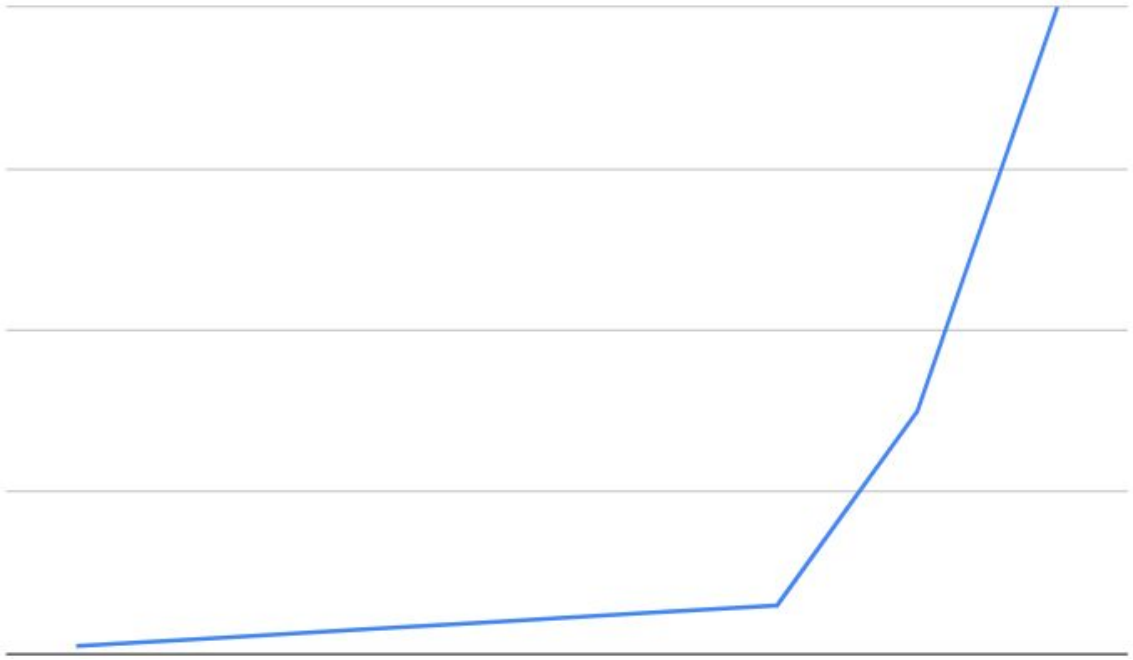
04. Декоратори

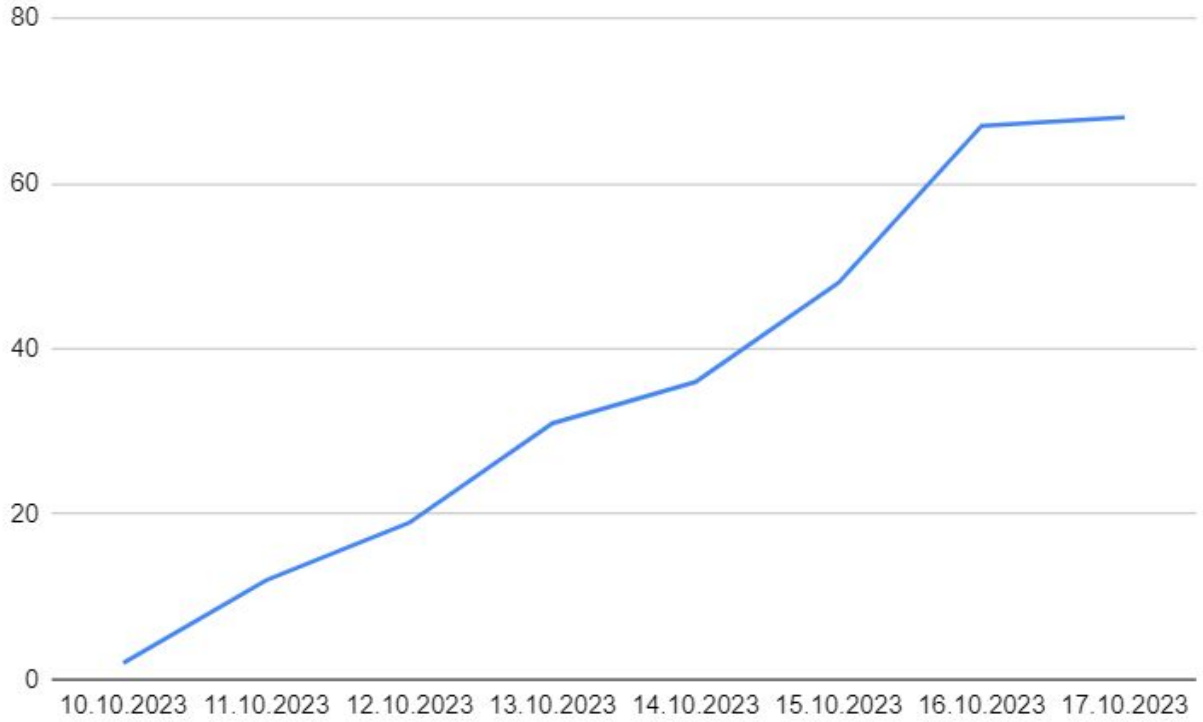
17 октомври 2023

Преди да започнем, един въпрос. Какво е това?



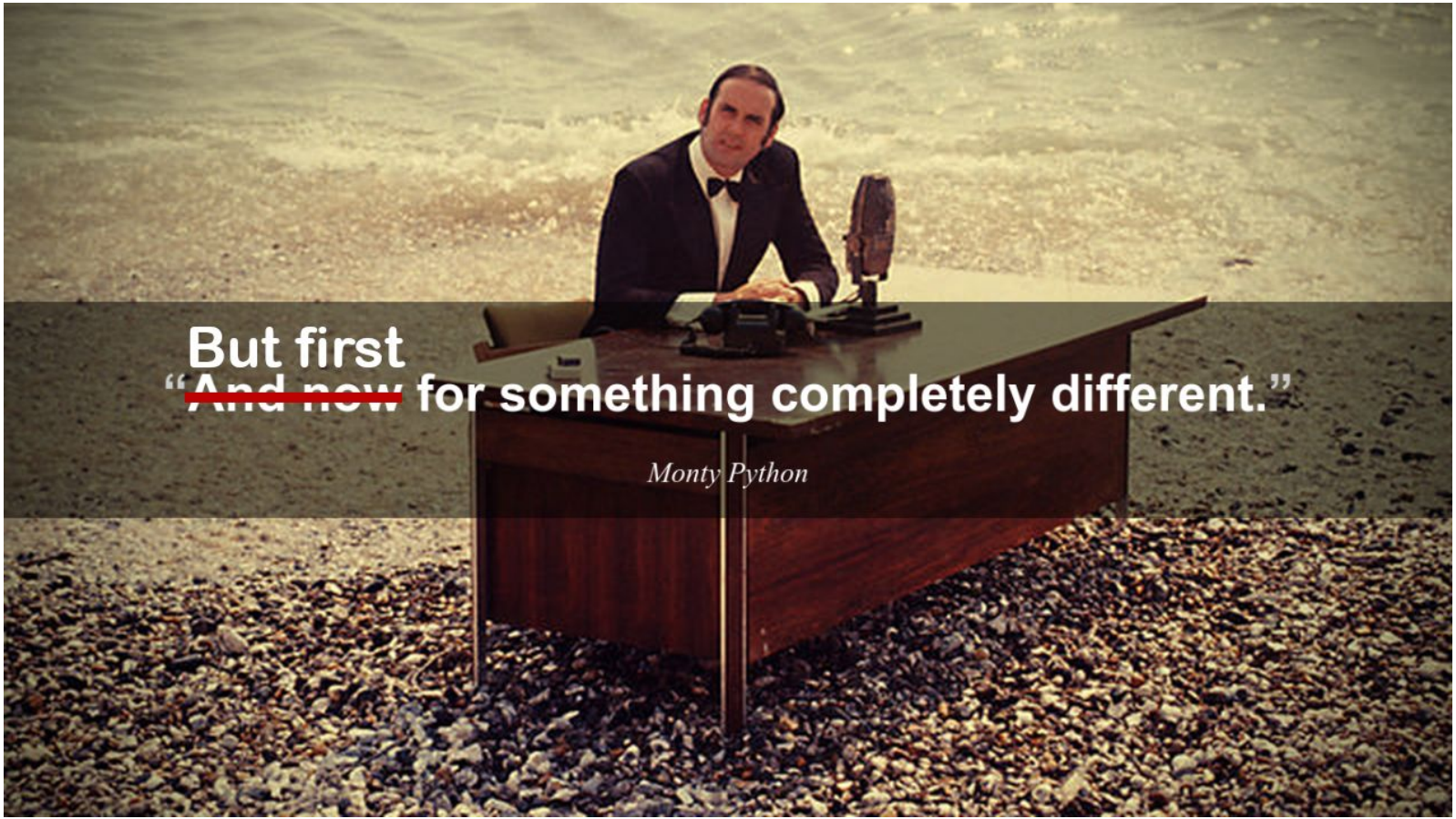
Прогресът по предаване на първото домашно от миналата година.

Тази година е далеч по-добре. Браво!



04. Декоратори

17 октомври 2023


But first
"And now for something completely different."
Monty Python

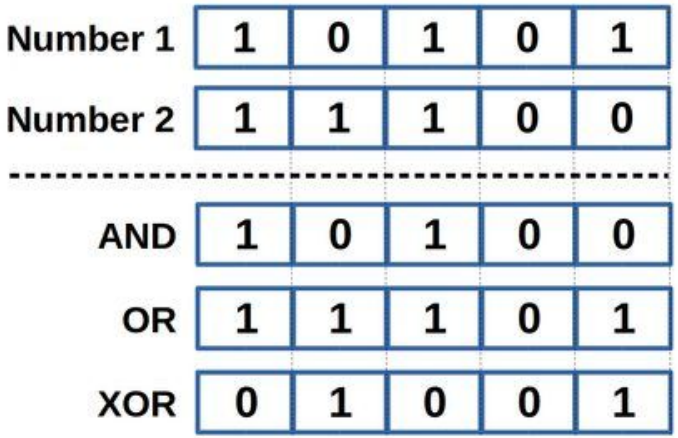
04. ДеФЛОратори

a.k.a по-битови операции

защото всяка жена обича мъже, които
владеят “битовите операции”

Побитови операции

- Числата в програмирането се съхраняват като комбинация от 0 и 1-ци, това го знаете
- Обикновенно работим с human-readable числа, а не с нули и единици
- Какво правим ако искаме да пипнем по-дълбоко, обаче?
- Enter bitwise operations



&, |, ^, ~, >>, <<, (ノ °□°)ノ ︵ ┻━┻

- Всяка от тези операции работи директно с двоичната репрезентация на числата
- & - побитово (логическо) **И**
- | - побитово **ИЛИ**
- ^ - побитово изключващо или (**XOR**)
- ~ - побитово отрицание (**НЬЕ**)
- >> - **shift надясно**
- << - **shift наляво**
- (ノ °□°)ノ ︵ ┻┻ **-** побитово ‘бал съм му майката

Особености

```
>>> bin(-2)
'-0b10'
>>> bin(-3)
'-0b11'
>>> -2 & -3
-4
```

- Отрицателните числа в Python са имплементирани използвайки механизмът two's complement
- Реално `-2` не е `-0b10`, това е human-readable двоична репрезентация
- Всъщност е `1.....11111111111110`
- Няма да навлизаме в детайли как работят компютрите, защото става дълбоко, но предлагаме добро четиво по темата

Особености 2

```
>>> 0.1 & 0.2
Traceback (most recent call last):
  File "<pyshell#30>", line 1, in <module>
    0.1 & 0.2
TypeError: unsupported operand type(s) for &: 'float' and 'float'
```

- Числата с плаваща запетая са имплементирани с експонента и мантиса
- Ерго побитовите операции нямат смисъл в този контекст

Защо бихме ги използвали?

- Защото пишем на друг език преди 20 години
- Защото са секси и мацките / пичовете (много сме прогресивни) им се кефят
- Защото искаме да слагаме ей такива коментари в кода си:

```
i  = * ( long * ) &y;        // evil floating point bit level hacking
i  = 0x5f3759df - ( i >> 1 );               // what the fuck?
y  = * ( float * ) &i;
y  = y * ( threehalfs - ( x2 * y * y ) );   // 1st iteration
```

- Флагове
- `chmod 644` някой?


“And now for something completely different.”
Monty Python

Какво видяхте в предишната лекция?

- Какво е колекция (нещо итерируемо)
- List (“указатели”, индексиране и слайсване, основни методи)
- Tuple (immutable, unpacking, променлив брой стойности (*), сравнения)
- Set (уникални стойности, константно търсене, сечения на множества)
- Dict (ключ-стойност, константно търсене, (не)подреден, хеш функции)
- map/filter/all/any (lazy генериране на колекции и функции за булева оценка)
- lambda функции (анонимни едноредови функции)
- Comprehensions (list, generator, set, dict)
- Голи снимки
- Който видял - видял

Въпроси?

Подай Snickers-а там (1)

```
>> spam = {}
>> type(spam) # ?
```

```
<class 'dict'>
```

Подай Snickers-а там (2)

```
>> lectures_location = {'Tuesday': '210', 'Thursday': '101'}
>> lectures_location['Wednesday'] # ?

Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    lectures_location['Wednesday']
KeyError: 'Wednesday'
```

Подай Snickers-а там (3)

```
>> type(map(lambda x: '-'.join(x.split('_ @')), ['this_is_not_funny'])) # ?
<class 'map'>
```


Коя колекция може да бъде итерирана с for?
A: Dict
B: Set
C: List
D: Tuple

За да разберем какво са декоратори, трябва да знаем...

- Какво е функция
- Какво е област на видимост
- Какво са вложени функции (е, то е очевидно)
- Какво значи, че функциите са първокласни обекти
- Какво е closure

Едно по едно...

Какво е функция

В python функциите са обекти!

```
def baba():
    print('баница')

def call(function, times):
    for _ in range(times):
        function()

call(baba, 4)
# баница
# баница
# баница
# баница
```

Предговор

- Какво различава функцията от повечето други обекти?
  - `__call__`
- Какви типове обекти може да връща една функция?
  - Всякакви.

Области на видимост

- Всяка променлива (име) може да бъде свързана със стойност (binding)
- Има операции, които променят свързването, например =

```
global_one = 1

def foo():
    local_one = 2
    print(locals())

print(globals())    # {..., 'global_one': 1}
foo()               # {'local_one': 2}
```

locals/globals

Вградени функции

- `locals()` – връща речник с всички имена в локалната област на видимост
- `globals()` – връща речник с всички имена в глобалната област на видимост

Области на видимост**2

- Всеки блок от код (напр. функция, модул, дефиниция на клас) си има своя област на видимост, в която стоят локално дефинираните променливи
- Ако една функция не може да намери дадена променлива в локалния си скоуп, търси в обграждащия (глобалния) за променлива със същото име

```
global_one = 1

def foo():
    print(global_one)

foo()
```

Области на видимост**2**2

А какво ще изведе следният код?

```
global_one = 1

def foo():
    global_one = 2
    print(global_one)
    print(locals())

foo()
print(globals())
```

- По подразбиране пренасочването на имена става в локалния скоуп
- Използването на ключовата дума global позволява пренасочването на глобални имена
- **НЕ ИСКАТЕ ДА ПОЛЗВАТЕ** `global`

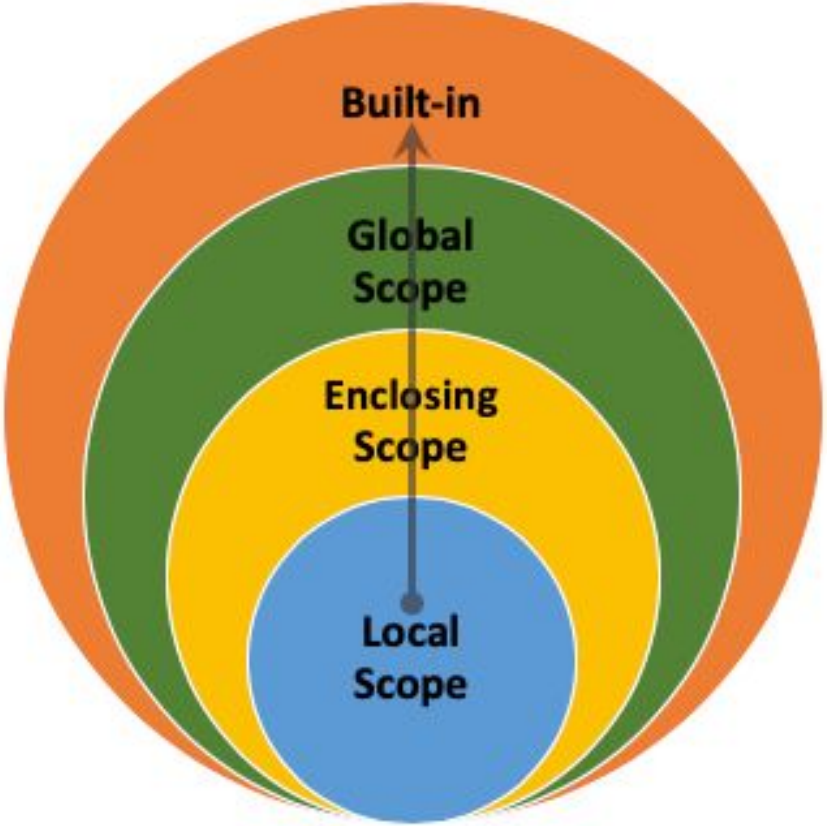
Животът на една променлива

- Една променлива "умира" заедно със своя скоуп...
- Но ние не страдаме, защото такъв е живота на кода
- Амин!

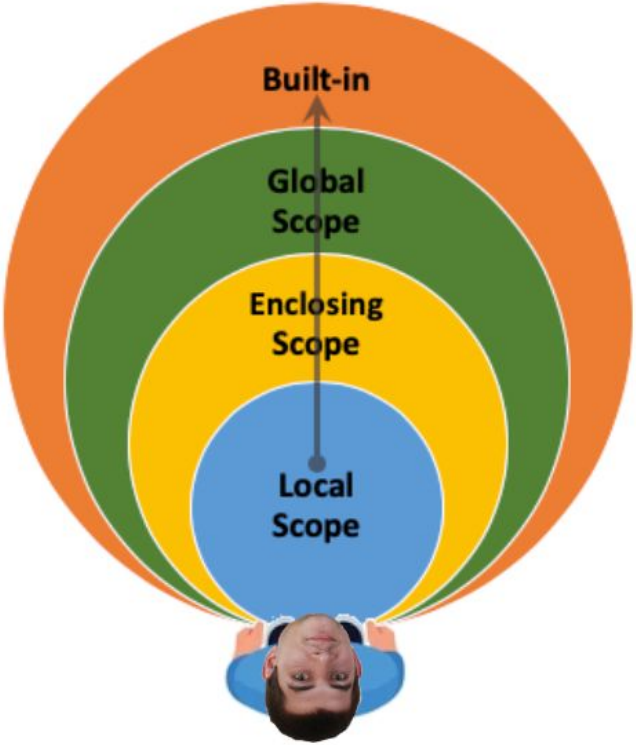
Пре(д)говор - аргументи

- Можем да ги подаваме като позиционни или като именувани
- След именуван аргумент не можем да подадем позиционен
- Подадените аргументи отиват в locals
- Очевидно умират с приключването на функцията си

Local/Globals



Народна мъдрост



размерът на корема на Георги ∝ изпитата от него бира

точно както

броят обекти ∝ броят скоупове, в които те се търсят

Вложени функции

- Можем да дефинираме функция тялото на друга функция
- Друга тема е кога това е добра идея
- Какво се случва тогава с променливите на двете функции и къде отиват?

```
def outer(x):
    print(x)
    def inner():
        x = 0
        print(x)
    inner()
    print(x)
```

- Името `inner` също отива в `locals()` на outer
- Ключовата дума nonlocal позволява пренасочване на име, дефинирано в обграждащ блок
- Познайте на какво мнение сме за ползването на nonlocal

Функциите са първокласни обекти

- Те са като всички останали обекти
- Можем да ги подаваме като аргументи
- Можем да ги връщаме като резултат
- Можем да ги записваме в колекции
- Можем да ги присвояваме на променлива
- Имат идентитет - `id()`

Closures

Имаме closure, когато вложена функция достъпва променлива, дефинирана в обграждаща функция

```
def start(x):
    def increment(y):
        return x + y
    return increment

first_inc = start(0)
second_inc = start(8)

first_inc(3)
second_inc(3)

first_inc(1)
second_inc(2)
```

[Edit] За да разберем какво са декоратори, трябва да знаем...

- Какво е функция ✅
- Какво е област на видимост ✅
- Какво са вложени функции (е, то е очевидно) ✅
- Какво значи, че функциите са първокласни обекти ✅
- Какво е closure ✅
- **+ Сериозен проблем**

Един сериозен проблем

- Занимаваме се с известен ресторант
- В него може да се поръчва храна със следните функции
- Игнорирайте това, че функциите ще се държат странно с n <= 1
- Ще трябва да ни повярвате, че знаем как да се справим с проблема

```
def spam(n):
    spams = ("spam", ) * (n - 1)
    return "I would like {} and spam".format(", ".join(spams))

def eggs(n):
    return "I would like {} eggs".format(n)
```

Обноски

- Внезапно се сещаме, че преувеличените обноски са хубаво нещо
- Искаме след поръчката да кажем "dear sir" или "dear madam" в зависимост от пола на обслужващия ни този ден
- No offence, LGBT, но всяка жаба да си знае гьола

Начин 1

```
def spam(n, server):
    spams = ("spam", ) * (n - 1)
    return "I would like {} and spam, dear {}".format(", ".join(spams), server)

def eggs(n, server):
    return "I would like {} eggs, dear {}".format(n, server)

spam(3, "sir")
```

- Easy! Caveman approach. Добавяме втори аргумент
- Да, но сега всеки път, когато си поръчваме нещо, ще трябва да се сещаме какъв беше полът на сервитьора. Би било хубаво ако можеше някакси само веднъж да се занимаваме с това.
- Ако ресторантът ни имаше 100 различни неща за поръчване? Ако искаме да сменим формата от "dear madam" на нещо друго?

До преди малко говорихме за функции

```
def spam(n):
    spams = ("spam", ) * (n - 1)
    return "I would like {} and spam".format(", ".join(spams))

def eggs(n):
    return "I would like {} eggs".format(n)

def served_by(func, server):
    def cached_server(n):
        return "{}, dear {}".format(func(n), server)
    return cached_server

eggs = served_by(eggs, "sir")
spam = served_by(spam, "sir")
```

Да благодарим

Когато поръчваме яйца, винаги да благодарим

```
def thank_you(func):
    def with_thanks(n):
        return "{}. Thank you very much!".format(func(n))
    return with_thanks

eggs = thank_you(served_by(eggs, "sir"))
spam = served_by(spam, "sir")
```

Започна да става сложно

- Доста се натовари създаването на нашите функции
- Има по - добър начин. Ще го покажем след малко
- Но преди това един друг пример

Фибоначи

```
def fibonacci(x):
    if x in (0, 1):
        return 1
    return fibonacci(x - 1) + fibonacci(x - 2)
```

Рекурсивната версия на `fibonacci`, освен че е бавна, е много бавна. Особено усезаемо, когато x >= 40.

Проблемът е, че `fibonacci` се извиква стотици пъти с един и същ аргумент. Можем спокойно да прегенерираме първите стотина резултата в един речник или...

Да изчисляваме всеки резултат само по веднъж...

```
if x not in memory:
    memory[x] = fibonacci(x)

print(memory[x])
```

Разбира се, тази идея може да се използва и на много повече места! Можем да я направим още по-елегантно.

Функции, които опаковат други функции

- f(функция) -> функция
- резултатът е нова функция, която "опакова" старата и може да разшири нейната функционалност

memoize

```
def memoize(func):
    memory = {}
    def memoized(*args):
        if args in memory:
            return memory[args]
        result = func(*args)
        memory[args] = result
        return result
    return memoized

fibonacci = memoize(fibonacci)
```

Красивият синтаксис

```
def fibonacci(x):
    if x in (0, 1):
        return 1
    return fibonacci(x - 1) + fibonacci(x - 2)

fibonacci = memoize(fibonacci)
```

Е същото като...

```
@memoize
def fibonacci(x):
    if x in (0, 1):
        return 1
    return fibonacci(x - 1) + fibonacci(x - 2)
```

Sugar Bae

Друг пример за декоратор

```
def notifyme(f):
    def logged(*args, **kwargs):
        print(f.__name__, ' called with ', args, ' and ', kwargs)
        return f(*args, **kwargs)
    return logged

@notifyme
def square(x):
    return x * x
result = square(25) # 625

# square was called with (25,) and {}.
```

Яйца?

```
def served_by(server):
    def decorator(func):
        def cached_server(n):
            return "{}, dear {}".format(func(n), server)
        return cached_server
    return decorator

def thank_you(func):
    def with_thanks(n):
        return "{}. Thank you very much!".format(func(n))
    return with_thanks

@served_by("sir")
def spam(n):
    spams = ("spam", ) * (n - 1)
    return "I would like {} and spam".format(", ".join(spams))

@thank_you
@served_by("sir")
def eggs(n):
    return "I would like {} eggs".format(n)
```

Въпроси?