Advanced Institute for Artificial Intelligence

Programação em Python

https://advancedinstitute.ai

Programação Python

Definição de Interfaces e Encapsulamento

Referências e Fontes das Imagens

- Domine Decorators em Python
- Python Metaclasses
- Python 3 Object Oriented Programming (Book)
- Learning Python: Powerful Object-Oriented Programming (Book)

Decorators

- Ferramenta geral para **adicionar lógica que gerencia funções e classes**;
- Forma de executar **etapas de processamento extras** no momento da definição de funções e classes com **sintaxe explícita**;
- Por exemplo, podem ser usados para **aumentar funções com código que registra chamadas feitas a eles**, verifica os tipos de argumentos passados durante a depuração e assim por diante.
- Um decorador de função é codificado com um símbolo `@`, seguido pelo que chamamos de metafunção na linha acima da instrução `def`;
- **método para envolver uma função, modificando seu comportamento**

Decorators

```python
import time
def contador_de_tempo(funcao):
    def funcao_wrapper():
        inicio = time.time()
        funcao()
        fim = time.time()
        print(f"Tempo de Execução: {fim - inicio}")
    return funcao_wrapper

@contador_de_tempo
def main():
    for i in range(1, 10000):
        pass

main()
```

Classes Abstratas

- Define trechos da classe que serão implementadas pelas subclasses;
- Não podem ser instanciadas;
- Em Python, utilizam ***Metaclasses***

```
1  from abc import ABC, abstractmethod
2  class Poligono(ABC):
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
```

Classes Abstratas

- Definição de interfaces de programação de aplicações: APIs

```
1  class Quadrado(Poligono):
2      def __init__:
3      ...
4
5      def calcula_area(self):
6          return self.lado**2
7
8  >>> q1 = Quadrado()
9  >>> q1.calcula_area()
10 '4'
```

Programação Python

Encapsulamento

Referências e Fontes das Imagens

- Public, Private, and Protected – Access Modifiers in Python
- Python 3 Object Oriented Programming (Book)
- Learning Python: Powerful Object-Oriented Programming (Book)

Escondendo a Implementação

- Linguagens orientadas a objetos usam várias palavras-chave para **controlar e restringir o uso de recursos de uma classe**.
  - *public*, *private* e *protected*
- Python **não suporta proteção de acesso**
  - *"Somos todos adultos aqui."*
- Documente suas classes e insista para que seus colaboradores **leiam e sigam a documentação**.

Escondendo a Implementação - Convenções

- Acesso público:
  - Acesso por todos módulos/classes
  - Sem restrições ao nome dos membros da classe;
- Acesso protegido:
  - Acesso restrito a **própria classe e suas subclasses**
  - Nomes começam com **um *underscore*** (e.g., `_minha_variavel_protegida`)
- Acesso privado:
  - Restrito a própria classe
  - Nomes começam com **dois *underscores*** (e.g., `__minha_variavel_privada`
  - *Mangling* de variáveis/métodos

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
```

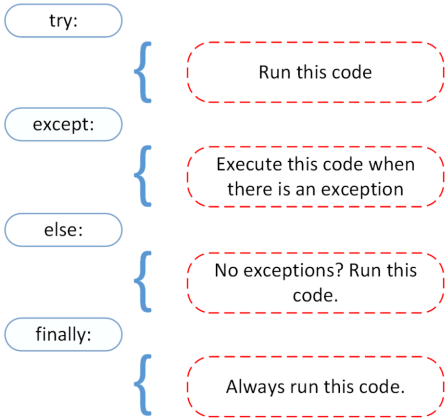
Exceções

```
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 trecho 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 caso não ocorra Exceção")
11 finally:
12     print("Código de limpeza que sempre é executado")
```

Exceções - Formato Geral



Exceções personalizadas

- Subclasse de algum tipo de *Builtin Exception*
- Podem ser organizadas em **Categorias** utilizando o **mecanismo de herança**

```
1  >>> class MyGeneralException(Exception): pass
2  >>> class MySpecificException1(MyGeneralException): pass
3  >>> class MySpecificException2(MyGeneralException): pass
4
5  >>> def raiser0(): raise MyGeneralException()
6  >>> def raiser1(): raise MySpecificException1()
7  >>> def raiser2(): raise MySpecificException2()
```

Exceções personalizadas - cont

```
>>> for func in (raiser0, raiser1, raiser2):
...     try:
...         func()
...     except MyGeneralException as e:
...         print(f"{e.__class__}")

<class '__main__.MyGeneralException'>
<class '__main__.MySpecificException1'>
<class '__main__.MySpecificException2'>
```

Dúvidas?