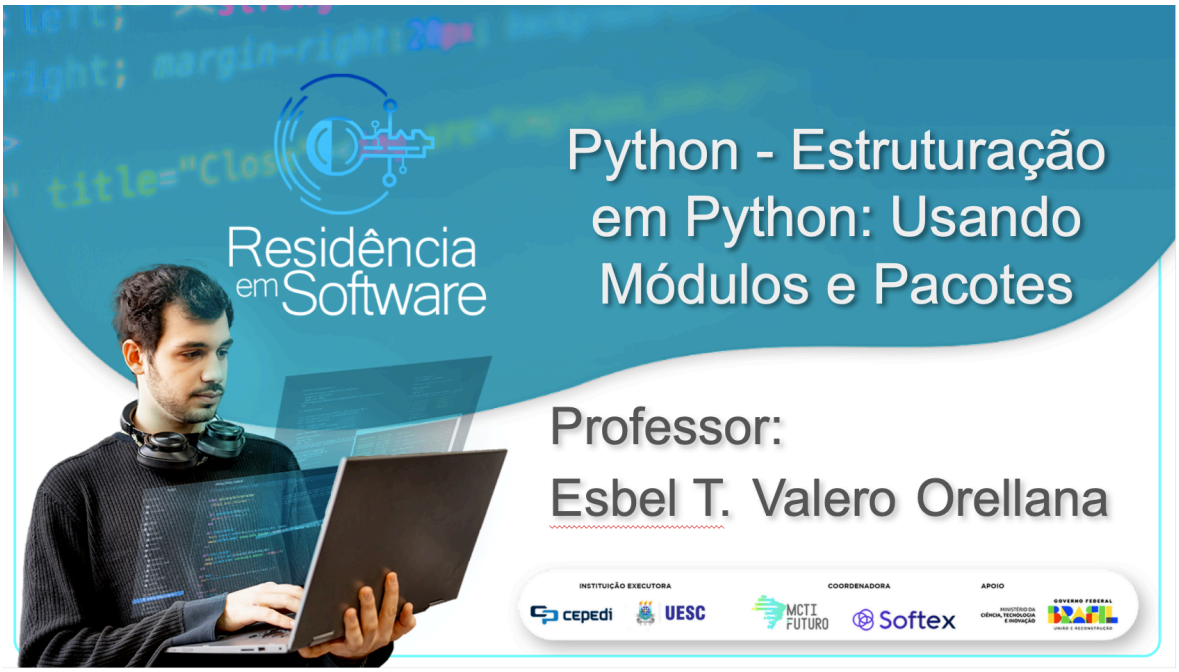
Módulo de Programação Python

Trilha Python - Aula 1: Estruturando um código em Python usando módulos e pacotes

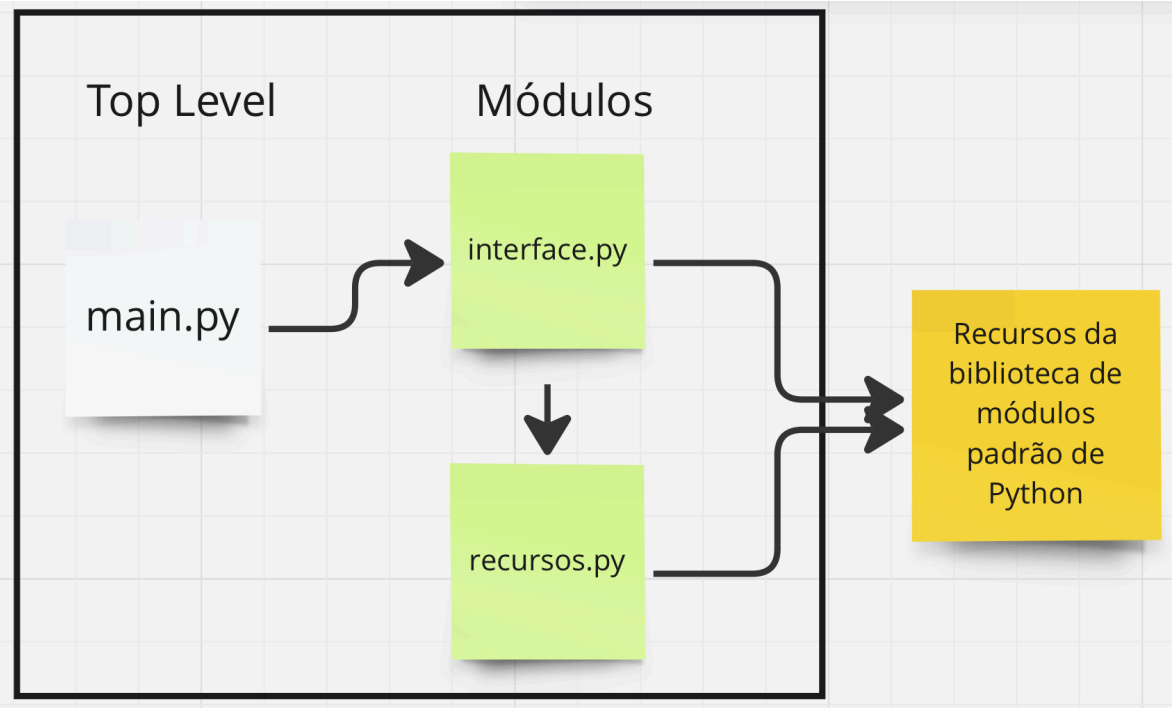


Objetivo: Introduzir o conceito de pacotes e módulos em Python e de estruturação do código com ajuda dos mesmos. Apresentar as formas de carregar um módulo e entender o caminho de busca. Discutir as implicações de cada uma das estratégias para importar um módulo nos escopos dos espaços de nomes (namespace) de cada módulo, em relação ao namespace do programa;

Módulos

Módulos em **Python** representam a unidade mais alta de organização de um programa, capas de armazenar códigos e dados para serem reutilizados. Os mecanismos disponíveis para carregar os recursos dos módulos foram pensados visando serem simples de usar e, na medida do possível, para minimizar conflitos de nomes.

- Um programa em Python consiste, basicamente, em um ou mais arquivos de texto contendo declarações Python.
- Um dos arquivo é o arquivo principal, que pode fazer uso ou não, de arquivos suplementares que são chamados de módulos. O arquivo principal passa a ser então o módulo `__main__` .



Como funciona o processo de importar um módulo?

- Procura-se o arquivo do módulo;
- Uma vez encontrado o arquivo é compilado para *byte code*;
  - Explore a pasta `__pycache__` onde estão os módulos que você está usando;
- Se executa o módulo para gerar os objetos nele definidos;

Onde e em qual ordem buscar pelos arquivos de módulos

1. Na pasta da aplicação;
2. No caminho do **PYTHONPATH**;
3. No caminho padrão das bibliotecas;
4. Nas pastas indicadas nos arquivos .pth;
5. Nos sítios definidos por bibliotecas de terceiros;

Podemos utilizar

a. Módulos da biblioteca padrão de **Pytho**;

b. Módulos desenvolvidos por terceiros;

c. Nossos próprios módulos.

Desta forna, criar módulos é algo que a maioria dos programadores **Python** faz o tempo todo, mesmo sem pensar nisso: Sempre que você salva um novo script **Python**, você cria um novo módulo.

Uma vez criado você pode, por exemplo, importar seu módulo em outro módulo ou executar ele com módulo `__main__` .

Um pacote é uma coleção de módulos relacionados de alguma forma. As coisas que você importa para seus scripts da biblioteca padrão são módulos ou pacotes. Nesta aula vamos aprender como criar módulos e pacotes.

Começamos pela criação de um módulo que é mais simples.

Nesta pasta criamos o arquivo `moduloMat.py` .

Os nomes dos módulos viram variáveis quando importados. Desta forma, na hora de escolher nomes dos seus arquivos de script **Python**, a eles se aplicam as mesmas restrições que a variáveis.

Os módulos podem ser importados de forma simples:

In [1]:
```
# variável de escopo global
soma = 0
# importando o módulo que implementa a função de soma
import moduloMat
print("type(soma) = ", type(soma))
print("type(moduloMat) = ", type(moduloMat))
print("type(moduloMat.soma) = ", type(moduloMat.soma))
a = 4
b = 2
# chamando a função soma do módulo
soma = moduloMat.soma(a, b)
print("A soma de", a, "e", b, "é", soma)
```

```
type(soma) =  <class 'int'>
type(moduloMat) =  <class 'module'>
type(moduloMat.soma) =  <class 'function'>
A soma de 4 e 2 é 6
```

Quando importamos um módulo com `import` atribuímos a uma variável todos os recursos declarados no mesmo.

A variável assume o mesmo nome do módulo. Para acessar seus recursos utilizamos o nome da variável seguido de ponto e o nome do recursos que desejamos utilizar.

Esta forma de importar separa, num espaço de nomes associado à variável, o espaço de nomes declarado no módulo, evitando conflitos com o espaço de nomes local.

Entretanto, utilizar o nome do módulo pode ser trabalhoso, sobre tudo quando utilizamos os recursos com muita frequência e o nome é longo e complexo. Neste caso podemos utilizar uma versão alternativa do `import` que define um alias .

In [2]:
```
# variável de escopo global
soma = 0
# importando o módulo que implementa a função de soma
import moduloMat as mm
print("type(soma) = ", type(soma))
print("type(mm) = ", type(mm))
print("type(mm.soma) = ", type(mm.soma))
a = 4
b = 2
# chamando a função soma do módulo
soma = mm.soma(a, b)
print("A soma de", a, "e", b, "é", soma)
```

```
type(soma) =  <class 'int'>
type(mm) =  <class 'module'>
type(mm.soma) =  <class 'function'>
A soma de 4 e 2 é 6
```

Também é possível importar os recursos de um módulo utilizando `from ... import` .

In [3]:
```
# variável de escopo global
soma = 0
print("type(soma) = ", type(soma))
# importando o módulo que implementa a função de soma
from moduloMat import soma
print("type(soma) = ", type(soma))
a = 4
b = 2
# chamando a função soma do módulo
novaSoma = soma(a, b)
print("A soma de", a, "e", b, "é", novaSoma)
```

```
type(soma) =  <class 'int'>
type(soma) =  <class 'function'>
A soma de 4 e 2 é 6
```

Vejam que, neste caso, trazemos uma varável do namespace do módulo para o namespace local. O exemplo anterior mostra que este tipo de abordagem pode criar conflitos de nomes. Com `from ... import` podemos importar vário recursos e até o namespace total do módulo.

In [4]:
```
# variável de escopo global
soma = 0
print("type(soma) = ", type(soma))
# importando o módulo que implementa a função de soma
from moduloMat import *  # importa todas as funções do módulo
print("type(soma) = ", type(soma))
a = 4
b = 2
# chamando a função soma do módulo
novaSoma = soma(a, b)
print("A soma de", a, "e", b, "é", novaSoma)
print("A multiplicação de", a, "e", b, "é", multiplicação(a, b))
```

```
type(soma) =  <class 'int'>
type(soma) =  <class 'function'>
A soma de 4 e 2 é 6
A multiplicação de 4 e 2 é 8
```

A utilização de módulos pode ser útil também para organizar e estruturar o desenvolvimento de um aplicativo. Veja a estrutura criada na pasta `meuApp` .

In [5]:
```
%ls meuApp
```

```
__main__.py   __pycache__/  interface.py  recursos.py
```

Repare que temos Três módulos e um deles tem o incomum nome `__main__.py` . Quando queremos empacotar uma aplicativo de forma que **Python** saiba qual script é módulo `__main__` , basta criar um script com este nome. Repare na implementação

dos módulos.

__main__.py

```
from . interface import *
from . recursos import cart2pol, pol2cart
def main():
    op = menu()
    while op != 3:
        if op == 1:
            x, y = entradaCartesiana()
            rho, theta = cart2pol(x, y)
            saídaPolar(rho, theta)
        elif op == 2:
            rho, theta = entradaPolar()
            x, y = pol2cart(rho, theta)
            saídaCartesiana(x, y)
        else:
            print('Opção inválida!')
        op = menu()


if __name__ == '__main__':
    main()
```

interface.py

```
1
2   def menu():
3       print('1 - Cartesianas para Polares')
4       print('2 - Polares para Cartesianas')
5       print('3 - Sair')
6       return int(input('Escolha uma opção: '))
7
8   def entradaCartesiana():
9       x = float(input('Digite o valor de x: '))
10      y = float(input('Digite o valor de y: '))
11      return x, y
12
13  def entradaPolar():
14      rho = float(input('Digite o valor de rho: '))
15      theta = float(input('Digite o valor de theta: '))
16      return rho, theta
17
18  def saídaCartesiana(x, y):
19      print('x =', x)
20      print('y =', y)
21
22  def saídaPolar(rho, theta):
23      print('rho =', rho)
24      print('theta =', theta)
```

recursos.py

```
1   import math  # Importando das biblioteca padrão de módulos do Python
2
3   def cart2pol(x, y):
4       rho = (x**2 + y**2)**0.5
5       theta = math.atan2(y, x)
6       return rho, theta
7
8   def pol2cart(rho, theta):
9       x = rho * math.cos(theta)
10      y = rho * math.sin(theta)
11      return x, y
```

Repare que no `__main__.py__` utilizamos um `.` no `import`. Isto se deve a que o módulo não está no caminho. Vamos falar sobre como isso mais para frente no curso. Por enquanto esta sintaxes especifica para procurar o pacote na pasta local do módulo.

Vamos executar o aplicativo!!!

Execute num terminal, na pasta deste notebook o comando: `python -m meuApp`

```
evalero@Jorel ~/W/G/ResTIC18_PythonBasico (main)> cd Notebooks/
evalero@Jorel ~/W/G/R/Notebooks (main)> python -m meuApp
1 - Cartesianas para Polares
2 - Polares para Cartesianas
3 - Sair
Escolha uma opção: 3
evalero@Jorel ~/W/G/R/Notebooks (main)>
```

Esta forma de executar módulos será utilizada outras vezes e é muito útil.

As vezes se faz necessário estruturar os recursos num conjunto de módulos que estão relacionados entre se. Nestes casos se pode adotar a estratégia de organizar os módulos em pacotes.

Pacotes

A principal diferença entre um módulo e um pacote é que um pacote é uma coleção de módulos e possui um arquivo `__init__.py`. Dependendo da complexidade do pacote, ele pode ter mais de um `__init__.py`. Vamos dar uma olhada em uma estrutura de pastas simples para tornar isso mais óbvio e, em seguida, criaremos um código para seguir a estrutura que definimos.

In [6]:
```
print("Na pasta raiz deste notebook tempos uma pasta pacote: ")
%ls
```

```
Na pasta raiz deste notebook tempos uma pasta pacote:
Aula-01.ipynb
Aula-02.ipynb
Aula-03.ipynb
Aula-04.ipynb
Aula-05.ipynb
Aula-06.ipynb
Aula-07.ipynb
Aula-08.ipynb
Aula-09.ipynb
Figuras/
IP-P001.ipynb
IP-P002.ipynb
IP-P003.ipynb
Notas Para um Curso de Introdução a POO com Python.ipynb
PP-P001.ipynb
README.md
__pycache__/
main.py
meuApp/
moduloMat.py
pacote/
script_001.py
testeModuloMat.py
```

In [7]:
```
print("Na pasta pacote temos os módulos:  classes e recursos")
print("Ainda temos a pasta testando")
%ls pacote
```

```
Na pasta pacote temos os módulos:  classes e recursos
Ainda temos a pasta testando
__init__.py  __main__.py  __pycache__/ classes.py   recursos.py  t
estando/
```

In [8]:
```
print("Na pasta testando temos o módulo:  testes")
%ls pacote/testando
```

```
Na pasta testando temos o módulo:  testes
__init__.py  __pycache__/ testes.py
```

Nas respectivas pastas do pacote tem os arquivos `__iniy__.py` . Vamos renomear estes arquivos inicialmente.

In [ ]:
```
print("Removendo o arquivo __init__.py da pasta pacote")
%mv pacote/__init__.py pacote/init.py
%ls pacote
print("Removendo o arquivo __init__.py da pasta testando")
%mv pacote/testando/__init__.py pacote/testando/init.py
%ls pacote/testando
```

Agora vamos implementar nossos módulos.

In [ ]:
```
%cat pacote/classes.py
```

In [ ]:
```
%cat pacote/recursos.py
```

In [ ]:
```
%cat pacote/testando/testes.py
```

Agora vamos utilizar o pacote

In [ ]:
```
import pacote
import pacote.testando
print(dir(pacote))
print(dir(pacote.testando))
```

Como não temos o script de inicialização do pacote a pasta vira um grande repositório de módulos difícil de utilizar.

In [ ]:
```
from random import randint
```

In [ ]:
```
import pacote.classes
import pacote.recursos
import pacote.testando.testes

pontos = []
for i in range(10):
    x = randint(0, 10)
    y = randint(0, 10)
    pontos.append(pacote.classes.Ponto((x, y)))
for p in pontos:
    print(p.distânciaAté(pacote.classes.Ponto((0, 0))))
pontosOrd = pacote.recursos.ordenaPontos(pontos)
for p in pontosOrd:
    print(p.distânciaAté(pacote.classes.Ponto((0, 0))))

#print(pacote.teste(pontos[0], pontos[1]))
print(pacote.testando.testes.teste(pontos[0], pontos[1]))
```

Os arquivos `__init__.py` permitem trabalhar o conceito de pacote ao integrar todos os módulos num único namespace. Vamos recuperar o arquivo da pasta raiz do pacote.

In [ ]:
```
%mv pacote/init.py pacote/__init__.py
```

In [ ]:
```
%cat pacote/__init__.py
```

Agora vamos usar os módulos do pacote de forma mais simples.

In [ ]:
```
import pacote
for item in dir(pacote):
    print(item)
```

In [ ]:
```
pontos = []
for i in range(10):
    x = randint(0, 10)
    y = randint(0, 10)
    pontos.append(pacote.Ponto((x, y)))
for p in pontos:
    print(p.distânciaAté(pacote.Ponto((0, 0))))
pontosOrd = pacote.ordenaPontos(pontos)
for p in pontosOrd:
    print(p.distânciaAté(pacote.Ponto((0, 0))))

print(pacote.teste(pontos[0], pontos[1]))
#print(pacote.testando.teste(pontos[0], pontos[1]))
```

Podemos testar usar a pasta testando como um sub-módulo, se isto for interessante para a estruturação do código. Neste caso podemos comentar va última linha do arquivo `__init__.py` , pasta rais e utilizar o arquivo `__init__.py` da pasta testando.

In [ ]:
```
%cat pacote/__init__.py
```

In [ ]:
```
%mv pacote/testando/init.py pacote/testando/__init__.py
%cat pacote/testando/__init__.py
```

In [ ]:
```
import pacote
import pacote.testando
print(dir(pacote))
print(dir(pacote.testando))
```