

Programação II

Classes e objetos

Hugo Pacheco

DCC/FCUP

21/22

Duas perspectivas

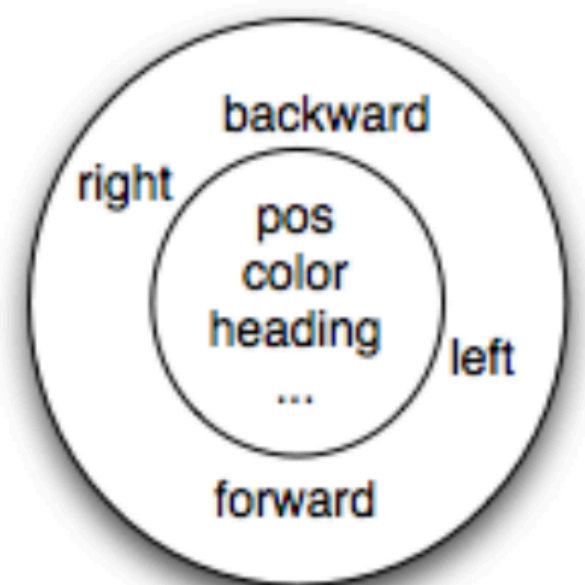
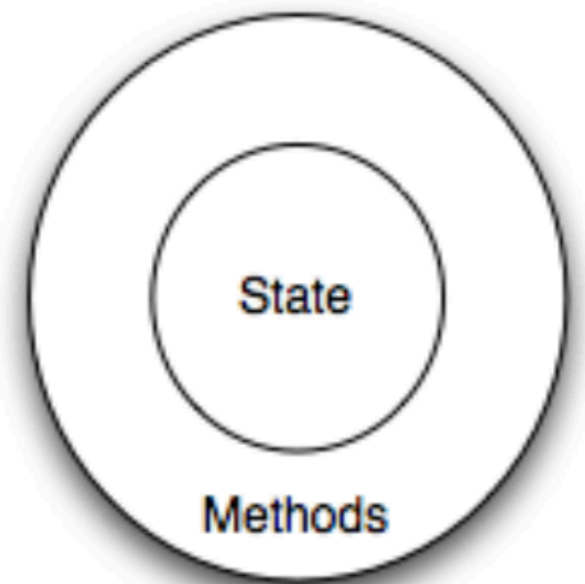
- Programação por procedimentos:
 - ênfase na modularização do código utilizando **funções**, que recebem argumentos e produzem resultados
 - Programação funcional (*map, filter, sum, etc*): funções não podem ter estado
 - Programação imperativa: funções podem alterar estado global
 - *drawCircle(tess)* = “Hey, drawCircle! Here’s a turtle object for you to draw with.”
- Programação orientada a objetos:
 - ênfase na representação de conceitos utilizando **objetos**, que contêm estado e funcionalidade próprios
 - *tess.circle()* = “Hey tess! Please use your circle method!”

Objetos

- Em Python, todos os valores são objetos: *Turtle*, *list*, *int*, etc
- Cada objeto é **mutável** e tem:
 - um **estado** interno — informação acerca do próprio objeto, e que o distingue dos outros
 - uma coleção de **métodos** — ações que o objeto pode executar, possivelmente alterando o seu estado
- Classes:
 - o tipo de objetos
 - Podemos criar vários objetos da mesma classe (e.g., duas tartarugas)

Objetos (Turtle)

- Classe: *Turtle*
- Construtor: *Turtle()*
- Objeto: *alex = Turtle()*
- Estado “escondido” (relação da tartaruga com a janela,...)
- Atributos = estado “visível” (*pos,color,...*)
 - ler *alex.color()*
 - escrever *alex.color('blue')*
- Métodos = funcionalidades:
 - avançar 10 unidades *alex.forward(10)*
- Documentação completa: [aqui](#)



Atributos (Python)

- *Estado “escondido” não existe*
- Todo o estado interno é visível = variáveis da classe
- Atributos = convenção = métodos para ler/escrever variáveis internas da classe

```
class Class:
    def __init__(self):
        self._attr = 0
    def attr(self, value=None):
        if value:
            self._attr = value
            return None
        else:
            return self._attr
```

```
obj = Class()
print(obj._attr)
obj._attr = 4
print(obj.attr())
obj.attr(value=3)
print(obj._attr)
```

Objetos (NumPy)

- Vamos utilizar várias bibliotecas que oferecem objetos avançados
- E.g., *NumPy* para manipulação eficiente de arrays multidimensionais
- Objeto esconde muita da complexidade

```
import numpy
# construtor
>>> x = numpy.array([[1, 4, 3], [5, 6, 2]])
>>> type(x)
<class 'numpy.ndarray'>
>>> x
[[1 4 3]
 [5 6 2]]

# atributos
>>> x.shape
(2, 3)
>>> x.dtype
dtype('int32')

# métodos e funções
>>> x.max()
6
>>> numpy.max(x)
6
>>> x.sort()
>>> x
[[1 3 4]
 [2 5 6]]
```

Novas classes

- Nesta cadeira vamos essencialmente utilizar objetos eventualmente **complexos** pré-definidos
- Em certos problemas, podemos querer criar novas classes **simples** para ajudar a estruturar a solução para além da definição de funções
- Veremos nesta aula alguns exemplos de como definir novas classes

Classe (Ponto 2D)

- Em duas dimensões:
 - ponto é caracterizado por dois números (coordenadas)
 - estado de um ponto: pode ser representado por (x, y)
 - $(0, 0)$ = origem
 - (x, y) = ponto x unidades à direita e y unidades para cima

Construtores (Ponto 2D)

- Cria uma nova classe *Ponto* com um construtor e 2 atributos
- Tal como para funções, pode-se documentar uma classe
- Atributos são variáveis internas da classes

```
class Ponto:
    """ Classe para representar pontos 2D """
    def __init__(self):
        """ Cria um novo ponto na origem """
        self.x = 0
        self.y = 0
```

```
p = Ponto()
print(p, p.x, p.y)
# <__main__.Ponto object at 0x10d74aa60> 0 0
p.x = 5
print(p, p.x, p.y)
# <__main__.Ponto object at 0x10d74aa60> 5 0
```

Construtores (Ponto 2D)

- Pode-se alterar os atributos diretamente, mas há uma forma mais elegante, definindo construtores com argumentos opcionais

```
class Ponto:
    """ Classe para representar pontos 2D """

    def __init__(self, x=0, y=0):
        """ Cria um novo ponto em coordenadas dadas """
        self.x = x
        self.y = y

p1 = Ponto()
p2 = Ponto(4, 5)

print(p1.x, p1.y, p2.x, p2.y)
# 0 0 4 5
```

Atributos (Ponto 2D)

- Recomenda-se que leitura/escrita de atributos seja feita através de métodos

```
class Ponto:
```

```
...
```

```
def getX(self): return self.x
```

```
def getY(self): return self.y
```

```
def setX(self,x): self.x = x
```

```
def setY(self,y): self.y = y
```

```
p = Ponto(4, 5)
```

```
print(p.getX())
```

```
# 4
```

```
p.setX(6)
```

```
print(p,p.x,p.y)
```

```
# <__main__.Ponto object at 0x10998ca60> 6 5
```

Funções (Ponto 2D)

- Podemos definir funções externas à classe

```
import math
```

```
class Ponto:
```

```
...
```

```
def distance(p1:Ponto,p2:Ponto):
```

```
    """Calcula distância entre 2 Pontos"""
```

```
    return math.sqrt ((p2.x-p1.x) ** 2 + (p2.y-p1.y) ** 2)
```

```
p1 = Ponto(1,1)
```

```
p2 = Ponto(5,5)
```

```
print(distance(p1,p2))
```

```
# 5.656854249492381
```

Métodos (Ponto 2D)

- Mas em alguns casos pode fazer mais sentido a função ser definida como um método da classe, i.e., vista como uma propriedade do objeto

```
class Ponto:
    ...

    def distanceToOrigin(self):
        return ((self.x ** 2) + (self.y ** 2)) ** 0.5

p = Ponto(5, 5)
print(p.distanceToOrigin())
# 7.0710678118654755
```

Métodos (Ponto 2D)

- Quando a função deseja alterar o estado interno do objeto, deve sempre ser definida como um método (**convenção**)

```
class Ponto:
    ...

    def resize(self, w):
        """ Multiplica um ponto por um fator """
        self.x *= w
        self.y *= w
```

```
p = Ponto(5, 5)
p.resize(2)
print(p.x, p.y)
# 10 10
```

Métodos (Ponto 2D)

- Uma função ou método pode por sua vez criar novos objetos

```
class Ponto:
    ...

    def halfway(self, target):
        """Calcula um ponto intermédio entre 2 pontos"""
        mx = (self.x + target.x) / 2
        my = (self.y + target.y) / 2
        return Ponto(mx, my)

p1 = Ponto(1, 1)
p2 = Ponto(5, 5)
p3 = p1.halfway(p2)
print(p3.x, p3.y)
# 3.0 3.0
```

Pretty printing (Ponto 2D)

- Por defeito, quando se imprime um objeto é mostrado apenas o seu nome

```
>>> p = Ponto()  
>>> print(p)  
<__main__.Ponto object at 0x1008da820>
```

- Podemos redefinir um método especial `__str__` que por convenção converte um objeto para string e é chamado pela função *print*

```
class Ponto:
```

```
...
```

```
def __str__(self):
```

```
    return "(" + str(self.x) + "," + str(self.y) + ")"
```

```
p = Ponto(5, 5)
```

```
print(p)
```

```
# (5, 5)
```


Igualdade de objetos

- Em Python, por defeito, dois objetos são iguais se apontarem para a mesma região de memória
- É a chamada **igualdade superficial**, que corresponde ao operador *is*

```
>>> p1 = Ponto(1,1)
>>> p2 = Ponto(5,5)
>>> p3 = p1
>>> p1 is p1
True
>>> p1 is p2
False
>>> p1 is p3
True
```

Igualdade de objetos

- O operador `==`, se não for redefinido, devolve a igualdade superficial

```
>>> p1 = Ponto(1,1)
>>> p2 = Ponto(1,1)
>>> p1 == p2
False
```

- Podemos redefinir o método especial `__eq__`

```
class Ponto:
```

```
...
```

```
    def __eq__(self, p):
        return self.x == p.x and self.y == p.y
```

```
p1 = Ponto(1,1)
p2 = Ponto(1,1)
print(p1 == p2)
# True
```

Igualdade de objetos

- Cuidado com a igualdade de objetos!
- Pode estar redefinida ou não

```
>>> l1 = [1, 2, 3]
>>> l2 = [1, 2, 3]
>>> l1 == l2
True
```

```
>>> d1 = {'a': 1}
>>> d2 = {'a': 1}
>>> d1 == d2
True
```

Métodos especiais

- A lista completa pode ser consultada [aqui](#)

- Inclui:

- Cópia de objetos

- Comparação de objetos

- Conversões de objetos

- ...

<code>__add__</code>	<code>self + other</code>
<code>__mul__</code>	<code>self * other</code>
<code>__truediv__</code>	<code>self / other</code>
<code>__lt__</code>	<code>self < other</code>
<code>__ge__</code>	<code>self >= other</code>
<code>...</code>	<code>...</code>