Разработка на софтуер

Лекция 7 – Python функции

Милен Спасов

Функции и област на видимост (1)

Какви типове обекти може да връща една функция?

- Всякакви

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}
```

Вградени функции:

locals – връща речник с всички имена в локалната област на видимост

globals – връща речник с всички имена в глобалната област на видимост

Функции и област 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()
```

А какво ще изведе следният код?

```
global_one = 1

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

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

Аргументи

Можем да ги подаваме като позиционни или като именовани

След именован аргумент не можем да подадем позиционен

Подадените аргументи отиват в locals

Очевидно умират с приключването на функцията си

Вложени функции

Можем да дефинираме функция тялото на друга функция

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.

Генерално е лоша практика да ползвате пренасочвания на променливи.

Функциите са 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

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)
```

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 когато x >= 40.

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

Да изчисляваме всеки резултат само по веднъж...

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

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 ф-я, която "опакова" старата и може да разшири нейната функционалност

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

fibonacci = memorize(fibonacci)
```

Красивият синтаксис

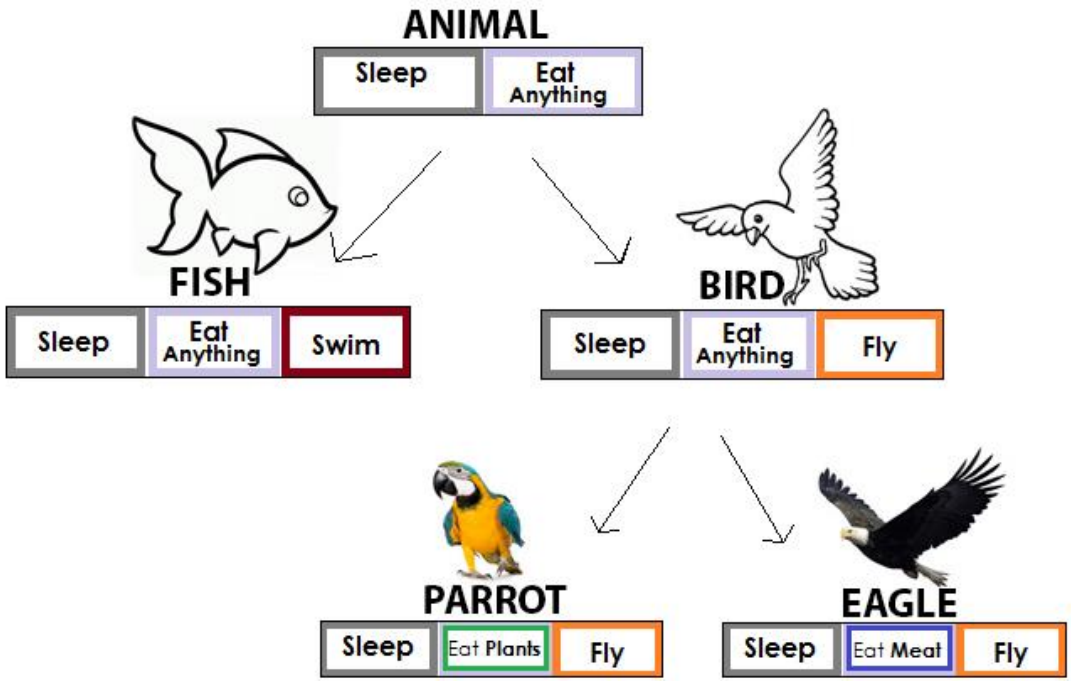
```
def fibonacci(x):
    if x in [0,1]:
        return 1
    return fibonacci(x-1) + fibonacci(x-2)

fibonacci = memorize(fibonacci)
```

Декорацията става след дефиницията на функцията.

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

ООП принципи – Абстракция

[…] a technique for managing complexity of computer systems. It works by establishing a level of complexity on which a person interacts with the system, suppressing the more complex details below the current level. The programmer works with an idealized interface (usually well defined) and can add additional levels of functionality that would otherwise be too complex to handle.

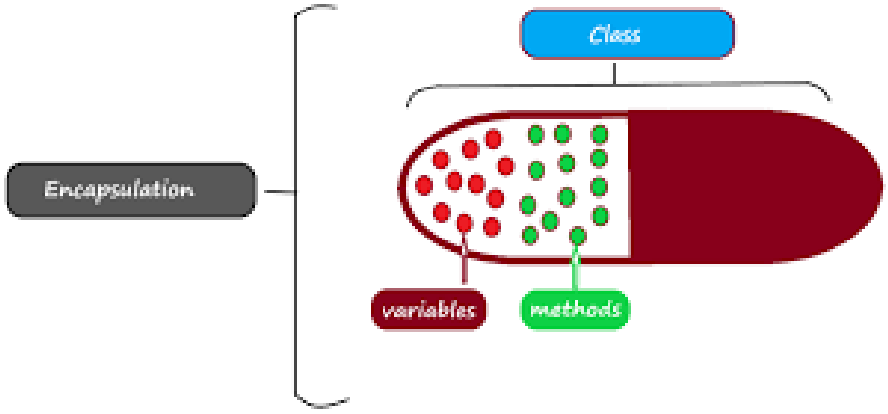


ООП принципи – Енкапсулация

Encapsulation is the packing of data and functions into a single component.

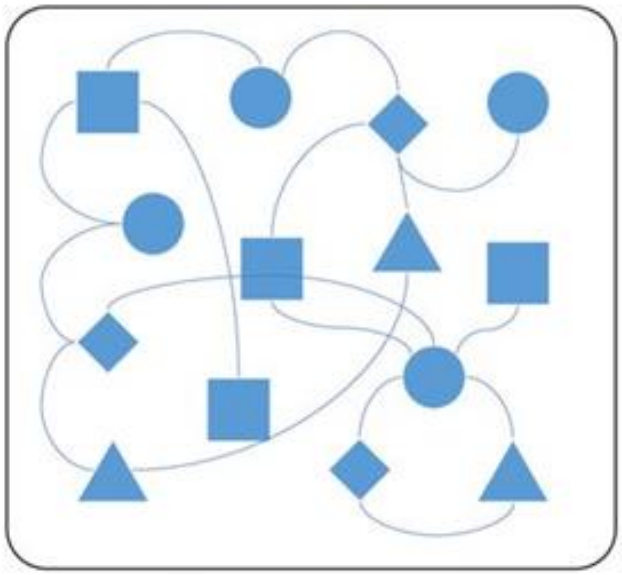
Има много механизми, с които можем да го реализираме, но обектно ориентираното програмиране е един от най-популярните.

Abstraction and encapsulation are complementary concepts: abstraction focuses on the observable behavior of an object... encapsulation focuses upon the implementation that gives rise to this behavior... encapsulation is most often achieved through information hiding, which is the process of hiding all of the secrets of object that do not contribute to its essential characteristics.

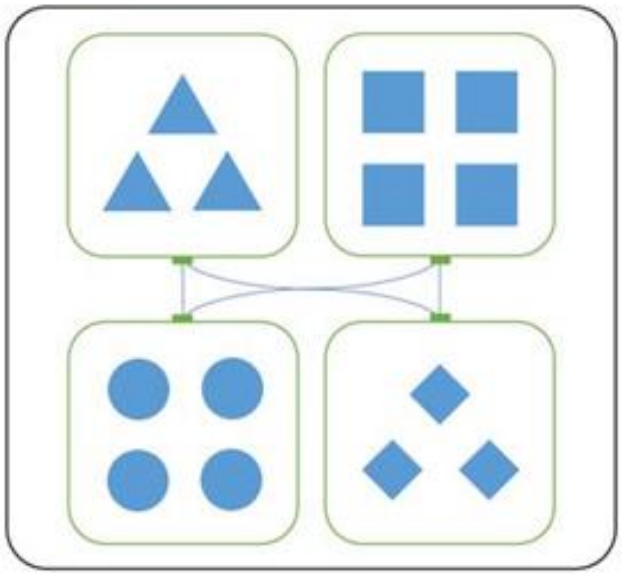


ООП принципи – Модулярност

Механизъм за организиране на сходна логика работеща върху свързани видове данни, обработваща сходни видове процеси от моделирания свят в добре обособени и ясно разделени парчета от кода ни



Non-modular software



Modular software

Класове

- Създаваме класове с ключовата дума class, след което всяка функция дефинирана в тялото на класа е метод, а всяка променлива е клас променлива

```
class Vector:
   def __init__(self, x, y):
      self.x = x
      self.y = y

spam = Vector(1.0, 1.0)
print(spam.x)
```

- „Конструктура“ се казва __init__ и той не връща стойност.
- Първия аргумент на методите винаги е инстанцията, върху която се извикват. Той може да се казва всякак, но винаги се казва self, иначе никой не иска да си играе с вас и всички ви мразят.
- Атрибутите не се нуждаят от декларации (обектите са отворени).
- Инстанцираме клас, като го „извикаме“ със съответните аргументи, които очаква __init__ метода му и като резултат получаваме новоконстурирания обект.

Примерен клас Vector

```
import math

class Vector:
    def __init__(self, x, y): ...

    def length(self):
        return math.sqrt(self.x**2 + self.y**2)

    spam = Vector(1.0, 2.0)
    print(spam.length())
```

- „Конструктор“ е думата, с която сте свикнали, но в случая далеч по-подходяща е „инициализатор“, както си личи от името.
- В методите атрибутите могат да се достъпват само през self, няма никакви магически имплицитни scope-ове.
- Методите се извикват през инстанцирания обект обект.метод().

Примерен клас Vector

```
import math

class Vector:
    def __init__(self, x, y): ...

    def length(self):
        return math.sqrt(self.x**2 + self.y**2)

    spam = Vector(1.0, 2.0)
    print(spam.length())
```

- „Конструктор“ е думата, с която сте свикнали, но в случая далеч по-подходяща е „инициализатор“, както си личи от името.
- В методите атрибутите могат да се достъпват само през self, няма никакви магически имплицитни scope-ове.
- Методите се извикват през инстанцирания обект обект.метод().

Примерен клас Vector (2)

```
class Vector:
    def __init__(self, x, y, z): ...

    def _coords(self):
        return (self.x, self.y, self.z)

    def length(self):
        return sum(_ ** 2 for _ in self._coords()) ** 0.5
```

- _coords е protected метод
- Отново, методите се извикват върху self
- _ е валидно име за променлива

Private / Protected

- Казахме, че класовете са отворени. Това ще рече, че private и protected концепциите не са това, за което сте свикнали да мислите в езици като C++/Java/C#

- Ограниченията за използване на защитени и частни методи в класовете в Python е отговорност на програмиста, което по никакъв начин не прави живота ви по-труден

- Методи/атрибути започващи с _ са защитени, т.е. би следвало да се ползват само от методи на класа и наследяващи го класове

- Методи/атрибути започващи с __ са частни, т.е. би следвало да се ползват само от методи на класа

- Достатъчно очевидно е, а някои много редки случаи може да се наложи тези ограничения да не се спазят

Сравняване на обекти

- Можете да проверите дали два обекта са равни по стойност с ==
- Можете да проверите дали две имена сочат към един и същи обект с is
- Можете да предефинирате равенството за обекти от даден клас с метода __eq__
- По подразбиране, __eq__ е имплементирана с is

```
class Vector:
    def __init__(self, x, y, z):
        self._coords = (x, y, z)

    def __eq__(self, other):
        return self._coords == other._coords
```

Аритметични оператори

- __add__(self, other) - self + other
- __sub__(self, other) - self - other
- __mul__(self, other) - self * other
- __truediv__(self, other) - self / other
- __floordiv__(self, other) - self // other
- __mod__(self, other) - self % other
- __lshift__(self, other) - self << other
- __rshift__(self, other) - self >> other
- __and__(self, other) - self & other
- __xor__(self, other) - self ^ other
- __or__(self, other) - self | other

- Преобразуване до стандартни типове
- __int__(self) - int(обект)
- __float__(self) - float(обект)
- __complex__(self) - complex(обект)
- __bool__(self) - bool(обект)


Thank
You
Asante
Shukria
Dhanyavadagalu
Vinaka
köszönöm
Kiitos
Maake
Manana
Dankon
Matondo
감사합니다
Kam Sah Hammida
Dankscheen
спасибо
Dank Je
Mauruuru
Biyan
Terima Kasih
Taiku
Blagodaram
Ngiyabonga
Dziekuje
Chokrane
Diolch i Chi
Tack
Grazie
Juspaxar
Arigato
Mochchakkeram
Gracias
Tingki
நன்றி
धन्यवाद
cảm ơn bạn
Gràzas
Dàkujem
Bedankt
Ua Tsaug Rau Koj
Paldies
Gratias Tibi
Khap
Obrigado
Děkuji
Nirringrazzjak
Di Ou Mèsi
Kia Ora
Kop Khun
ありがとう
Suksama
Hvala
Djiere Dieuf
Welalin
Matur Nuwun
Rahmat
谢
谢
хвала
Danke
Merci
Go Raibh Maith Agat
Misaotra
ขอขอบคุณคุณ
Salamat
Najis Tuke
Eskerrik Asko