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 Point:
2 ...     'Essa classe representa um ponto no espaço bi-dimensional
      utilizando coordenadas geométricas'
```

Docstrings - Visualizando a documentação

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

```
>>> help(Point)
Help on class Point in module __main__:

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

>>> Point.__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 move(self, x, y):
2      """Altera a posição do ponto no espaço
3
4      Verifica se a nova posição é diferente da atual. Caso seja, move o
           ponto para nova posição
5
6      Args:
7          x (float): Coordenada X.
8          y (float): Coordenada y.
9
10     Returns:
11         bool: Um Booleano informando se o movimento foi feito ou não
12     """
```

Programação Python

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)

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'
```

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:

```
class Ponto(FormaGeometrica):
    def __init__(self, x=0, y=0):
        self.__x, self.__y = x, y
>>> p1 = Ponto()
>>> p1.desenhe()
'Ponto localizado em (0, 0)'
>>> p1.__x
--------------------------------------------------------------------------
    -
AttributeError
...
AttributeError: 'Ponto' object has no attribute '__x'
```

Classes Abstratas

- Define trechos da classe que serão implementadas pelas subclasses;
- Não podem ser instanciadas;

```
1  from abc import ABCMeta, abstractmethod
2  class Poligono(metaclass=ABCMeta):
3      @abstractmethod
4      def calcula_area(self):
5          pass
6  >>> p1 = Poligono()
7  ----------------------------------------------------
8  TypeError
9  ...
10 TypeError: Can't instantiate abstract class Poligono with abstract method
       calcula_area
```

Programação Python

Exceções

Referências e Fontes das Imagens

- Python Exceptions: An Introduction
- Python 3 Object Oriented Programming (Book)
- Learning Python: Powerful Object-Oriented Programming (Book)

Exceções

- Seria ideal se o código sempre retornasse um resultado válido, mas **às vezes um resultado válido não pode ser calculado**.
  - Não é possível dividir por zero ou acessar o oitavo item em uma lista de cinco itens;
- Normalmente, as **funções tinham valores de retorno especiais** para indicar uma condição de erro:
  - Número negativo par indicar falha na execução e outros números para indicar outros erros;
- **Exceções** são objetos que sinalizam erros especiais
  - Que só precisam ser **tratados quando fizer sentido**.
  - **Muitas classes** de exceções disponíveis;
  - **Facilidade para definição** da nossa própria;

Exceções

- Herdam de uma classe interna chamada **`BaseException`**.
- Às vezes são indicadores de algo errado em nosso programa
- Também ocorrem em situações legítimas.

```
1  >>> x = 5 / 0
2  Traceback (most recent call last):
3  File "<stdin>", line 1, in <module> ZeroDivisionError: int division or
       modulo by zero
4
5  >>> lst = [1,2,3]
6  >>> print(lst[3])
7  Traceback (most recent call last):
8  File "<stdin>", line 1, in <module> IndexError: list index out of range
```

Exceções

- Lançando uma exceção
  - O fazer se um programa precisa informar o usuário ou uma função que as entradas são inválidas?
  - Utilizar a instrução **raise** seguida da classe que define a Exceção:

```
1  class Ponto():
2      def __init__(self, x, y):
3          if not isinstance(x, float) or not isinstance(y, float):
4              raise TypeError("Somente valores Reais")
5
6  >>> p1 = Ponto("a", "b")
7
8  TypeError: Somente valores Reais
```

Exceções

- Quando uma exceção é lançada, o fluxo de execução do programa é interrompido;
- a menos que a exceção seja tratada, o programa será encerrado com uma mensagem de erro;
- Se encontrarmos uma situação de exceção, **como nosso código deve reagir a ela ou se recuperar dela**?
  - Envolver qualquer código que possa lançar uma dentro de uma cláusula `try ... except`

Programação Python

```
1
2  def metodo_que_lanca_excecao():
3      print("Antes da Exceção")
4      raise Excepion("Sempre lançada")
5      print("Depois da Exceção")
6
7  >>> metodos_que_lanca_excecao()
8  Antes da Exceção
9  ---------------------------------------------------------------
10 Exception
11 ...
12 <ipython-input-68-dcf5b792b994> in metodo_que_lanca_excecao()
13 ...
14 ----> 3     raise Exception("Sempre lançada")
15 ...
16
17 Exception: Sempre lançada
```

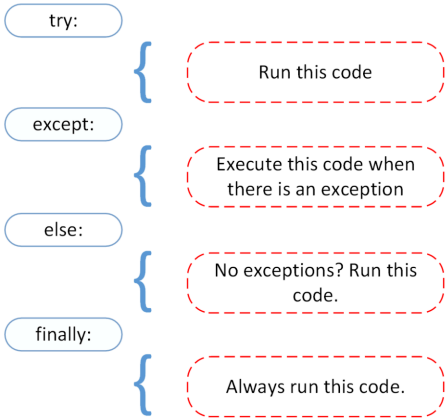
Exceções

```
1  try:
2      metodo_que_lanca_excecao()
3  except:
4      print("Exceção capturada")
5  print("Após execução do trech de Exceção")
6
7  'Antes da Exceção'
8  'Exceção capturada'
9  'Após execução do trech de Exceção'
```

Exceções - Formato Geral

```
1  try:
2      codigo_que_lanca_excecao()
3  except ValueError:
4      print("ValueError")
5  except TypeError:
6      print("TypeError")
7  except Exception as e:
8      print("Outro Erro")
9  else:
10     print("Código executado cajo não ocorra Exceção")
11 finally:
12     print("Código de limpeza que sempre é executado")
```

Exceções - Formato Geral



Dúvidas?