



Prof. Luciano Nunes
2020/1
lnunes at gmail

Aulas 1 e 2

- Linguagens de Programação
- Importância da Orientação a Objetos
- Boas práticas de Programação
- Algoritmo
- Ambiente de Desenvolvimento
- Introdução ao Python
 - Sintaxe Básica
 - Variáveis
 - Tipos de Dados
 - Funções mais comuns
 - Controle de Fluxo
 - Funções
 - Operadores

Aula 3

- Módulos e Pacotes
 - datetime
 - time
 - calendar
- Entrada e Saída (I/O)
- Erros e Exceções
 - Captura e Tratamento de Exceções

Aula 4 - Final do módulo

- Lambda, map, filter e zip
- List Comprehension
- Orientação a Objetos



Lambda

- Conceito e nomes vieram da linguagem funcional Lisp
- São funções anônimas que aceitam argumentos e que só suportam uma expressão
- São usadas basicamente quando há a necessidade de uma função que não seja complexa o suficiente que justifique sua criação
- Nesse tipo de expressão, o Python retorna a própria função, invés de atribuí-la a um nome como acontece em um statement "def", daí vem a particularidade de serem anônimas

Lambda is the keyword, equivalent to def.

Notice that there is no function name (anonymous functions)

Lambda function can contain only one expression

The result of this expression is returned.

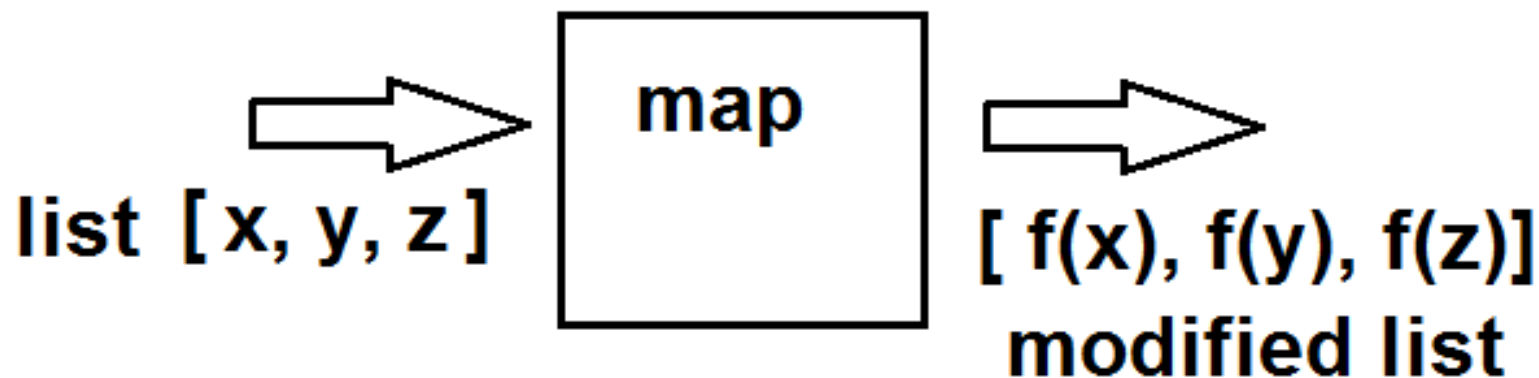
```
sqr = lambda (x, y) : x**y
```

Lambda functions can be assigned to variable
But not mandatory

Lambda function accepts any number of arguments
Parenthesis is not mandatory

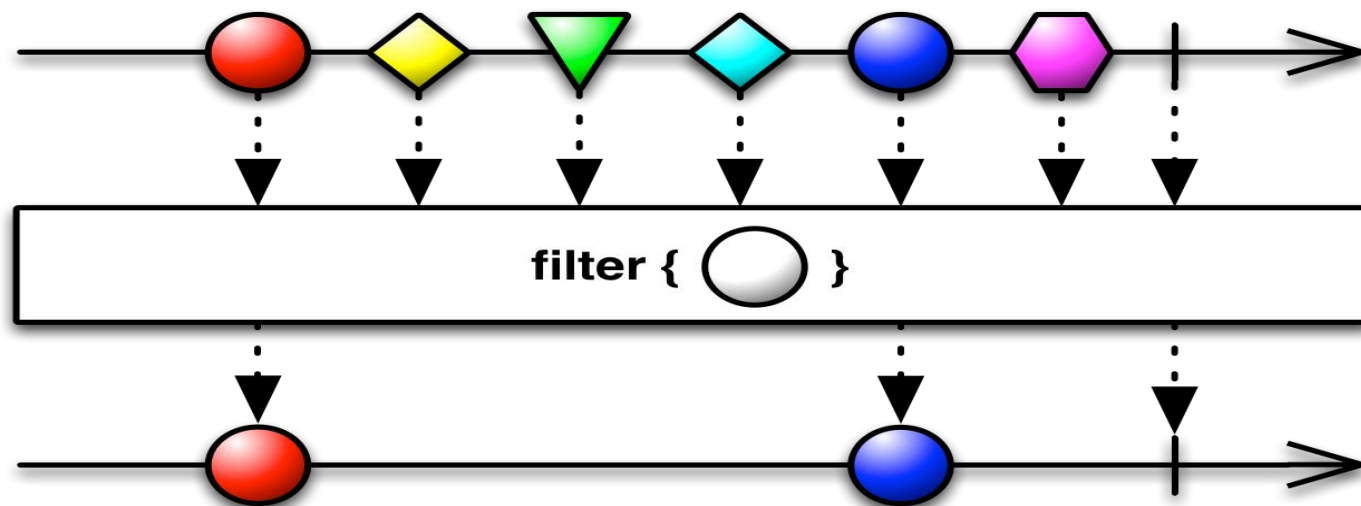
Map

- Map é uma função embutida (*built in*) do Python
- Basicamente map executa uma determinada função a uma sequência de valores, aplicando essa função a cada elemento dessa sequência
- Utilizado em conjunto com as instruções lambda, podem ser muito úteis para a transformação de elementos de uma lista de forma massiva



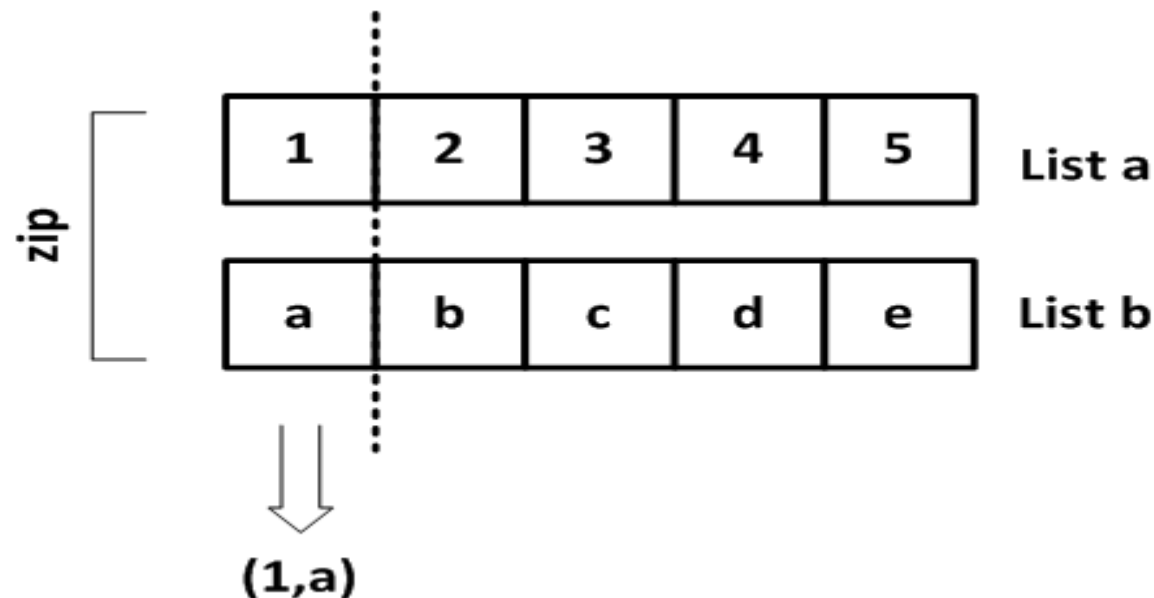
Filter

- Filter é uma função embutida (*built in*) do Python
- Irá iterar e retornar um iterador para os elementos de uma lista, caso a condição da função passada como primeiro argumento seja satisfatória
- Admite None como parâmetro da função, irá retornar um iterador para os elementos verdadeiros da lista de entrada
- Assim como o map, pode ser utilizado em conjunto com instruções lambda



Zip

- Zip é uma função embutida (*built in*) do Python
- Retorna um iterador para uma lista com a combinação dos elementos de cada iterador passado como argumento
- Caso os iteradores passados como argumentos tenham tamanhos diferentes, será retornado um iterador com o tamanho do menor iterador passado como argumento



List Comprehension

- Embora as expressões lambda e suas associações com funções built in do Python como map e filter sejam interessantes, o criador do Python optou por mantê-las na linguagem apenas para atrair programadores de outras linguagens funcionais como Lisp
- No entanto sua preferência é pelas *list comprehensions* que nada mais são do que a forma *Pythonica* ou mais elegante de escrever a mesma coisa
- Portanto, praticamente tudo o que é possível de se fazer usando expressões lambda, map e filter, também é possível usando *list comprehensions*
- Tanto em desempenho quanto em concisão ambas as formas são bastante semelhantes, apenas o estilo de programação é diferente

[expression for name in list if filter]

```
>>> li = [1,2,3,4,5,6,7,8,9,10]
```

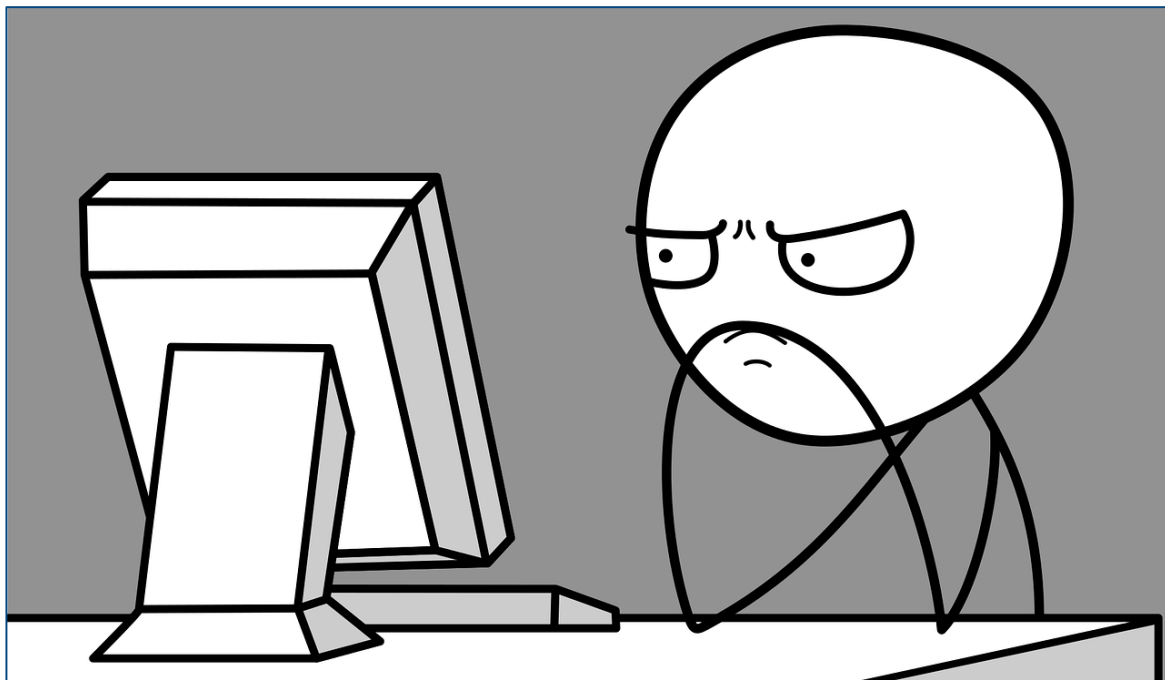
```
>>> [x**2 for x in li if x%2==0]
```

```
[4, 16, 36, 64, 100]
```


Lambdas X List Comprehension

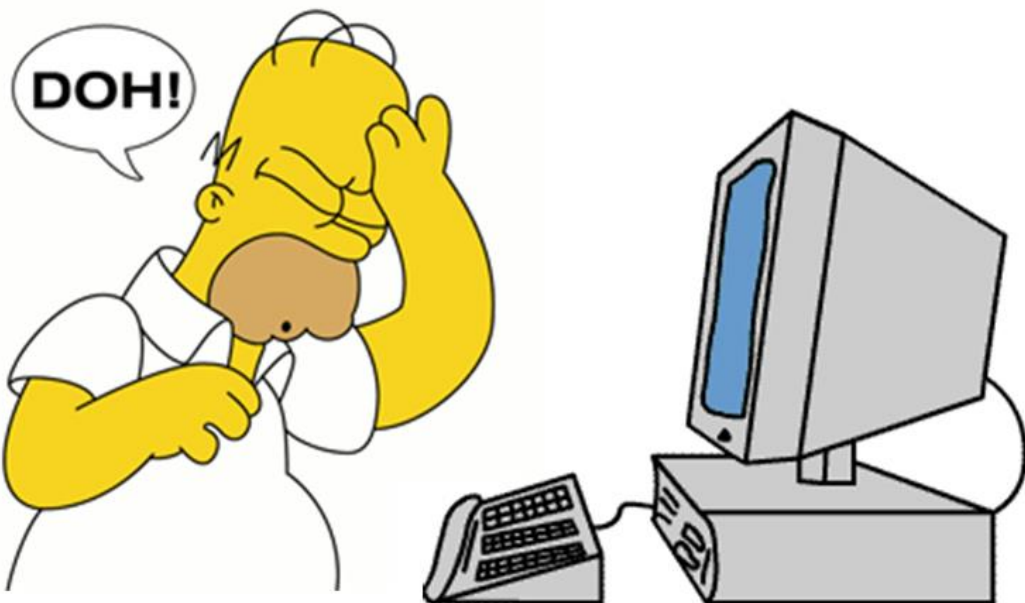
```
comprehensions.py x
1      # imperative form
2      odd = []
3      for i in range(1000):
4          if i % 2:
5              odd.append(i)
6      print(odd)
7
8      # functional form
9      print(list(filter(lambda i: i % 2, range(1000))))
10
11     # comprehension form
12     print([i for i in range(1000) if i % 2])
13
14
15
```

Lambdas X List Comprehension



- Uma das questões muito discutidas é sobre a complexidade que pode estar relacionada à utilização de expressões lambdas complexas, assim como list comprehensions com muitas camadas de lógica aninhadas
- Procure manter um único estilo
- Eventualmente várias linhas de código podem ser substituídas por apenas uma linha utilizando-se desse tipo de facilidade que a linguagem oferece, no entanto é importante utilizar esses recursos com sabedoria, lembrando-se sempre de comentar seu código
- Você pode imaginar-se não compreendendo um programa escrito por você mesmo 6 meses atrás?

Orientação a Objetos



- Herança
- Polimorfismo
- Abstração
- Encapsulamento
- Associação (Composição / Agregação)
- Generalização / Especialização
- Dependencia
- Classe
- Instância / Objeto

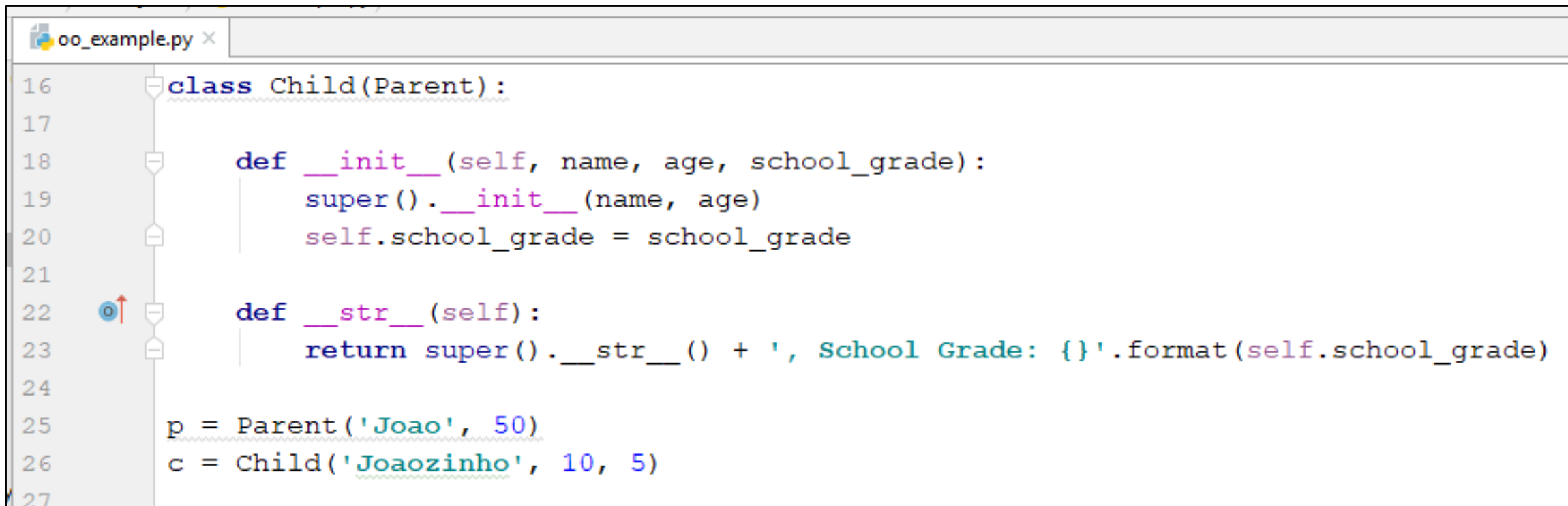
Orientação a Objetos

- O statement *class* indica uma uma definição de uma nova classe
- A função `__init__(self, args...)` define o construtor da classe, sendo o primeiro argumento obrigatoriamente a própria instância da classe (*self*)
- Classes podem ter variáveis de classe que são compartilhadas entre todas as instâncias dessa mesma classe e variáveis de instância que são exclusivas a cada instância
- Instâncias podem receber atributos dinamicamente

```
oo_example.py x
1 class Parent:
2     # Class variable
3     count = 0
4
5     # constructor
6     def __init__(self, name, age):
7         # instance variables
8         self.name = name
9         self.age = age
10        Parent.count += 1
11
12    # overriding str
13    def __str__(self):
14        return 'Name: {}, Age: {}'.format(self.name, self.age)
15
16
17    p1 = Parent('Joao', 10)
18    p2 = Parent('Maria', 11)
19    print(p1)
20    print(p2)
21    print(Parent.count)
22
```

Orientação a Objetos

- Herança é definida quando na definição da classe é indicada a classe da qual ela extenderá



```
oo_example.py x
16 class Child(Parent):
17
18     def __init__(self, name, age, school_grade):
19         super().__init__(name, age)
20         self.school_grade = school_grade
21
22     def __str__(self):
23         return super().__str__() + ', School Grade: {}'.format(self.school_grade)
24
25 p = Parent('Joao', 50)
26 c = Child('Joaozinho', 10, 5)
27
```

Orientação a Objetos

- Herança multipla funciona da mesma forma, inserindo-se outras classes que deverão ser extendidas

```
oo_example.py x
16  class Walker:
17
18      def walk(self):
19          print('Walking...')
20
21      def run(self):
22          print('Running...')
23
24  class Child(Parent, Walker):
25
26      def __init__(self, name, age, school_grade):
27          super().__init__(name, age)
28          self.school_grade = school_grade
29
30      def __str__(self):
31          return super().__str__() + ', School Grade: {}'.format(self.school_grade)
32
```

Orientação a Objetos

- O encapsulamento consiste em “esconder” os atributos de uma classe de forma que não possam ser acessados diretamente
- Em Python não é comum a criação de métodos get/set para atributos caso eles não tenham a real necessidade de serem privados
- A necessidade de privar o acesso a um atributo se dá quando o controle do seu valor é necessário
- Para tornar um atributo privado, basta iniciar seu nome com um duplo underscore `__attrib`
- Os *statements* `@property` e `@attr.setter` é a forma “Pythonica” de definir métodos *getter* e *setter*

```
properties.py x
1 class Account(object):
2
3     def __init__(self, number):
4         number = number
5         self.__special = False
6         self.__balance = 0
7
8     def deposit(self, amount):
9         self.__balance += amount
10
11    def withdrawal(self, amount):
12        if not self.__special and amount > self.__balance:
13            raise Exception('Balance not enough')
14        self.__balance -= amount
15
16    def upgrade(self):
17        self.__special = True
18
19    @property
20    def balance(self):
21        return self.__balance
22
```

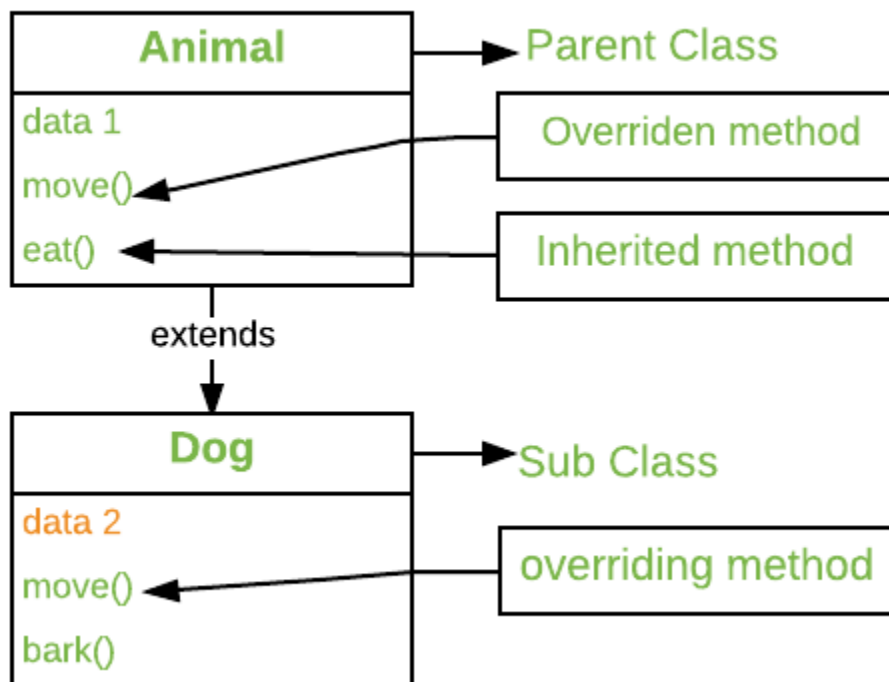
Orientação a Objetos

- Em Python não existe o conceito de interfaces para definição de comportamentos, pois tem suporte a herança múltipla
- Para a definição de classes abstratas, Python provê uma infraestrutura denominada ABC (Abstract Base Classes)
- Para tornar uma classe abstrata, basta estender a classe ABC
- O *statement* **@abstractmethod** instrui o interpretador de que esse método não está implementado e caso uma nova instancia dessa classe seja criada, um erro de execução será retornado
- O método **super()** é utilizado para acesso ao objeto pai

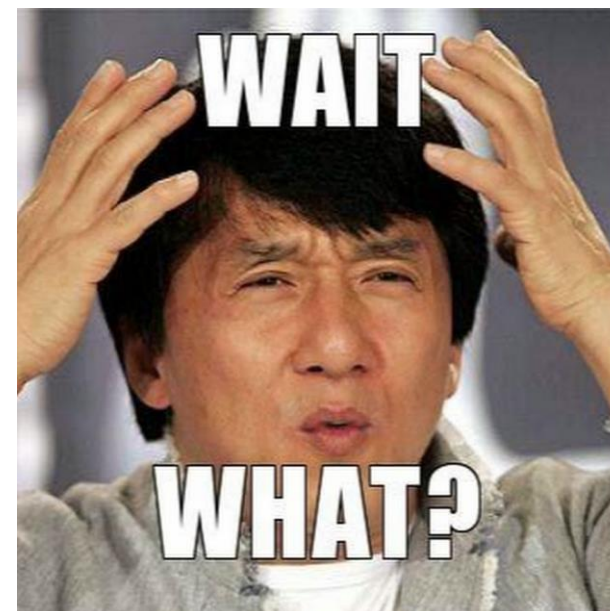
```
abstract.py x
1      from abc import ABC, abstractmethod
2
3      class Employee(ABC):
4          def __init__(self, name, salary):
5              self.name = name
6              self.salary = salary
7
8          @abstractmethod
9          def bonus(self):
10             pass
11
12     class Operator(Employee):
13         def bonus(self):
14             return self.salary / 2
15
16     class Manager(Operator):
17         def bonus(self):
18             return super().bonus() + self.salary / 3
19
20     class Director(Manager):
21         def bonus(self):
22             return super().bonus() + self.salary * 2
23
```


Orientação a Objetos

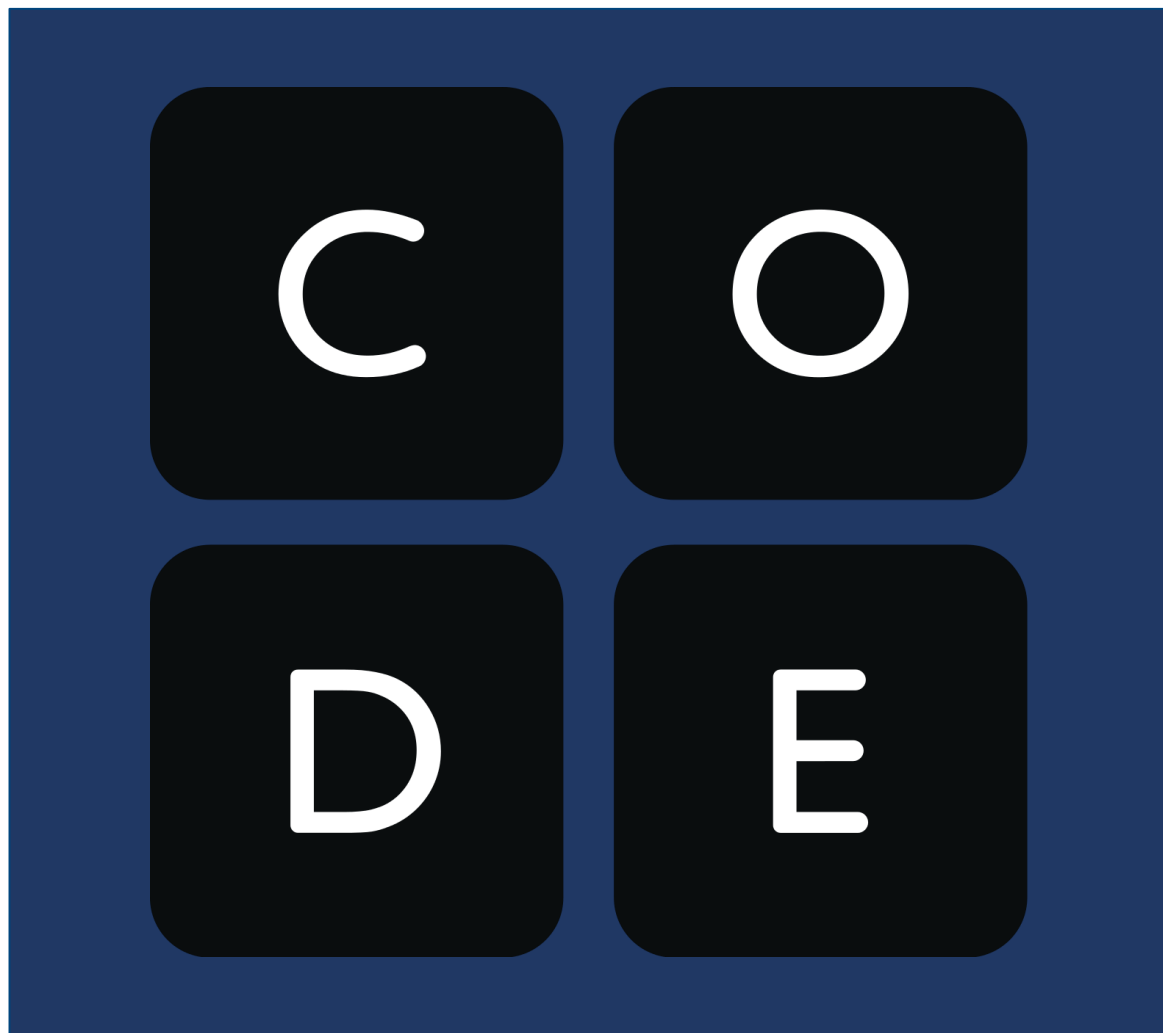
- Sobrescrita de métodos (*overriding*) é permitida, bastando para isso escrever na subclasse o mesmo nome do método com a mesma quantidade de parâmetros



- Sobrecarga de métodos (*overloading*) não é possível em Python, em virtude da tipagem dinâmica de variáveis, porém é possível simular esse comportamento através de *Default Parameters*



Challenges Time



A hand holding a white chalk marker, drawing the word "THANK YOU" in large, bold, white letters on a blackboard. Surrounding the central text are various translations of "Thank You" in different languages, including: DĀKUJEM, GRACIAS, СПАСИБИ, תודה, DANKU, TAK, СПАСИБО, OBRIGADO, ACK, MER, DZIĘKUJĘ, ありがとう, БЛАГОДАРЯ ТИ, धन्यवाद, TAKK, 감사합니다, GO RAIBH MAITH AGAT, Ευχαριστώ, GRAZIE, and شکرا.



Facens

AQUI TEM ENGENHARIA