Introdução a Python

IPL 2021

MIT BRAZIL

UNICAMP

Classes e métodos

- Vimos que funções podem operar em instâncias de classes que criamos da mesma forma que outros objetos
- Para que um tipo personalizado seja útil, é provável que escreveremos várias funções específicas a esse tipo que recebem pelo menos uma instância
- Apenas escrevendo as funções diretamente, não fica claro que todas operam em objetos de tipo específico: essa é a motivação para *métodos*

Relação entre funções e objetos Point não é explícita quando simplesmente definidas

```
class Point:
    n = 2


def distance_to_origin(pt):
    return (pt.x ** 2 + pt.y ** 2) ** 0.5


def euclidean_distance(pt1, pt2):
    return ((pt1.x - pt2.x) ** 2 + (pt1.y - pt2.y) ** 2) ** 0.5


def manhattan_distance(pt1, pt2):
    return abs(pt1.x - pt2.x) + abs(pt1.y - pt2.y)
```

Métodos

- Um **método** é uma função associada a uma classe específica
- Semanticamente, métodos são iguais a funções
- Sintaticamente, há duas diferenças principais:
  - Métodos são definidos *dentro de uma definição de classe* para tornar explícita a relação com a classe
  - A sintaxe para chamar um método é diferente

Criação de métodos torna explícita a relação com a classe Point

```
class Point:
    n = 2

    def distance_to_origin(pt):
        return (pt.x ** 2 + pt.y ** 2) ** 0.5

    def euclidean_distance(pt1, pt2):
        return ((pt1.x - pt2.x) ** 2 + (pt1.y - pt2.y) ** 2) ** 0.5

    def manhattan_distance(pt1, pt2):
        return abs(pt1.x - pt2.x) + abs(pt1.y - pt2.y)
```

Chamando métodos

- Uma vez que criamos métodos dentro de definições de classe, temos duas formas de chamá-los:

cls.method(inst)

Forma mais familiar: para uma instância **inst** da classe **cls**, procuramos a função **cls.method** e chamamos com **inst** como argumento

inst.method()

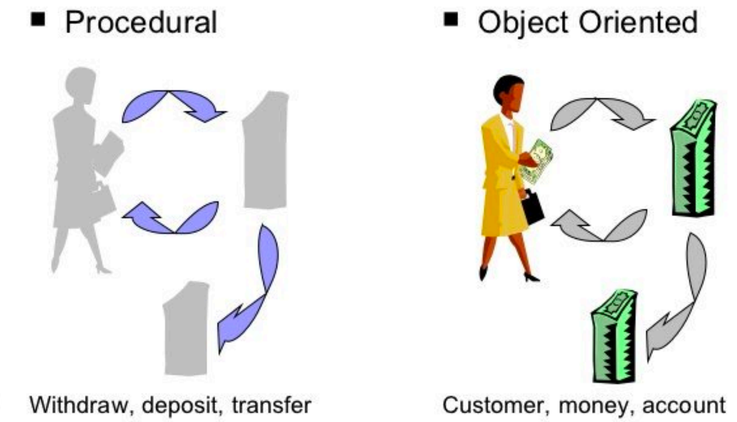
Forma mais comum: se um método for pesquisado em uma instância em vez de uma classe, *Python irá inserir automaticamente a instância como o primeiro argumento para o método*

- Essa sintaxe pode parecer estranha, mas estamos usando-a já! Para adicionar um elemento a uma lista, usamos **x.append(e)**, mas também poderíamos ter escrito **list.append(x, e)**

```python
class Point:
    n = 2

    def distance_to_origin(pt):
        return (pt.x ** 2 + pt.y ** 2) ** 0.5


p = Point()
p.x = 3.0
p.y = 4.0

print(Point.distance_to_origin(p))  # 5.0
print(p.distance_to_origin())  # 5.0
```

Mudança de paradigma

- O primeiro formato lembra de uma abordagem mais *procedural*, em que o foco está em executar uma sequência de procedimentos (operações/funções)
- O segundo formato passa a agência da ação ao objeto, dando código que é *object-oriented*: foco no conceito de objetos que contém dados e código (classes)



MIT BRAZIL UNICAMP

self

- Por convenção, o primeiro parâmetro de um método é chamado **self**
  - Note que **self** *é apenas um nome* normalmente usado, não uma palavra-chave nem um termo com propriedades especiais
  - Independentemente do nome, o primeiro argumento sempre denota *a instância que está sendo operada no momento*
- A ideia do nome é que o primeiro argumento é invariavelmente uma instância da classe / um caso específico da classe em si (*class it**self***)

Utilidade de *self*

- **self** é útil: permite *acessar e modificar atributos* dentro de um método
- Variáveis definidas dentro de uma função são acessíveis apenas nesse escopo e são deletadas ao fim da execução
- Contudo, os atributos definidos dentro de **self** podem ser modificados dentro de métodos e serão acessíveis posteriormente

other

- **other** é um nome comumente usado para representar uma segunda instância passada como argumento para um método

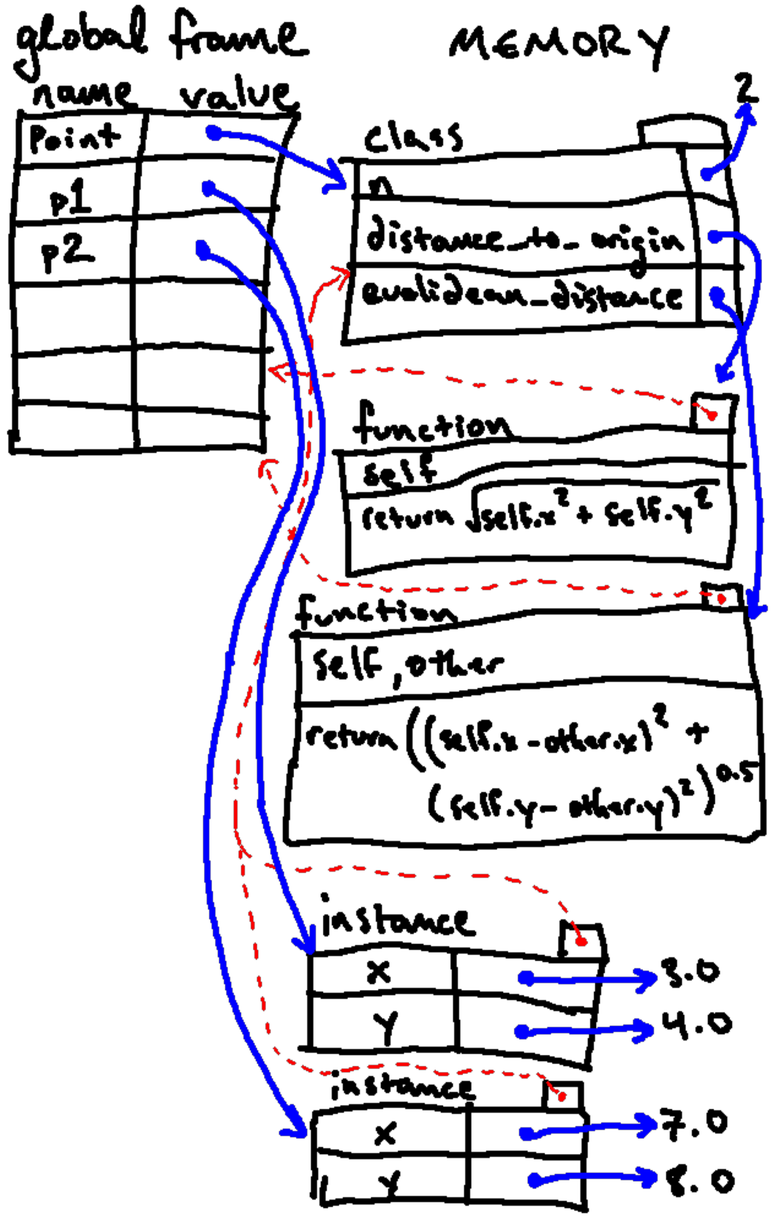
```python
class Point:
    n = 2

    def distance_to_origin(pt):
        return (pt.x ** 2 + pt.y ** 2) ** 0.5

    def euclidean_distance(pt1, pt2):
        return ((pt1.x - pt2.x) ** 2 +
                (pt1.y - pt2.y) ** 2) ** 0.5
```

Seguindo a convenção de parâmetros para métodos

```python
class Point:
    n = 2

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

    def euclidean_distance(self, other):
        return ((self.x - other.x) ** 2 +
                (self.y - other.y) ** 2) ** 0.5
```

Exemplo

```python
class Point:
    n = 2

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

    def euclidean_distance(self, other):
        return ((self.x - other.x) ** 2 +
                (self.y - other.y) ** 2) ** 0.5


p1 = Point()
p1.x = 3.0
p1.y = 4.0

p2 = Point()
p2.x = 7.0
p2.y = 8.0

print(p1.distance_to_origin())
print(p2.euclidean_distance(p1))
```



MIT BRAZIL

UNICAMP

Método **init** (construtor)

- Até agora, criar novas instâncias de classes que definimos era irritante; precisávamos criar a instância e então vincular atributos dentro do objeto resultante
- O método especial **init** é um *construtor* chamado quando um objeto é instanciado que facilita o processo
- Seu nome completo é **__init__** (dois *underscores*, **init**, mais dois *underscores*)

Interpretação de init

- Os argumentos passados durante a criação de uma instância *são passados para o método* **init** da classe
- No exemplo ao lado, Python:
  1. Cria uma instância de **Point** e associa a **p**
  2. Executa o método **__init__** com **p** passado como primeiro argumento (mesma ideia que chamar **Point.__init__(p, 4, 3)**)
  3. **self** se refere à instância, então o método armazena 4 e 3 como **x** e **y** dentro da instância

```
1   class Point:
2       n = 2
3
4       def __init__(self, x, y):
5           self.x = x
6           self.y = y
7
8
9   p = Point(4, 3)
10  print(p.x)  # 4
11  print(p.y)  # 3
12  print(p.n)  # 2
13
```

É comum que os parâmetros de **__init__** tenham os mesmos nomes dos atributos

Dunder methods

- Métodos dunder (também chamados de métodos “mágicos”) são marcados por underscores duplos, como **__init__**
- São comportamentos predefinidos que são geralmente chamados internamente a partir de operadores típicos
  - operador + chama **__add__**
  - a função integrada **len** chama **__len__**
  - o comparador > chama **__gt__**
- Definir dunders permite *emular o comportamento de tipos integrados* do Python em nossas classes
  - Permite o uso conveniente de operadores comuns para chamar funções “escondidas” / internas
- Alguns recursos com mais exemplos:
  - dbader.org/blog/python-dunder-methods
  - https://medium.com/python-features/magic-methods-demystified-3c9e93144bf7
  - https://docs.python.org/3/reference/datamodel.html#special-method-names


```python
class Point:
    n = 2

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        new_x = self.x + other.x
        new_y = self.y + other.y
        return Point(new_x, new_y)

    def __str__(self):
        return f"({self.x}, {self.y})"

    def __len__(self):
        """considerado como magnitude
        do vetor da origem ao ponto"""
        return int((self.x ** 2 +
                    self.y ** 2) ** 0.5)

    def __gt__(self, other):
        return len(self) > len(other)

```


MIT BRAZIL

UNICAMP

Alguns *dunders* comuns

__add__ Chamado internamente com operador +

__sub__ Chamado internamente com operador –

__mul__ Chamado internamente com operador *

__truediv__ Chamado internamente com operador /

__floordiv__ Chamado internamente com operador //

__pow__ Chamado internamente com operador **

__str__ Retorna uma string. Chamado quando a instância é impressa ou convertida para str

__len__ Retorna um int. Chamado com a função integrada **len**

__getitem__ Recebe um argumento **key**. Chamado com **inst[key]**

__setitem__ Recebe dois argumento **key, value**. Chamado com **inst[key] = value**

__gt__	__ge__	__eq__	__ne__	__lt__	__le__
>	>=	==	!=	<	<=

__bool__ Retorna um bool. Chamado quando o valor booleano do objeto precisa ser determinado

__contains__ Chamado internamente com operador **in**

__iter__ Chamado quando o objeto precisa ser iterado (e.g. **for _ in inst**)

__call__ Torna o objeto "chamável" como uma função, mesmo que uso seja raramente útil