1. Partial function application

2. Pattern matching

Ciastocna aplikacia - Partially applied functions

http://blog.dhananjaynene.com/tags/functional-programming/ (http://blog.dhananjaynene.com/tags/functional-programming/)

- Ciastocna aplikacia transformuje funkciu s nejakym poctom parametrov na inu funkciu s mensim poctom parametrov
- Cize zafixuje nejake parametre

f:(X × Y × Z) → N

partial(f):(Y × Z) → N

Vcera som naznacil, ako sa nieco taketo da spravit s pomocou uzaveru

In [ ]:
```
def add(a, b):
    return a + b
```

In [ ]:
```
def make_adder(a) :
    def adder(b) :
        return add(a, b)
    return adder
```

In [ ]:
```
add_two = make_adder(20)
add_two(4)
```

Iny priklad

In [ ]:
```
def make_power(exponent):
    def power(x):
        return x**exponent
    return power
```

In [ ]:
```
square = make_power(2)
print(square(3))
print(square(30))
square(300)
```

Balicek functools ma na to funkciu, ktora definiciu takychto funkcii robi este pohodlnejsiu

In [ ]:
```
from functools import partial

def power(base, exponent):
    return base ** exponent

cube = partial(power, 3)
cube(2)
```

In [ ]:
```
def power(base, exponent):
    return base ** exponent

cube = partial(power, exponent=3)
cube(2)
```

Iny priklad, uprveny konstruktor int

In [ ]:
```
basetwo = partial(int, base=2)
```

```
In [ ]: basetwo = partial(int, base=2)
        basetwo('111010101')
```

Problem je v tom, ze skoro vsetky priklady na internete, ktore najdete su z toho ako vyrobit power funkcie alebo nieco podobne trivialne

Skusme nieco trivialne, ale praktickejsie

Napriklad funkciu, ktora ma vypisovat do nejakeho specialneho suboru. Napriklad chyboveho vystupu

```
In [ ]: import sys
        from functools import partial

        print_stderr = partial(print, file=sys.stderr)
```

```
In [ ]: print_stderr("pokus")
```

Toto by som vedel dosiahnut aj dekoratorom, aj lambdou aj pomocou closure ale takto je to asi najjednoduchsie

Skusme si niektore z toho naprogramovat v ramci opakovania

```
In [ ]: # print_stderr = partial(print, file=sys.stderr)
        print_stderr = lambda x: print(x, file=sys.stderr)
        print_stderr('hahahhaha')
```

Skusme partial application pouzit na refaktorovanie takehoto kodu

```
In [ ]: for text in lines:
            if re.search('[a-zA-Z]\=', text):
                some_action(text)
            elif re.search('[a-zA-Z]\s\=', text):
                some_other_action(text)
            else:
                some_default_action()
```

regularne vyrazy sa daju vytiahnut do funkcie

```
In [ ]: def is_grouped_together(text): # skuste z tohoto spravit partial
            return re.search("[a-zA-Z]\s\=", text)

        def is_spaced_apart(text):
            return re.search("[a-zA-Z]\s\=", text)

        def and_so_on(text):
            return re.search("pattern_188364625", text)

        for text in lines:
            if is_grouped_together(text):
                some_action(text)
            elif is_spaced_apart(text):
                some_other_action(text)
            else:
                some_default_action()
```

Vidite tam to opakovanie kodu?

Ako by to bolo cele prerobene?

In [ ]:
```
is_spaced_apart = partial(re.search, '[a-zA-Z]\s\=')
is_grouped_together = partial(re.search, '[a-zA-Z]\=')


for text in lines:
    if is_grouped_together(text):
        some_action(text)
    elif is_spaced_apart(text):
        some_other_action(text)
    else:
        some_default_action()
```

Dalsie priklady na pouzitie partial pri refactoringu

http://chriskiehl.com/article/Cleaner-coding-through-partially-applied-functions/ (http://chriskiehl.com/article/Cleaner-coding-through-partially-applied-functions/)

A preco to nepouzit na specializovany konstruktor?

In [ ]:
```
class Tovar:
    def __init__(self, typ, mnozstvo=0):
        self.typ=typ
        self.mnozstvo=mnozstvo

    def write(self):
        return '{}: {}'.format(self.typ, self.mnozstvo)

nakup_jablk = Tovar('jablka', 3)
print(nakup_jablk.write())
```

In [ ]:
```
Jablko = partial(Tovar, 'jablka')
Jablko(4).write()
```

To iste by fungovalo aj na "objekt" vytvoreny pomocou closure

In [ ]:
```
import pyrsistent as ps

def Tovar(typ, mnozstvo):
    def write():
        return '{}: {}'.format(typ, mnozstvo)
    return ps.freeze({'write': write})
```

In [ ]:
```
Jablko = partial(Tovar, 'jablka')
Jablko(5).write()
```

Viete si tak vytvorit viacere konstruktory pre tu istu triedu

Co vam brani vytvorit si konstruktor pre nejaky specialny typ loggera alebo objektu na citanie nejakeho specialneho typu suboru.

Nemsuite stale opakovat tie iste parametre vo volani konstruktora / funkcie.

Viete to pouzit nie len na specializovanie, ale aj na oddelenie zadavania parametrov funkcie a jej vykonania v case.

Kolko krat sa vam stalo, ze ste vedeli davno v programe aku funkciu budete musiet zavolat a aj s castou argumentov, ale museli ste cakat az do nejakeho casu, kde ste dostali aj zvysok a museli ste parametre predavat spolu s funkciou / objektom na ktorom bola metoda

Ak by ste vedeli vyrobit funkciu, s niektorymi parametrami prednastavenymi, tak by vam stacilo posuvat si tuto jednu funkciu a nemuseli by ste si presuvat vsetky parametre az do miesta, kde ich nakoniec vlozite pri volani funkcie

In [ ]:
```
def query_database(userid, password, query) :
    # do query
    # return results

def bar(userid, password):
    return query_database(userid, password)
```

```
def foo(userid, password) :
    return bar(userid, password)

def main(userid, password) :
    # .. lot of code here .. eventually reaching
    foo(userid, password)
```

Takto by sa to dalo spravit ak by sme pouzili partial application pomocou vnorenej funkcie.

In [ ]:
```
def get_query_agent(userid, password)
    def do_query(query) :
        # do query
        # return results
    return do_query

def bar(querying_func):
    return func(querying_func)

def foo(querying_func) :
    return bar(querying_func)

def main(userid, password) :
    query_agent = get_query_agent(userid, password)
    # .. much further down the line
    foo(query_agent)
```

Teraz o cool funkcionalej vlastnosti, ktora v Pythone *nie je*

Pattern matching

Multimethods

Multiple dispatch

Multiple dispatch (and poor men's patter matching) in Java

http://blog.efftinge.de/2010/03/multiple-dispatch-and-poor-mens-patter.html (http://blog.efftinge.de/2010/03/multiple-dispatch-and-poor-mens-patter.html) odkaz davam hlavne kvoli nazvu clanku :)

In [ ]:
```
# -- JAVA --

static void print(Fruit f) {
 sysout("Hello Fruit");
}

static void print(Banana b) {
 sysout("Hello Banana");
}

Banana banana = new Fruit();

print(banana)
```

Toto nebol multiple dispatch. Toto bol overloading pretoze sa to rozhodovalo v case kompilacie.

preto by sa vypisalo "Hello banana" na zaklade typu premennej a nie "Hello Fruit" na zaklade typu objektu

multiple dispatch sa rozhoduje dynamicky na zaklade objektu

Multiple dispatch by som dosiahol napriklad ak by print bola metoda objektu.

Nanestastie Python nema ani multiple dispatch a ani overloading

Nanestastie Python nema ani multiple dispatch a ani overloading

Nema zmysel definovat dve funkcie s rovnakym menom

In [ ]:
```
def pokus(a):
    print('pokus1')

def pokus():
    print('pokus2')

pokus()
```

A je jedno, ci maju rovnaky pocet parametrov alebo rozny. Ani definovanie typu pomocou anotacie v pythone 3 mi nepomoze

Vzdy si len prepisem funkciu inou

Nikdy sa nerozhodne na zaklade parametrov, ktora by sa mala pouzit (tak ako je to napriklad v jave)

In [ ]:
```
def pokus(a:str, b:list):
    print('pokus1')

def pokus(b:int):
    print('pokus2')

pokus('3', [])
```

V standardnej kniznici jendoducho nie su prostriedky na to, aby som vedel definovat vacero rovnakych fukcii a na zaklade atributov rozhodnut ktora sa ma zavolat

toto plati aj pre metody

nevieme napriklad definovat ani metodu triedy a objektu, ktora sa rovnako vola :(

Vela ludom uz napadlo, ze by nieco take bolo celkom cool a spravili nejake pokusy o zapracovanie do jazyka

http://www.grantjenks.com/docs/pypatt-python-pattern-matching/ (http://www.grantjenks.com/docs/pypatt-python-pattern-matching/)

- https://github.com/lihaoyi/macropy (https://github.com/lihaoyi/macropy) - module import
- https://github.com/Suor/patterns (https://github.com/Suor/patterns) - decorator with funky syntax - Shared at Python Brazil 2013
- https://github.com/mariusae/match (https://github.com/mariusae/match) - http://monkey.org/~marius/pattern-matching-in-python.html (http://monkey.org/~marius/pattern-matching-in-python.html) - operator overloading
- http://blog.chadselph.com/adding-functional-style-pattern-matching-to-python.html (http://blog.chadselph.com/adding-functional-style-pattern-matching-to-python.html) - multi-methods
- http://svn.colorstudy.com/home/ianb/recipes/patmatch.py (http://svn.colorstudy.com/home/ianb/recipes/patmatch.py) - multi-methods
- http://www.artima.com/weblogs/viewpost.jsp?thread=101605 (http://www.artima.com/weblogs/viewpost.jsp?thread=101605) - the original multi-methods
- http://speak.codebunk.com/post/77084204957/pattern-matching-in-python (http://speak.codebunk.com/post/77084204957/pattern-matching-in-python) - multi-methods supporting callables
- http://www.aclevername.com/projects/splarnektity/ (http://www.aclevername.com/projects/splarnektity/) - not sure how it works but the syntax leaves a lot to be desired
- https://github.com/martinblech/pyfpm (https://github.com/martinblech/pyfpm) - multi-dispatch with string parsing
- https://github.com/jldupont/pyfnc (https://github.com/jldupont/pyfnc) - multi-dispatch
- http://www.pyret.org/ (http://www.pyret.org/) - It's own language

Ziadna z tychto kniznic nie je taka dobra ako plnohodnotne zapracovana vlastnost do funkcionalneho jazyka, ale skusim aspon na takomto chabom priklade ukazat, co by sa s niecim takymto dalo robit.

Multimethods

uz aj Guido van Rossum si vsimol, ze by to mohlo byt celkom fajn

http://www.artima.com/weblogs/viewpost.jsp?thread=101605 (http://www.artima.com/weblogs/viewpost.jsp?thread=101605)

In [ ]:
```
# casto sa stava, ze kod vyzera nejak takto
def foo(a, b):
    if isinstance(a, int) and isinstance(b, int):
        # ...code for two ints...
    elif isinstance(a, float) and isinstance(b, float):
        # ...code for two floats...
    elif isinstance(a, str) and isinstance(b, str):
        # ...code for two strings...
    else:
        raise TypeError("unsupported argument types (%s, %s)" % (type(a), type(b)))
```

Nevyzeralo by to ovela lepsie takto?

Tento slajd nevidite.

je tu len pre to, aby bol kod na dalsom slajde vykonatelny. Je to kod, ktorym vkladam zelanu funkcionalitu do jazyka

In [ ]:
```
registry = {}

class MultiMethod(object):
    def __init__(self, name):
        self.name = name
        self.typemap = {}
    def __call__(self, *args):
        types = tuple(arg.__class__ for arg in args) # a generator expression!
        function = self.typemap.get(types)
        if function is None:
            raise TypeError("no match")
        return function(*args)
    def register(self, types, function):
        if types in self.typemap:
            raise TypeError("duplicate registration")
        self.typemap[types] = function

def multimethod(*types):
    def register(function):
        name = function.__name__
        mm = registry.get(name)
        if mm is None:
            mm = registry[name] = MultiMethod(name)
        mm.register(types, function)
        return mm
    return register
```

In [ ]:
```
@multimethod(int, int)
def foo(a, b):
    print('int int')

@multimethod(float, float)
def foo(a, b):
    print('float float')

@multimethod(str, str)
def foo(a, b):
    print('str str')
```

In [ ]:
```
foo(1,1)
```

Co na to treba?

- dekorator, ktory do nejakej struktury bude odkladat funkcie a parametre
- je potrebne overenie, ktora funkcia je ta spravna
- dekorator musi vratit funkciu, ktora sa pozrie do struktury s funkciami, postupne bude overovat, ci sa typy a pocty atributov zhoduju a potom jednu fukciu zavolat
- cele to ma menej ako 20 riadkov (koho to zaujima, moze sa pozriet o par saljdov vyssie ako sa to da spravit)

Obmedzenia?

- nefunguje to na zaklade pomenovanych atributov
- neda sa pouzit premenlivy pocet atributov
- atributy sa porovnavaju len na zaklade typov. Napada mi milion sposobov, ako by som chcel atributy porovnavat zlozitejsie

Mozno ina implementacia mi da vacsiu volnost

http://blog.chadselph.com/adding-functional-style-pattern-matching-to-python.html (http://blog.chadselph.com/adding-functional-style-pattern-matching-to-python.html)

In [ ]:

```python
from patternmatching import ifmatches, Any, OfType, Where

@ifmatches
def greet(gender=OfType(str), name="Joey"):
    print("Joey, whats up man?")
@ifmatches
def greet(gender="male", name=Any):
    print("Hello Mr. {}".format(name))
@ifmatches
def greet(gender="female", name=Any):
    print("Hello Ms. {}".format(name))
@ifmatches
def greet(gender=Any, name=Where(str.isupper)):
    print("Hello {}. IMPORTANT".format("Mr" if gender == 'male' else "Ms"))
@ifmatches
def greet(gender=Any, name=Any):
    print("Hello, {}".format(name))
```

In [ ]:

```python
greet('male', 'JAKUB')
```

No a posledna kniznica so zaujimavou syntaxou

https://github.com/Suor/patterns (https://github.com/Suor/patterns)

In [ ]:

```python
from patterns import patterns, Mismatch

@patterns
def factorial():
    if 0: 1
    if n is int: n * factorial(n-1)
    if []: []
    if [x] + xs: [factorial(x)] + factorial(xs)
```