Trabalho Final

Thiago Laidler Vidal Cunha

Março 2021

1 Construção de Módulos em Python

Um módulo é um arquivo contendo definições e instruções Python, como se fosse um 'contêiner' contendo funções, classes e variáveis, que são reaproveitadas posteriormente. Durante o desenvolvimento de programas grandes, é uma boa prática dividi-lo em arquivos menores, para facilitar a manutenção.Além disso, pode-se usar um arquivo separado para uma função que seria escrita em vários programas diferentes, poupando o programador de copiar a definição de função em cada um deles. Para permitir isso, o Python tem uma maneira de colocar as definições em um arquivo e então usá-las em um script ou em uma execução interativa do interpretador.

O nome do arquivo é o nome do módulo escolhido acrescido do sufixo .py.

Exemplo

Criando módulo.py

```
def mult(n,m):
    return n*m

def soma(n,m):
    return n+m
```

Utilizando módulo.py

```
import modulo

modulo.mult(2,3)
6
modulo.soma(2,5)
7
```

Também é possível limitar as importações de apenas algumas funções do módulo escolhido, de forma que não é necessário chamar 'NomeDoModulo.função' sempre.

```
from modulo import mult

mult(2,3)
6
```

A utilização de módulos se torna vantajosa na feitura de grandes programas, em que dividimos um grande código em arquivos menores e mais fáceis de serem manipulados. Assim sendo, possibilita-se o desenvolvimento de sistemas cada vez maiores e mais complexos, aproimorando sua legibilidade.

2 Programação procedural vs. Programação orientada a objetos

Na programação estruturada (ou procedural), um programa é composto por três tipos básicos de estruturas: sequências (os comandos a serem executados), condições (sequências que só devem ser executadas se uma condição for satisfeita, como if-else, switch...), repetições (sequências que devem ser executadas repetidamente até uma condição for satisfeita, como for e while por exemplo). Essas estruturas são usadas para processar a entrada do programa, alterando os dados até que a saída esperada seja gerada.

No entanto, a diferença principal é que na programação estruturada, um programa é tipicamente escrito em uma única rotina (ou função) podendo ser quebrado em subrotinas. Mas o fluxo do programa

continua o mesmo, no final, só há uma grande rotina que executa todo o programa.

A programação orientada a objetos (POO) surgiu como uma alternativa a essas características da programação estruturada (PE). Trata-se de um modelo de programação onde diversas classes possuem características que definem um objeto na vida real. Cada classe determina o comportamento do objeto definido por métodos e seus estados possíveis definidos por atributos. São exemplos de linguagens de programação orientadas a objetos: Python, C++, Java, C, entre outras. Este modelo foi criado com o intuito de aproximar o mundo real do mundo virtual. Para dar suporte à definição de Objeto, foi criada uma estrutura chamada Classe, que reúne objetos com características em comum, descreve todos os serviços disponíveis por seus objetos e quais informações podem ser armazenadas.

Entres os paradigmas PE e POO, não existe certo e errado. A POO tende a dar melhores resultados em programas maiores com reuso de partes/sub-rotinas dos programas. Ambos os paradigmas possuem vantagens e desvantagens. A melhor prática é evitar extremismo: há casos em que é melhor priorizar a POO ou a PE, e mesmo quando uma estratégia é evidentemente melhor, o purismo tende a gerar software menos organizado ao custo de mais trabalho. Ademais, tratam-se de duas formas diferentes de organizar o código. No fim, a escolha é pessoal.

2.1 Classes, objetos, métodos e atributos:

A Programação Orientada a Objetos é formada por alguns itens, dentre os quais destacamos: Classes, Objetos, Atributos, Métodos, Construtores. Vejamos alguns exemplos de como elas são aplicadas nos códigos:

Exemplo de classe em Python

```
class Computer:
        def __init__(self, marca, ram,
        ↪  placavi):
                self.marca = marca
                self.ram = ram
                self.placa_video = placavi
                pass

comp1 = Computer('Asus','8gb','Nvidia')
comp2 = Computer('Dell','4gb','ATM')
comp3 = Computer('Acer','16gb','GeForce')
print(comp1.marca,comp2.marca,comp3.marca)
print(comp1.ram,comp2.ram,comp3.ram)
print(comp1.placavi,comp2.placavi,comp3.placavi)
```

Exemplo de métodos em Python.

```
class Computer:
        def __init__(self, marca,
        ↪  memoria_ram, placa_de_video):
                self.marca = marca
                self.memoria_ram =
                ↪  memoria_ram
                self.placa_de_video =
                ↪  placa_de_video
        def Ligar(self): ##Metodo1
                print('estou ligando')
        def Desligar(self): ##Metodo2
                print('estou desligando')
```

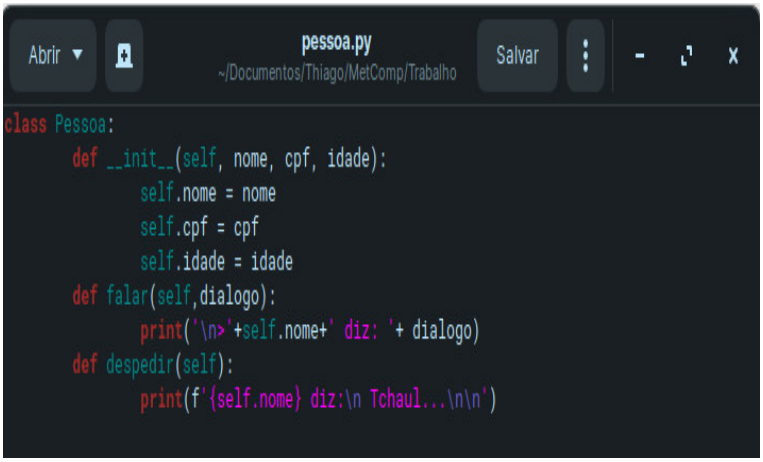

Figura 1: Criação de um arquivo (módulo) nomeado pessoa.py:

```
from pessoa import Pessoa #importando a classe dentro do modulo

Ge = Pessoa('Geralt de Rívia','11534635758','? anos')

print(Ge.nome)
print(Ge.cpf)
print(Ge.idade)
```

```
Geralt de Rívia
11534635758
? anos
```

Figura 2: Aplicando a classe 'Pessoa' do modulo 'pessoa'

2.2 Encapsulamento, Herança, Interface e Polimorfismo:

Encapsulamento é um dos pilares da programação orientada a objetos, segundo o qual é intuito do programador esconder dos usuários toda a informação não necessárias ao uso da classe.
Por exemplo, suponha que seja preciso criar uma classe para armazenar informações de funcionários de uma empresa, como ilustrado no exemplo abaixo:

```
class Funcionario:
    def __init__(self, nome, cargo,
    ↪  valor_hora_trabalhada):
        self.nome = nome
        self.cargo = cargo
        self.valor_hora_trabalhada =
        ↪  valor_hora_trabalhada
        self.horas_trabalhadas = 0
        self.salario = 0

    def registra_hora_trabalhada(self):
        self.horas_trabalhadas += 1

    def CalculaSalario(self):
        self.salario =
        ↪  self.horas_trabalhadas *
        ↪  self.valor_hora_trabalhada
```

Na classe acima, o salário de um funcionário é calculado com base no valor por hora trabalhada e na quantidade de horas trabalhadas. A classe, embora funcional, possui alguns problemas. Informações sigilosas de funcionários, como o salário, são expostas a clientes da classe, o que não é desejável.

Em Python, existe uma convenção de que dados ou métodos cujo nome começa com dois underscores não devem ser acessados fora da classe, como ilustrado no exemplo abaixo.

```
class Funcionario:
    def __init__(self, nome, cargo,
    ↪  valor_hora_trabalhada):
        self.nome = nome
        self.cargo = cargo
        self.valor_hora_trabalhada =
        ↪  valor_hora_trabalhada
        self.__horas_trabalhadas = 0
        self.__salario = 0

    def registra_hora_trabalhada(self):
        self.horas_trabalhadas += 1

    def CalculaSalario(self):
        self.__salario =
        ↪  self.__horas_trabalhadas *
        ↪  self.valor_hora_trabalhada
```

Repare que houve um adicionamento de dois underscores nas variáveis 'horas trabalhadas' e 'salario', sinalizando aos clientes da classe que essas variáveis são privadas.
Além disso, para o programador, encapsulamento permite que a implementação das funcionalidades da classe seja alterada sem que o código que usa a classe precise mudar. Em outras palavras, dado que a interface da classe (o que é exposto aos clientes) não mude, o programador tem a liberdade de mudar a implementação da funcionalidade que a classe oferece.
Por exemplo, caso a forma de cálculo do salário mude, basta que o programador altere a implementação do método CalculaSalario() e clientes da classe continuarão a usar o método sem precisar sofrer alterações.
Herança, no contexto de OPP, é o mecanismo pelo qual estende-se a funcionalidade de uma classe:

```
class Veiculo:
    def __init__(self, tipo, chassi, marca,
    ↪  modelo, ano):
```

```
        self.tipo = tipo
        self.chassi = chassi
        self.marca = marca
        self.modelo = modelo
        self.ano = ano
```

A classe acima representa um veiculo qualquer. Para que possamos estender a classe para abranger motocicletas, pode-se considerar dados sobre a cilindrada, por exemplo. Ou seja, para representar uma motocicleta, pode-se criar uma classe que herde o estado e comportamento da classe Veiculo e adicione a informação sobre a cilidrada.

```
1 class Motocicleta(Veiculo): #Classe
  ↪ motocicleta herda a classe veiculo
2     def __init__(self, tipo, chassi,
      ↪ marca, modelo, ano, cilindrada):
3         super().__init__(tipo, chassi,
          ↪ marca, modelo, ano) #Utiliza o
          ↪ metodo init da super classe
4         self.cilindrada = cilindrada
```

No exemplo acima, dizemos que a classe Veiculo é uma classe base ou super classe e que Motocicleta é uma classe derivada ou uma classe filha da classe Veiculo. Dizemos também que a classe Motocicleta herda da classe Veiculo (daí o nome herança) ou que a classe Motocicleta estende a classe Veiculo.

Polimorfismo, em Python, é a capacidade que uma subclasse tem de ter métodos com o mesmo nome de sua superclasse, e o programa saber qual método deve ser invocado, especificamente (da super ou sub). Ou seja, o objeto tem a capacidade de assumir diferentes formas.

Figura 3: Polimorfia World of Warcraft

Para exemplificar, criemos a classe Superclasse que tem apenas um método, o hello(). Instanciamos um objeto e chamamos esse método, onde o resultado será 'Olá, sou a superclasse!'. Em seguida, criemos outra classe (Sub) que vai herdar a Superclasse e vamos definir nela um método de mesmo nome hello(), mas com um texto diferente, assim como fizemos com as classes Veiculo e Motocicleta anteriormente :

```
class Super:
 def hello(self):
  print("Olá, sou a superclasse!")

class Sub (Super):
 def hello(self):
  print("Olá, sou a subclasse!")

teste = Sub()
teste.hello()
```

Assim, temos o resultado da segunda classe sendo alterada para 'Olá, sou a subclasse!', embora herde características da classe Super. Vemos que quando chamamos o método hello(), ele invoca o método da subclasse invés da superclasse. Percebe-se, portanto, que o método da subclasse se sobrepôs ao método da superclasse. O polimorfismo fica mais evidente no exemplo extrapolado abaixo:

```
    class Super:
 def hello(self):
  print("Olá, sou a superclasse!")

class Sub (Super):
 def hello(self):
  print("Olá, sou a subclasse!")

class Subsub (Sub):
 def hello(self):
  print("Olá, sou a subsubclasse!")

teste = Subsub()
teste.hello()
```

Ou seja, se a Subsubclasse herda a Sub, e a Sub herda a Super, então a Subsubclasse também herda tudo da Super. Porém, quando instanciamos um objeto da Subsub e invocamos o método hello(), o método da Subsub que será rodado.

Quanto a interface, em algumas linguagens de programação, o termo é uma referência à característica

que permite a construção de uma 'fronteira' que define a forma de comunicação entre programa e usuário, isolando do mundo exterior os detalhes de implementação de um componente de software.
A implementação de uma interface em Python é diferente de algumas linguagens como Java, Go e C++, que possúem codigos específicos para isso.
Para quem trabalha com desenvolvimento em Python, existem diversos frameworks e ferramentas que permitem a criação interfaces gráficas: WxWidgets, Tkinter, Kivy, PyGTK, PySide e QT. No caso do exemplo abaixo é usado Tkinter, devido a sua facilidade de uso e ao fato de acompanhar a instalação padrão da linguagem.

```
    from tkinter import *
class Application:
def __init__(self, master=None):
      self.widget1 = Frame(master)
      self.widget1.pack()
      self.msg = Label(self.widget1,
      ↪  text="Primeiro widget")
      self.msg.pack ()
root = Tk()
Application(root)
root.mainloop()
```

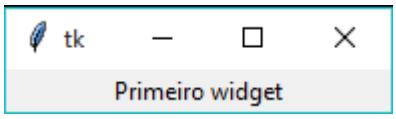

Figura 4: Criando primeiro widget

2.3 Vantagens e desvantagens

Quanto a POO, as vantagens são fáceis de serem indicadas: Software mais confiável (ao alterar uma parte nenhuma outra é afetada), manutenção mais próxima da realidade (cada classe atuando como algo real, facilita a organização), software extensível (o software não é estático, deve crescer para permanecer útil), código reutilizado mais facilmente (podemos usar o objeto de um sistema que criamos em outro sistema que viermos a criar). Entretanto, também há desvantagens como: Execução mais lenta (devido à complexidade do modelo, que traz representações na forma de classes), pior desempenho (devido a inumeros desvios na feitura do código) e mais dificil compreensão do código para terceiros (excesso de saltos, arquivos diferentes e desvios).
Quanto a PE, as vantagens mais evidentes são: melhoror clareza, qualidade e tempo de desenvolvimento do software, a localização das falhas e erros do programa são facilitadas assim como sua correção, os custos de manutenção são reduzidos e é melhorado o rendimento dos programadores.No entanto, acaba não sendo muito prático se tratando de programas muito grandes e complexos.

3 Ferramentas importantes para Astronomos

Uma forma rápida de pesquisar informações sobre qualquer objeto de estudo astronomico, exterior ao sistema solar, é atraves da base de dados Simbad, servindo mais ou menos como um 'google para astronomos'. É possível encontrar referencias de estudos sobre o objeto, coordenadas, imagens, medições e etc. No entanto, se o objetivo é procurar sobre um grupo de objetos, é mais útil procurar um catálogo específico, e para isso, deve-se utilizar a base de dados VizieR. Nele será listado diferentes opções de catálogos, com as informações mais específicas sobre cada objeto/aglomerado abertas pelo Simbad.

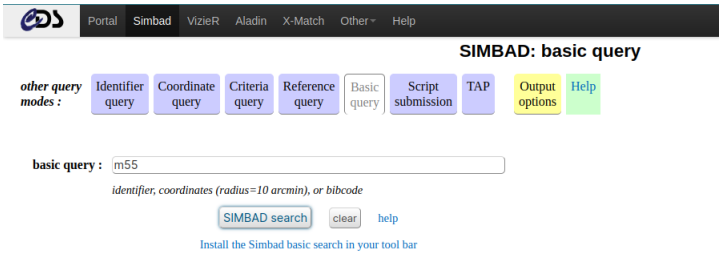

Figura 5: Pesquisando um aglomerado qualquer no Simbad.

Aladin é um visualizador do céu, bastante útil para estudo das imagens astronômicas, sendo possível alterar o contraste, navegar por entre inúmeras imagens interligadas segundo suas coordenadas. É também possível abrir as imagens de catálogos e procurar objetos específicos.

Figura 6: As informações encontradas em Simbard

Figura 8: Topcat

Topcat é uma ótima ferramenta para tratar gráficos, alterar suas configurações, aparência e informações sobre os eixos, resultando numa imagem em png facilmente utilizada em qualquer tipo de trabalho/apresentação.

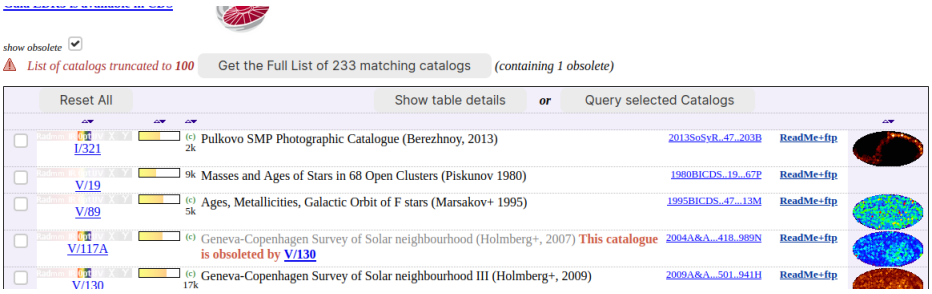

Figura 7: Exemplo sobre o resultado de uma pesquisa no VizieR