly at his student and replied, "Foolish pupil - objects are merely a poor man's closures."

Chastised, Anton took his leave from his master and returned to his cell, intent on studying closures. He carefully read the entire "Lambda: The Ultimate..." series of papers and its cousins, and implemented a small Scheme interpreter with a closure-based object system. He learned much, and looked forward to informing his master of his progress.

On his next walk with Qc Na, Anton attempted to impress his master by saying "Master, I have diligently studied the matter, and now understand that objects are truly a poor man's closures." Qc Na responded by hitting Anton with his stick, saying "When will you learn? Closures are a poor man's object." At that moment, Anton became enlightened.

Aky-taky preklad

Ctihodný majster Qc Na šiel so svojím študentom, Antonom. Dúfajúc, že vyzve majstra do diskusie, Anton povedal: "Pane, počul som, že objekty sú veľmi dobrá vec - je to pravda?" Qc Na pozrel súcitne na svojho študenta a odpovedal: "pochabý žiak - objekty sú len chudákove uzávery."

Pokarhaný Anton odišiel od svojho majstra a vrátil sa do svojej cely, študovať uzávery. Starostlivo si prečítal celú "Lambda: The Ultimate ..." sériu článkov spolu s referenciami, a implementoval malý Scheme interpret s objektovým modelom založeným na uzáveroch. Naučil sa veľa, a tešil sa na to ako bude informovať svojho majstra o svojom pokroku.

Na jeho ďalšej ceste s Qc Na, sa Anton pokúšal zapôsobiť na svojho pána tým, že hovorí: "Majstre, usilovne som študoval a pochopil som, že objekty sú skutočne chudákove uzávery." Qc Na reagoval tým, že udrel Antona palicou. Hovorí: "Kedy sa poučíš? Uzávery sú objektami chudáka." V tej chvíli Anton dosiahol osvietenie.

Objekty a uzavery maju rovnaku vyjadrovaciu silu

Dolezite je vybrat si ktore pouzit v ktorej situacii, tak aby ste vyuzili pekne vlastnosti oboch.

Ak toto dokazete, tak ste sa stali skutocnymi odbornikmi a dosiahnete nirvanu :)

Decorator

Decorator (Python) vs. navrhovy vzor Dekorator

Navrhovy vzor

- Ciel: pridanie dodatocnej zodpovednosti objektu dynamicky. Za behu.
- Ak mame velmi vela moznych kombinacii rozsireni nejakeho objektu
- Priklad s kaviarnou (https://python-3-patterns-idioms-test.readthedocs.io/en/latest/Decorator.html) (Priklad je v pythone ale nepouziva konstrukciu decorator. Je to len implementacia anvrhoveho vzoru.)
- Zabalenie objektu do noveho objektu, kde ten stary je parametrom konstruktora a pri volani hociakej metody sa vola aj metoda obalovaneho objektu. Piklad v Jave (https://sourcemaking.com/design_patterns/decorator/java/3)
- Obalujuci objekt sa puziva namiesto obalovaneho

Konstrukcia v pythone

- Obalenie funkcie alebo triedy do vlastneho kodu
- Pouziva sa pri definicii metody alebo triedy
- Nie priamo urcene na individualne objekty
- Pomocou tejto konstrukcie by sa dal implementovat navrhovy vzor Dekorator, ale to by bolo len velmi obmedzene vyuzitie.

inspirovane - http://www.artima.com/weblogs/viewpost.jsp?thread=240808 (http://www.artima.com/weblogs/viewpost.jsp?thread=240808)

Co je dekorator konstrukcia v pythone

- Zabalenie funkcie do nejakej inej a pouzivanie obalenej namiesto povodnej
  - neznie vam to podobne ako aspektovo orientovane programovanie v jave? Je to podobne, ale jednoduchsie (pouzitim a aj moznostami)
- Moze byt implementovany hociakym zavolatelnym objektom (funkcia alebo objekt triedy implementujucej metodu `__call__`)
- Decorator moze byt hociaka funkcia, ktora prijma inu funkciu ako parameter a vracia funkciu

In [ ]:
```
# takyto dekorator s tou obalovanou funkciou vlastne nic nespravi
def najjednoduchsi_mozny_dekorator(param_fct):
    return param_fct # vsimnite si, ze tu niesu zatvorky. Cize sa vracia funkcia ako objekt a nevykonava s
a
```

Ked uz mame tu funkciu, tak s nou mozeme nieco aj spravit - napriklad obalit niecim inym

In [ ]:
```
def zaujimavejsi_dekorator(param_fct):
    def inner():
        do_stuff()
        result = param_fct()
        do_another_stuff()
        return result
    return inner
```

Alebo nahradit niecim uplne inym

In [ ]:
```
def nahradzujuci_dekorator(param_fct):
    def nieco_uplne_ine():
        pass
    return nieco_uplne_ine
```

stale plati to, ze je to funkcia, ktora dostava ako parameter funkciu a vracia funkciu

Ako potom takyto dekorator pouzit?

In [ ]:
```
def function(): # funkcia, ktoru chceme dekorovat
    pass

function = decorator(function)
```

In [ ]:
```
# syntakticky cukor
@decorator
def function():
    pass
```

Dekorovana funkcia sa pouziva namiesto povodnej

In [ ]:
```
def deco(func):
    def inner():
        print('running inner()')
    return inner
```

In [ ]:
```
@deco
def target():
    print('running target()')
```

In [ ]:
```
target()
target
```

Dekorator je spusteny pri importovani ale dekorovana funkcia az po explicitnom zavolani

In [ ]:
```
registry = []

def register(func):
    print('running register(%s)' % func) # nejaky kod sa vykona pri registrovani
    registry.append(func)
    return func # vracia sa ta ista funkcia bezo zmeny
```

In [ ]:
```
@register
def f1():
```

```
    print('running f1()')

@register
def f2():
    print('running f2()')

def f3():
    print('running f3()')
```

In [ ]:
```
registry
```

Co dava zmysel ak sa vlastne deje toto

In [ ]:
```
def f1():
    print('running f1()')

f1 = register(f1)

def f2():
    print('running f2()')

f2 = register(f2)

def f3():
    print('running f3()')
```

In [ ]:
```
f1()
f2()
f3()
```

Co ked ma dekorovana funkcia nejake parametre?

In [ ]:
```
def my_print(string):
    print(string)
```

In [ ]:
```
def param_decorator(param_fct):
    def wrapper(string): # wrapper musi mat tie iste parametre
        print('wrapper stuff')
        return param_fct(string)
    return wrapper
```

In [ ]:
```
@param_decorator # pri pouzivani dekoratora sa potom nic nemeni
def my_print(string):
    print(string)

my_print('hello')
```

Co ked ma tych parametrov viac?

To iste ako v predchadzajucom pripade. Wrapper musi mat tie iste parametre.

Skusme to zovseobecnit pre funkcie s hociakym poctom atributov

In [ ]:
```
def param_decorator2(param_fct):
    def wrapper(*args): # wrapper musi mat tie iste parametre
        print('wrapper stuff')
        return param_fct(*args) # co sa stane ked tu nebude *?
    return wrapper
```

In [ ]:
```
@param_decorator2
def my_print(string):
    print(string)

@param_decorator2
def my_print_more(string1, string2, string3):
    print(string1, string2, string3)

@param_decorator2
```

```
def my_print_many(*args):
    print(*args)

my_print('hello')
my_print_more('hello', 'hello2', 'hello3')
my_print_many('hello', 'hello2', 'hello3', 'hello4', 'hello5')
```

No a co pomenovane atributy?

In [ ]:
```
def my_print_optional(first, second='second', third='third'):
    print(first, second, third)

my_print_optional('1', '2', '3')
my_print_optional('1', '2')
my_print_optional('1')
my_print_optional('1', third='3', second='2')
my_print_optional('1', third='3')
```

In [ ]:
```
def param_decorator3(param_fct):
    def wrapper(*args, **kwargs): # wrapper musi mat tie iste parametre
        print('wrapper stuff')
        return param_fct(*args, **kwargs)
    return wrapper

@param_decorator3
def my_print_optional(first, second='second', third='third'):
    print(first, second, third)
```

In [ ]:
```
my_print_optional('1', '2', '3')
my_print_optional('1', '2')
my_print_optional('1')
my_print_optional('1', third='3', second='2')
my_print_optional('1', third='3')
```

Teraz si mozeme vyrobit napriklad uplne vseobecny dekorator, ktory bude pocitat pocty volani nejakej funkcie

In [ ]:
```
def counter_decorator(fct):
    counter = 0
    def wrapper(*args, **kwargs):
        nonlocal counter
        counter += 1
        return fct(*args, **kwargs)
    return wrapper
```

In [ ]:
```
@counter_decorator
def counted_fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return counted_fib(n-1) + counted_fib(n-2)

counted_fib(10)
print(counted_fib.__closure__[0].cell_contents)
```

dalo by sa to este vylepsit tak, aby som mal praktickejsi pristup k tomu pocitadlu, ale nateraz mi to staci

Pocita sa celkovy pocet volani funkcie

In [ ]:
```
counted_fib(5)
print(counted_fib.__closure__[0].cell_contents)
counted_fib(5)
print(counted_fib.__closure__[0].cell_contents)
```

Dekorator sa da pouzit na Memoizaciu (Memoization)

- Ak mame ciste funkcie, tak ich vystup zalezi len od vstupov.
- Ak mam dve volania funkcie s rovnakymi atributmi, tak to druhe viem nahradit predchadzajucou hodnotou bez toho, aby som realne spustal vypocet.
- Dostal by som teda cachovanie funkcii
- Dekorator sa da presne na toto pouzit

In [ ]:
```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

fib(10)
```

Potrebujeme nejaku strukturu, kde si budeme ukladat priebezne vysledky

- Napriklad slovnik

In [ ]:
```
def memoize(f):
    memo = {}
    counter = 0
    def wrapper(x):
        if x not in memo:
            nonlocal counter # toto by tu nemuselo byt, ale ja chcem vediet kolko som si usetril volani
            memo[x] = f(x)
            counter += 1 # toto by tu nemuselo byt
        return memo[x]
    return wrapper
```

In [ ]:
```
@memoize
def memoized_fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return memoized_fib(n-1) + memoized_fib(n-2)

memoized_fib(10)
print(memoized_fib.__closure__[0].cell_contents)
```

In [ ]:
```
@counter_decorator
def counted_fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return counted_fib(n-1) + counted_fib(n-2)

counted_fib(10)
print(counted_fib.__closure__[0].cell_contents)
```

Toto bola len jednoducha verzia, na memoizovanie funkcie s jednym prametrom.

- Rozsirenie na viacero atributov by malo byt pre vas jednoduche
- Rozsirenie na pomenovane atributy uz take jednoduche nie je pretoze `**kwargs` je slovnik a ten nie je hashovatelny

> kto vie preco nieje hashovatelny?

- Podobne to **nebude** fungovat ak by hociktory z parametrov nebol hashovatelny.

Odpoved na predchadzajucu otazku

na to aby mohol byt objekt hashovatelny, musi byt nemenny. Pri zmene objektu by sa totiz musel zmenut vysledok hashovacej funkcie (vysledok by

na to aby mohol byt objekt hashovatelny, musi byt nemenny. Pri zmene objektu by sa totiz musel zmenit vysledok hashovacej funkcie (vysledok by mal zavisiet od obsahu obejtku) a teda uplne straca svoj zmysel pri identifikacii objektu

Co ked chcem dat dekoratoru nejake parametre?

Tu je syntax trochu nestastna a musim to zabalit este do jednej funkcie

In [ ]:
```
def decorator(argument): # tuto jednu funkciu som tam pridal
    def real_decorator(param_funct): # tu bu zacinal dekorator bez parametrov
        def wrapper(*args, **kwargs):
            before_stuff()
            something_with_argument(argument)
            funct(*args, **kwargs)
            after_stuff()
        return wrapper
     return real_decorator

@decorator(argument_value)
def my_fct():
    pass
```

Pouzit to viem napriklad na vytvorenie dekoratora, ktory mi bude logovat volania funkcie a ja si zvolim uroven logovania

In [ ]:
```
def log(level, message):
    print("{0}: {1}".format(level, message))

def log_decorator(level):
    def decorator(f):
        def wrapper(*args, **kwargs):
            log(level, "Function {} started.".format(f.__name__))
            result = f(*args, **kwargs)
            log(level, "Function {} finished.".format(f.__name__))
            return result
        return wrapper
    return decorator

@log_decorator('debug')
def my_print(*args):
    print(*args)

my_print('ide to?')
```

Dekorovanim menim niektore atributy funkcie

In [ ]:
```
def some_fct():
    """doc string of some_fct"""
    print("some stuff")

some_fct()
print(some_fct.__name__)
print(some_fct.__doc__)
print(some_fct.__module__)
```

In [ ]:
```
def decorator(f):
    def wrapper_fct(*args, **kwargs):
        """wrapper_fct doc string"""
        return f(*args, **kwargs)
    return wrapper_fct

@decorator
def some_fct():
    """doc string of some_fct"""
    print("some stuff")
```

In [ ]:
```
some_fct()
print(some_fct.__name__)
print(some_fct.__doc__)
print(some_fct.__module__)
```

Ak dekorujem funkciu, tak nova funkcia dostane `__name__`, `__doc__`, `__module__` atributy z dekoratora a nie z tej povodnej funkcie.

`__module__` sa nezmeni, kedze dekorator je definovany v tom istom module, ak by som ho ale importoval ako balicek, tak by sa zmenilo aj to

Nastastie na to mame riesenie - dalsi dekorator

tento nastastie ale staci importovat a takmer nijak to nekomplikuje nas povodny kod

In [ ]:
```
from functools import wraps

def decorator(f):
    @wraps(f) #mame na to dekorator, ktory tieto atributy skopiruje
    def wrapper_fct(*args, **kwargs):
        """wrapper_fct doc string"""
        return f(*args, **kwargs)
    return wrapper_fct

@decorator
def some_fct():
    """doc string of some_fct"""
    print("some stuff")
```

In [ ]:
```
some_fct()
print(some_fct.__name__)
print(some_fct.__doc__)
print(some_fct.__module__)
```

Sumarizujeme - Rozne mozne formy dekoratorov

Nahradzajuci generator nahradi funkciu uplne niecim inym

In [ ]:
```
def nahradzujuci_dekorator(param_fct):
    def nieco_uplne_ine():
        pass
    return nieco_uplne_ine
```

Obalujuci dekorator prida nieco pred a/alebo za volanie funkcie

In [ ]:
```
def obalujuci_dekorator(param_fct):
    def inner():
        before_call()
        result = param_fct()
        after_call()
        return result
    return inner
```

Dekorator uchovavajuci si stav

In [ ]:
```
def obalujuci_dekorator(param_fct):
    stav = hodnota
    def inner():
        nonlocal stav # ak mame mutable objekt ako stav, tak netreba pouzivat nonlocal
        stav = ina_hodnota
        return param_fct()
    return inner
```

Parametrizovany dekorator

In [ ]:
```
def vonkajsi_decorator(argument):
    def decorator(param_funct):
        def fct_wrapper(*args, **kwargs):
            before_stuff()
            something_with_argument(argument)
```

```
            something_with_argument(argument)
            funct(*args, **kwargs)
            after_stuff()
        return fct_wrapper
     return decorator

@vonkajsi_dekorator(parameter)
def funkcia():
    pass
```

Registracny dekorator vykona nieco pri registracii funkcie

- vykona nieco pri registracii funkcie v case importovania a nie vykonavania samotenej funkcie.
- kludne si moze dekorator udrzovat nejaky stav pomocou lokalnych premennych

In [ ]:
```
def registracny_dekorator(param_func):
    when_registering_stuff()
```