Advanced Institute for Artificial Intelligence

Programação em Python

https://advancedinstitute.ai

Documentação de Código Python

Docstrings

Referências e Fontes das Imagens

- Google Python Style Guide
- Docstrings Styles
- The Hitchhiker's Guide to Python: Best Practices for Development (Book)

Docstrings

- Importante escrever a documentação da API que resuma claramente o que **cada objeto e método faz**.
- Manter a documentação atualizada é difícil;
- A melhor maneira de fazer isso é escrever **direto em nosso código**;
- Cada classe, função ou cabeçalho de método pode ter uma *string* na primeira linha após a definição;
- Essa linha deve estar no mesmo nível de indentação do conteúdo interno à definição;

```
1 >>> class Conta:
2 ...     'Essa classe representa informações sobre as contas dos clientes '
```

Docstrings - Visualizando a documentação

- A função `help()`e a variável especial `__doc__`

```
>>> help(Conta)
Help on class Conta in module conta:

class Conta(builtins.object)
 |  Essa classe representa um ponto no espaço bi-dimensional utilizando
     coordenadas geométricas

>>> Conta.__doc__
'Essa classe representa um ponto no espaço bi-dimensional utilizando
    coordenadas geométricas'
```

Docstrings - Comentários de Múltiplas Linhas

- Vários estilos, e.g., Estilo do Google

```
1  def transfere_para(self, destino, valor):
2      """Efetua uma transferência entre contas
3
4      Args:
5          destino (float): Conta destino.
6          valor (float): Valor a ser transferido.
7
8      Returns:
9          bool: Um Booleano informando se a transação aconteceu ou não
10     """
```

Programação Python

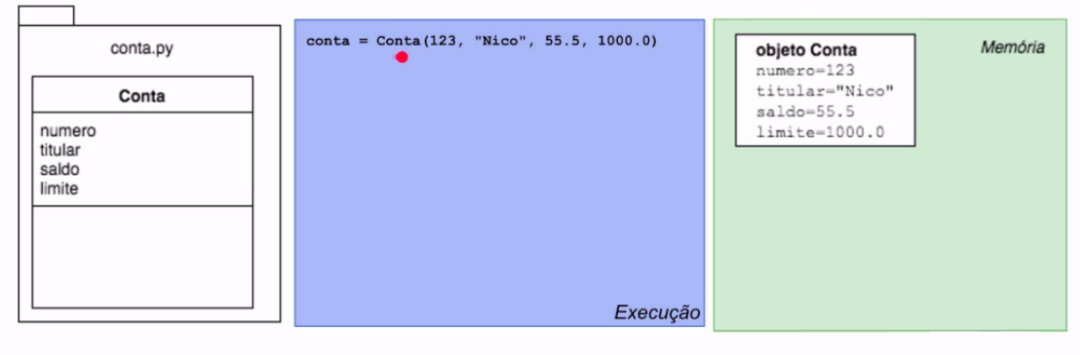
Encapsulamento, Herança e Polimorfismo

Referências e Fontes das Imagens

- Object-Oriented Programming and Python Tutorial
- Inheritance and Composition: A Python OOP Guide
- Python 3 Object Oriented Programming (Book)
- Learning Python: Powerful Object-Oriented Programming (Book)

Módulos

- A **Conta()** está dentro de um módulo que em algumas linguagens receberá o nome de *package* ou *namespace*, sendo que um módulo poderia ter uma ou mais sintaxes.



Encapsulamento

- **O Objetivo é "esconder" todos os membros de uma classe**
- Encapsular é fundamental para que seu sistema seja suscetível a mudanças
- O *underscore* alerta que ninguém deve modificar, nem mesmo ler, o atributo em questão.
- Uma classe deve ter apenas uma responsabilidade (ou deve ter apenas uma razão para existir)
- Em outras palavras, ela não deve assumir responsabilidades que não são delas

Métodos Getters e Setters

- É comum utilizarmos funcionalidades como estas para gerar relatórios, que nos mostre os dados principais da conta. Por serem recorrentes, existe uma nomenclatura padrão para esses método.

```python
# uma boa programação é adicionar para o objeto, um método para cada necessidade
def saldo(self):
    return self.__saldo

def devolve_titular(self):
    return self.__cliente

def retorna_limite(self):
    return self.__limite
```

Figure: Imprimindo resultados

Métodos Getters e Setters

- O uso do **getters** e **setters** são os primeiros conceitos aprendidos pelos desenvolvedores de linguagens como Java, C#, PHP. Métodos amplamente utilizados para retornar e modificar valores.

```python
# uma boa programação é adicionar para o objeto, um método para cada necessidade

def get_saldo(self):
    return self.__saldo

def get_titular(self):
    return self.__cliente

def get_limite(self):
    return self.__limite

def set_limite(self, limite):
    self.__limite = limite
```

Método Property

- Indicamos que este método representa uma propriedade, um termo já recorrente em outras linguagens, como *Delphi* e C#. Com @property, indicamos que estamos trabalhando com uma propriedade. Faremos isso com o método nome().

```
# uma boa programação é adicionar para o objeto, um método para cada necessidade

def get_saldo(self):
    return self.__saldo

def get_titular(self):
    return self.__cliente

def get_limite(self):
    return self.__limite

def set_limite(self, limite):
    self.__limite = limite
```

Herança

- **Toda classe que criamos já usa herança**
- Todas as classes são subclasses da classe especial chamada `object`
- Relação de Super-Sub Classe;
- Mecanismo de Reuso de Código;
- Definição de Comportamentos e Estados compartilhados entre instâncias;
- Requer uma **quantidade mínima de sintaxe extra** sobre uma definição de classe básica;
- O uso mais simples e óbvio de herança é **adicionar funcionalidade a uma classe existente**.
- Em geral, as classes podem **herdar, personalizar ou estender** o código existente em *superclasses*;

Herança

- Princípio de Substituição de Liskov
  - *Em um programa de computador, se **S** é um subtipo de **T**, então objetos do tipo **T** podem ser substituídos por objetos do tipo **S** sem alterar nenhuma das propriedades desejadas do programa.*
- E.g., uma classe `Eletronico` possui `TV` como subclasse
  - A classe `Eletronico` possui três operações: `ligar()`, `desligar()` e `esta_ligado()`
  - Instâncias da classe `TV` *herdam* as funcionalidade da classe `Eletronico`, evitando a duplicidade de código;

Importante

O ponto principal por trás de OOP em geral: programamos personalizando (especializando) o que já foi feito, em vez de copiar ou alterar o código existente;

Herança

```
In [2]: class Eletronico:
   ...:     def __init__(self):
   ...:         self.ligado = False
   ...:     def esta_ligado(self):
   ...:         return self.ligado
   ...:     def ligar(self):
   ...:         if not self.esta_ligado():
   ...:             self.ligado = True
   ...:     def desligar(self):
   ...:         if self.esta_ligado():
   ...:             self.ligado = False

In [3]: class TV(Eletronico):
   ...:     def mudar_canal(self, canal):
   ...:         self.canal = canal
```

Herança

```
In [4]: e1 = Eletronico()
In [5]: tv1 = TV()
In [7]: tv1.ligar()
In [8]: tv1.esta_ligado()
Out[8]: True
In [9]: tv1.mudar_canal(10)
In [10]: tv1.canal
Out[10]: 10
In [11]: e1.mudar_canal(5)
---------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-11-4028bef25be5> in <module>
----> 1 e1.mudar_canal(5)
AttributeError: 'Eletronico' object has no attribute 'mudar_canal'
```

Herança

- **Seguindo nosso exemplo:** A nomenclatura mais encontrada é que Funcionário é a superclasse de Gerente, e Gerente é a subclasse de Funcionário. Dizemos também que todo Gerente é um Funcionário. Outra forma é dizer que Funcionário é a classe mãe de Gerente e Gerente é a classe filha de Funcionário.

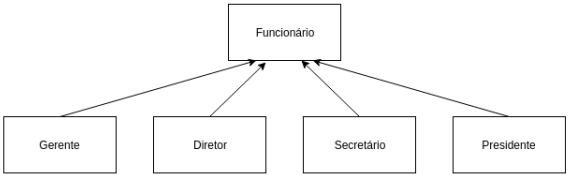


Figure: Herança

Estendendo métodos

- Acrescentar novas funcionalidades a um método definido na superclasse;
  - Função especial **`super()`** que faz referência a superclasse;

```
In [12]: class FormaGeometrica:
    ...:     def __init__(self, x, y):
    ...:         self.x, self.y = x, y

In [13]: class Retangulo(FormaGeometrica):
    ...:     def __init__(self,x, y, largura, altura):
    ...:         super().__init__(x,y)
    ...:         self.largura, self.altura = largura, altura

In [14]: ret = Retangulo(10, 20, 5, 5)
In [15]: ret.x, ret.y
Out[15]: (10, 20)
```

Sobrescrevendo Métodos

```
1  class FormaGeometrica:
2      def desenhe(self):
3          print(f"Iniciando Desenho no ponto ({self.x}, {self.y})")
4
5  class Retangulo(FormaGeometrica):
6      def desenhe(self):
7          print(f"Desenhando Retangulo com início no ponto ({self.x}, {self.
               y}), com L x A: {self.largura} x {self.altura}")
8
9  >>> ret1 = Retangulo(10, 20, 5, 5)
10 >>> ret1.desenhe()
11 'Desenhando Retangulo com início no ponto (10, 20), com L x A: 5 x 5'
```

Polimorfismo

- Comportamentos diferentes para mesma operação dependendo de qual subclasse é usada para instanciar o objeto.
- **Métodos com mesmos nomes** em classes e subclasses;
- Uma das razões principais para usar herança;
  - *Python duck typing*
    - O tipo do objeto é determinado por seu comportamento e estado;
- Fornecimento de uma **única interface** para entidades de diferentes tipos;

Polimorfismo

```
>>> f1 = FormaGeometrica(0,0)
>>> for forma in (f1, ret1):
...     forma.desenhe()

'Iniciando Desenho no ponto (0, 0)'
'Desenhando Retangulo com início no ponto (100, 200), com L x A: 10 x 10'
```

Encapsulamento

- Esconder detalhes de implementação de quem vai utilizar uma dada classe
- Variáveis "*privadas*" em Python:

```
1  class Ponto(FormaGeometrica):
2      def __init__(self, x=0, y=0):
3          self.__x, self.__y = x, y
4  >>> p1 = Ponto()
5  >>> p1.desenhe()
6  'Ponto localizado em (0, 0)'
7  >>> p1.__x
8  ---------------------------------------------------------------------------
       -
9  AttributeError
10 ...
11 AttributeError: 'Ponto' object has no attribute '__x'
```

Dúvidas?