Programação I / Introdução à Programação

Apêndice D, "Classes and Objects"

João Pedro Pedroso

2024/2025

Última aula

- Mais tipos de dados em Python:
  - conjuntos (`set`, `fronzenset`)
- Módulo `collections`

Hoje:

- Programação orientada a objetos

Paradigmas de programação:

- *Programação imperativa:* o programa especifica como efetuar uma série de cálculos
  - instruções agrupadas em procedimentos → *procedural programming*
  - instruções associadas aos objetos sobre os quais os procedimentos vão atuar → *object-oriented programming*
- *Programação declarativa:* o programa especifica propriedades do resultado, não como as calcular
  - P. funcional → resultado é declarado como o valor de uma série de aplicações de funções
  - P. em lógica → resultado é declarado como a resposta a uma questão, tendo em conta uma série de *factos* e de *regras*
  - P. matemática → resultado é a solução de um problema de otimização

Programação orientada a objetos

- Python fornece suporte para programação orientada a objetos:
  - foco na criação de objetos, contendo simultaneamente
    - dados $\rightarrow$ atributos
    - funcionalidade $\rightarrow$ métodos
  - objetos no programa $\leftrightarrow$ objetos ou conceitos na realidade

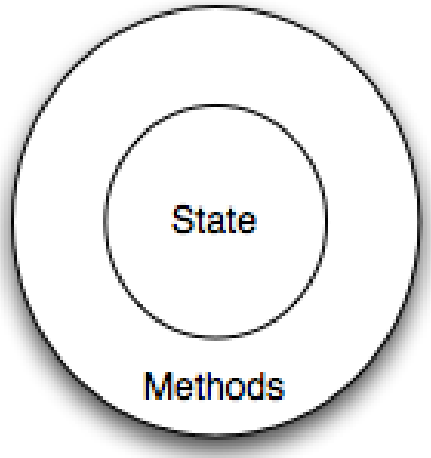
Programação orientada a objetos: Mudança de perspetiva

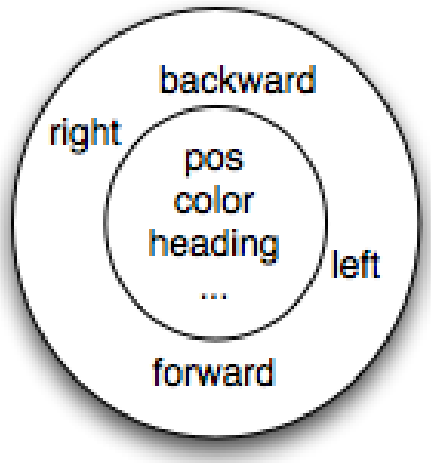
- Programação por procedimentos:
  - ênfase: escrita e utilização de funções
  - `drawCircle(tess)` → *"Hey, drawCircle! Here's a turtle object for you to use to draw with."*
  - agentes ativos: funções
- Programação orientada a objetos:
  - ênfase: representação da realidade/conceitos por objetos
  - `tess.circle()` → *"Hey tess! Please use your circle method!"*
  - agentes ativos: objetos
- Mudar a "responsabilidade" de funções para objetos:
  - torna programas mais versáteis
  - reutilização de código fica mais fácil
  - realidade ↔ classes/objetos.
- Nota: esta mudança de perspetiva nem sempre é óbvia ou útil

Objetos

- Em Python, todos os valores são objetos: `Turtle`, `list`, `int`
- Programas manipulam objetos:
  - utilizando-os em cálculos
  - executando métodos associados
- Cada objeto tem:
  - um **estado** — informação acerca do próprio objeto, e que o distingue dos outros
  - uma coleção de **métodos** — ações que o objeto pode executar

Estado e métodos: `Turtle`





Classes

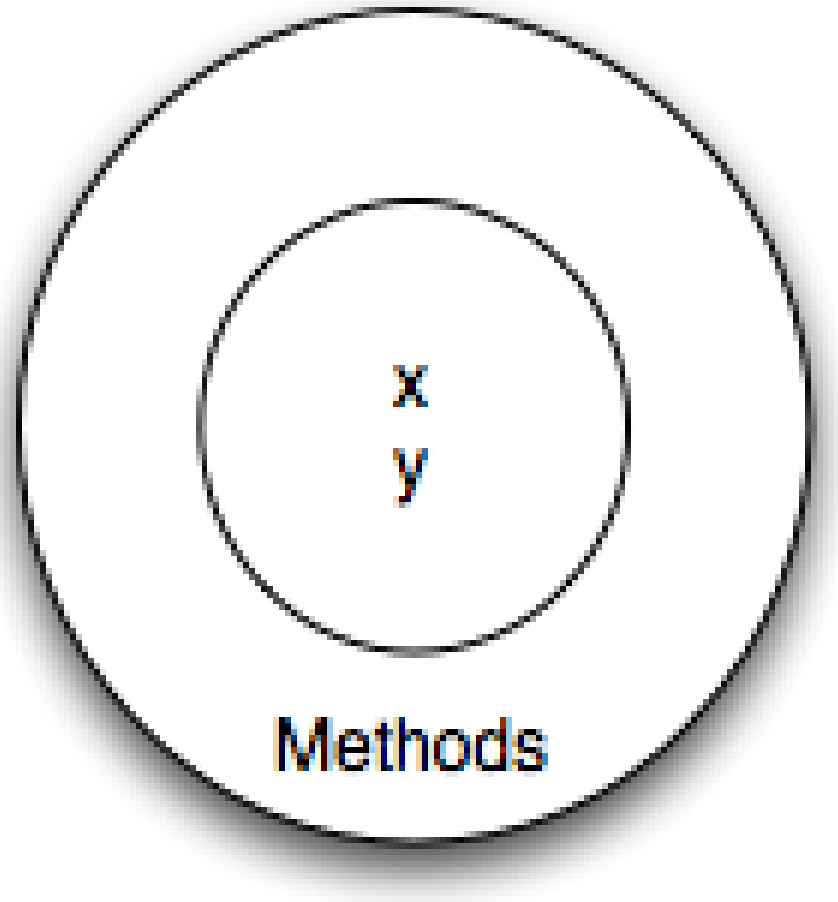
- Os objetos que utilizamos até agora estão predefinidos em Python
  - classes como `str`, `int`, `float`, `list`, `dict`, `Turtle`
- Na resolução de muitos problemas, queremos adaptar as estruturas de dados ao nosso problema
- Veremos de seguida como fazer isso com uma classe para representar *pontos no plano*

Classe para *pontos no plano*

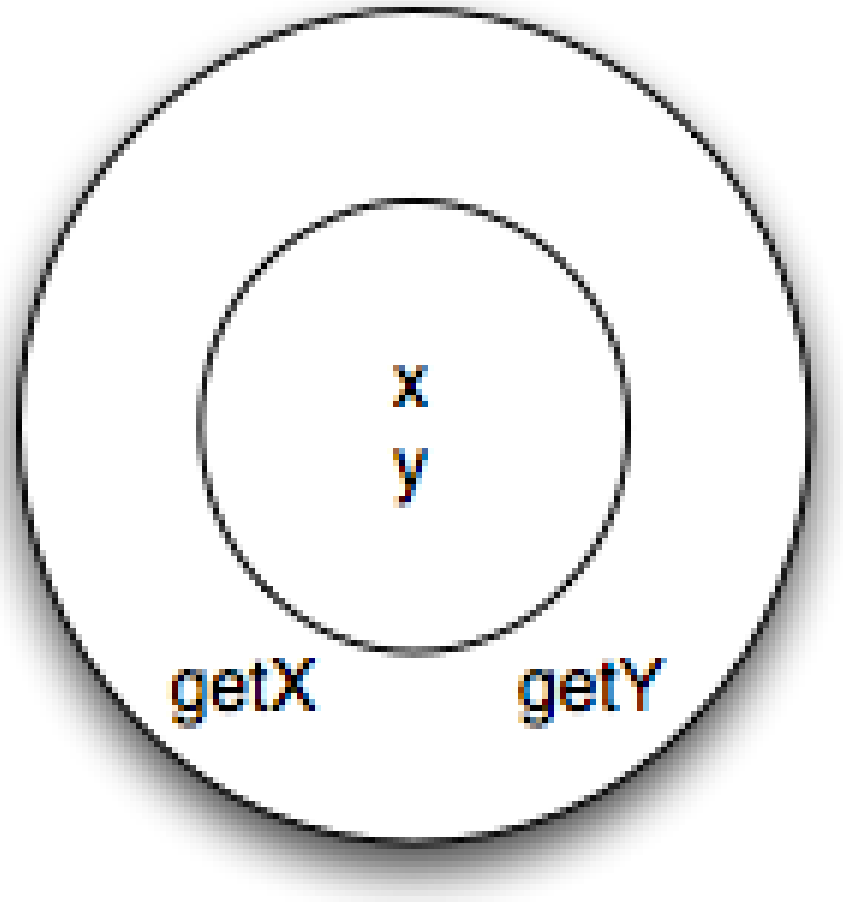
Em duas dimensões:

- ponto é caraterizado por dois números (coordenadas)
  - podem ser tratados como um único objeto
- exemplos:
  - origem: $(0, 0)$
  - $(x, y) \rightarrow$ ponto x unidades à direita e y unidades acima da origem
- estado de um ponto: pode ser representado por (x, y)

Estado e métodos: `Point`



Primeiros métodos da classe `Point`



Classes definidas pelo programador

Situação:

- Consideremos um ponto, na sua definição matemática, em duas dimensões: (x, y).
- Representação: forma mais simples é com dois valores em vírgula flutuante.
- Como os agrupar? Por vezes, uma *lista* ou um *tuplo* são as representações mais apropriadas
- Forma alternativa: usar um **tipo composto** definido pelo utilizador: uma **classe**
- Operações típicas: saber a coordenada x (`getX`) e a coordenada y (`getY`), calcular a distância à origem ou a outro ponto, ...

Primeira definição da classe

```
class Point:
    """Point class for representing and manipulating x,y coordinates."""

    def __init__(self):
        """ Create a new point at the origin """
        self.x = 0
        self.y = 0
```

- Com esta definição, criamos um novo tipo, `Point`
- Membros deste tipo são chamados **instâncias**, ou **objetos** desta classe
- Para criar objectos desta classe:

```
blank = Point()
```

- Esta instrução cria objetos de uma dada classe → construtor
  - utiliza o método `__init__`

Definição de uma classe

- Pode aparecer em qualquer parte de um programa, mas tipicamente aparece no início
- Regras sintáticas para definição de uma classe: as mesmas das outras instruções compostas
- Classe: *fábrica* de objetos
- Processo de construir um objeto e colocar os atributos no valor standard (“*factory default*”): instanciação
- A um objeto também se chama uma *instância* da classe

Sintaxe para definição de uma classe

- Cabeçalho: palavra reservada `class`, seguida do nome da classe
- Primeira linha depois do cabeçalho pode/deve ser uma *docstring* descrevendo a classe
- Método especial `__init__`:
  - utilizado para inicializar objetos da classe
  - é chamado automaticamente quando um novo objeto da classe é criado
  - permite atribuir valores a atributos desses objetos
- Parâmetro `self`:
  - sintaxe que nos permite aceder a atributos em qualquer parte do bloco da classe
  - utilizado para nos referirmos ao objeto com o qual estamos a trabalhar
  - praticamente todos os métodos de uma classe precisam de ter o objeto 'self' como primeiro argumento

Utilização desta classe

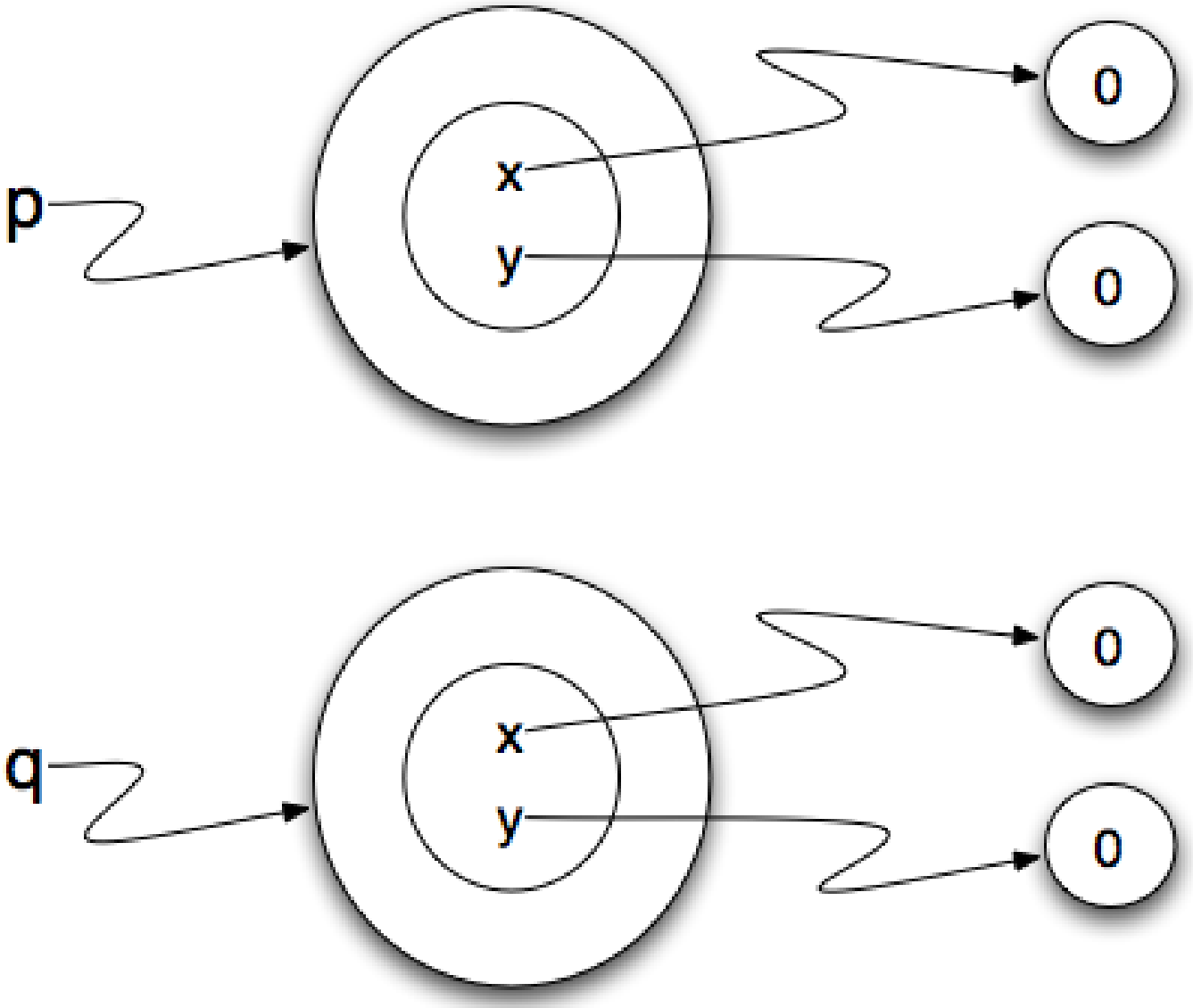
```
class Point:
    def __init__(self):
        self.x = 0
        self.y = 0
```

Depois de se definir a classe:

```
>>> p = Point()     # Instantiate an object of type Point
>>> q = Point()     # and make a second point
>>> p
<__main__.Point object at 0x1016cc860>
>>> q
<__main__.Point object at 0x1016cc400>
```

- Na inicialização criamos dois atributos: `x` e `y`, aos quais demos o valor zero;
- As variáveis `p` e `q` contêm referências para os novos objetos da classe `Point`;

Objetos da classe `Point`



Atributos

- Podemos aceder aos atributos de um objeto com:

```
>>> p.x = 7
>>> p.y = 6
```

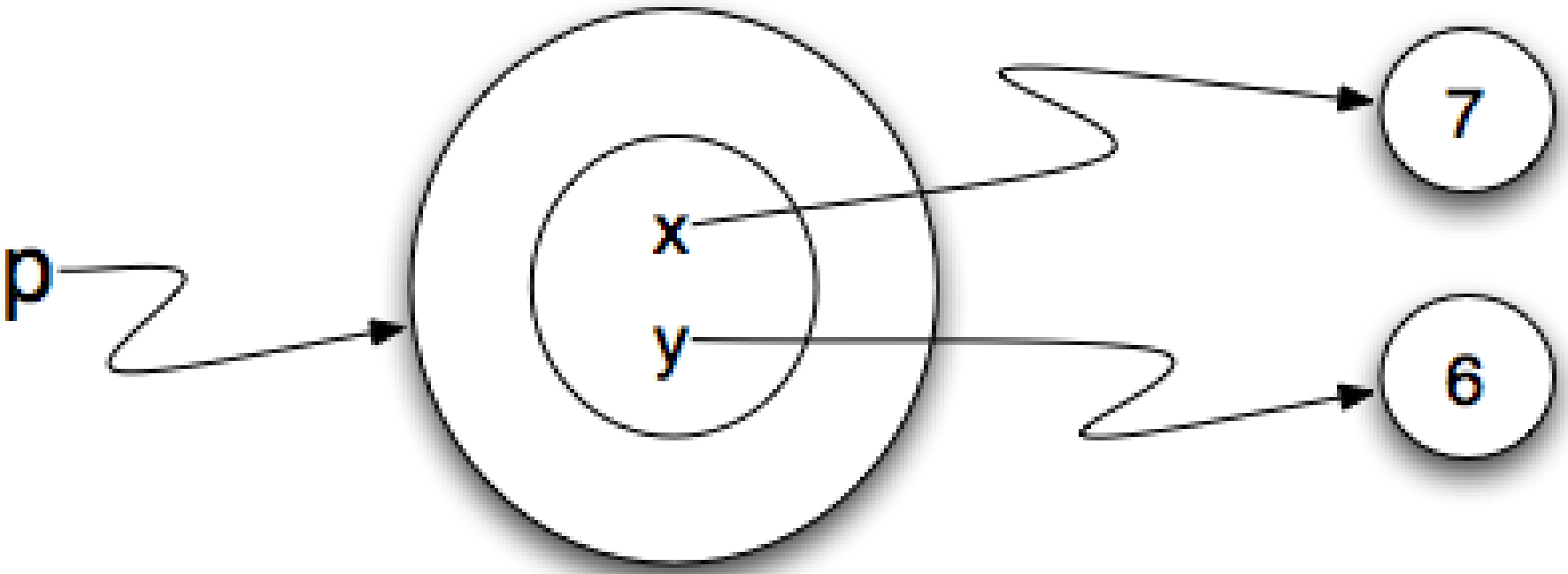
- No entanto, há uma forma mais elegante de fornecer estes valores na construção:

```
1  class Point:
2      def __init__(self, x=0, y=0):
3          self.x = x
4          self.y = y
```

- Instanciação:

```
>>> p = Point(7,6)
```

Instanciação



Acesso aos atributos

- Para aceder a estes valores, a sintaxe poderá ser:

```
>>> p = Point(7,6)
>>> p.y
6
>>> p.x
7
```

- `p.x` significa: ir ao objecto `p` e tomar o atributo `x`
- mas, mais uma vez: esta informação pode e deve ser acedida através de métodos

```
class Point:
    def __init__(self, x=0, y=0):
        """Create a new point at x, y"""
        self.x = x
        self.y = y

    def getX(self):
        return self.x

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

```
>>> p = Point(7,6)
>>> p.getX()
7
>>> p.getY()
6
```

Funções e métodos

- Podemos definir uma função exterior à classe `Point`:

```
def distanceFromOrigin(x, y):
    return (x * x + y * y) ** .5
```

```
>>> d = distanceFromOrigin(p.x, p.y)
>>> d
9.219544457292887
```

- Em muitos casos esta função fica melhor "arrumada" se for definida como um método da classe

```
1  class Point:
2      def __init__(self, x=0, y=0):
3          self.x = x
4          self.y = y
5
6      def getX(self):
7          return self.x
8
9      def getY(self):
10         return self.y
11
12     def distanceFromOrigin(self):
13         return ((self.x ** 2) + (self.y ** 2)) ** 0.5
```

```
>>> p = Point(7,6)
>>> p.distanceFromOrigin()
9.219544457292887
```

- Um método comporta-se como uma função, mas é invocado a partir de uma instância da classe.
- Invoca-se métodos utilizando *dot notation*, como acima.

Instâncias como argumentos e parâmetros

- Instâncias podem ser passadas a funções como parâmetros, na forma habitual:

```
1  def distance(point1, point2):
2      xdiff = point2.getX()-point1.getX()
3      ydiff = point2.getY()-point1.getY()
4
5      dist = (xdiff**2 + ydiff**2) ** 0.5
6      return dist
```

```
>>> p = Point(4,3)
>>> q = Point(0,0)
>>> distance(p,q)
5.0
```

- Função `distance`: argumentos são dois objetos da classe `Point`, devolve a distância entre eles
- Definido fora de `Point` → não é um método dessa classe
  - `self` não é um parâmetro desta função

Conversão de um objeto para `string`

- Quando se imprime um objeto da classe `Point` → resultado inesperado

```
>>> print(p)
<__main__.Point object at 0x1006e5b50>
>>>
```

- Podemos utilizar o método especial `__str__`:
  - chamado implicitamente por `print`:
  - responsável por devolver representação do objeto como uma *string*, numa forma definida pelo programador da classe
- Métodos especiais:
  - dois *underscores* antes e depois do nome
  - `__init__`, `__str__`, ...

```
class Point:
    def __init__(self, initX, initY):
    [...]
    def __str__(self):
        return "x=" + str(self.x) + ", y=" + str(self.y)
```

- Output:

```
>>> p = Point(7,6)
>>> print(p)
x=7, y=6
```

Sobrepor um método

- Python já tem uma definição da conversão para *string*
  - output misterioso que vimos acima

```
>>> print(p)
<__main__.Point object at 0x1006e5b50>
```

- Resultado não é o que pretendemos → redefinir esse método
- Nome técnico: sobrepor um operador (*override*)
- No caso concreto:

```
class Point:
    [...]
    def __str__(self):
        return "x=" + str(self.x) + ", y=" + str(self.y)
```

Instâncias como valores de retorno

- Funções e métodos podem devolver objetos;
- Exemplo: se quisermos calcular o ponto médio entre dois pontos, podemos definir:

```
class Point:
    [...]
    def halfway(self, target):
        mx = (self.x + target.x)/2
        my = (self.y + target.y)/2
        return Point(mx, my)
```

- Output:

```
>>> p = Point(3,4)
>>> q = Point(5,12)
>>> mid = p.halfway(q)
>>> print(p)
x=3, y=4
>>> print(q)
x=5, y=12
>>> print(mid)
x=4.0, y=8.0
```

Aula de hoje:

- Programação orientada a objetos

Próximas aulas

- Programação orientada a objetos (continuação)