Aula 13 - Import

Nem todos os comandos do Python estão disponíveis de cara. Para alguns deles é necessário fazer **importações**. Há 3 tipos de importações que podem ser feitas:

- Importar bibliotecas internas do Python.
- Importar outros arquivos nossos, que possuam funções que criamos.
- Importar pacotes externos para serem instalados na nossa máquina.

Vamos começar importando bibliotecas já incluídas no próprio Python.

Importando biblioteca interna

Há uma lista interminável de bibliotecas já incluídas no próprio Python. Não dá para ensinar todas. O que é necessário saber sobre elas é que basta utilizar o comando `import nome_da_biblioteca` no início do código-fonte para que vários comandos já estejam disponíveis para utilização no seu algoritmo. Para o exemplo abaixo, vamos usar uma biblioteca chamada `time`. Vamos criar um simples programa que exiba algumas mensagens na tela. A diferença é que criaremos um atraso de alguns segundos para a execução de cada comando, o que pode ser muito útil para algumas situações. Para isso, iremos chamar a função `sleep()` usando a biblioteca `time`. No final, o comando ficará `time.sleep(n)`, onde `n` é o número de segundos que iremos atrasar o loop:

In [ ]:
```
# importando biblioteca
import time

saidas = ['está sendo executado.', 'executando...', 'ainda executando...', 'foi

for saida in saidas:
    print(f'Programa {saida}')
    time.sleep(3) # comando que atrasa o loop em 3 segundos
```

```
Programa está sendo executado.
Programa executando...
Programa ainda executando...
Programa foi executado com sucesso.
```

Importando parte da biblioteca

Algumas dessas bibliotecas são tão grandes que não compensa importá-las totalmente, já que pesaria muito no programa apenas para podermos utilizar uma função. Nesse caso, seria mais prudente importar apenas "parte" da biblioteca, usando o comando `from nome_da_biblioteca import nome_do_recurso`. Veja no exemplo abaixo, onde criamos um programa que exibe a data atual, não importa o dia em que ele é

executado. Usaremos o comando `today()` incluso em `date` da biblioteca `datetime` para isso:

In [ ]:
```
# importando apenas um recurso da biblioteca
from datetime import date

dia = date.today()
print(dia)
```

```
2024-05-01
```

Vamos repetir o programa, mas desta vez executando cada elemento da data de forma separada e organizada:

In [ ]:
```
from datetime import date

dia = date.today().day
mes = date.today().month
ano = date.today().year

print(f'Dia de hoje: {dia}.')
print(f'Mês atual: {mes}.')
print(f'Ano atual: {ano}.')
```

```
Dia de hoje: 1.
Mês atual: 5.
Ano atual: 2024.
```

Vamos recriar o programa mais uma vez, mas desta vez iremos exibir a data por extenso em português:

In [ ]:
```
from datetime import date

meses = ['Janeiro', 'Fevereiro', 'Março', 'Abril', 'Maio', 'Junho', 'Julho', 'Ag

dia = date.today().day
mes = date.today().month
ano = date.today().year

print(f'Dia de hoje: {dia} de {meses[mes - 1]} de {ano}.')
```

```
Dia de hoje: 1 de Maio de 2024.
```

Fazendo mais de uma importação

E que tal exibirmos todos os dias do mês atual? Podemos fazer isso usando a biblioteca `calendar` em conjunto com a biblioteca `datetime` :

In [ ]:
```
# importando a biblioteca de calendário
import calendar
from datetime import date

mes = date.today().month
ano = date.today().year

print(calendar.month(ano, mes))
```

```
      May 2024
Mo Tu We Th Fr Sa Su
       1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31
```

Importação de funções de outro arquivo que você criou

Caso o código do seu programa fique muito grande, você pode separar em arquivos diferentes, e importar as funções para o arquivo principal. Para isso, é necessário uma série de procedimentos antes de fazer o `import`. Vamos seguir o passo a passo para criar um programa que puxa de outro arquivo as funções para as operações mátemáticas, como equações do 1º grau e 2º grau, apenas para exemplificar. Siga:

1. Crie uma pasta para o seu projeto. Não se esqueça de criar um ambiente virtual Python, conforme ensinado na aula 04.
2. Dentro dessa pasta, crie um arquivo com a extensão **.py**. Sugestão: **equacoes.py**.
3. No arquivo **equacoes.py**, escreva o código-fonte abaixo:

In [ ]:
```python
# importa função matemática para calcular raiz quadrada
import math

# função que calcula equação do 1º grau
calcular_1_grau = lambda a, b: -b/a

# função que calcula equação do 2º grau
def calcular_2_grau(a, b, c):
    delta = b**2 - (4*a*c)

    if delta > 0:
        x1 = (-b + math.sqrt(delta))/(2*a)
        x2 = (-b - math.sqrt(delta))/(2*a)
        yield f'x1 = {x1}'
        yield f'x2 = {x2}'
    elif delta == 0:
        x = -b/(2*a)
        return f'x = {x}'
    else:
        return 'A equação não possui uma solução real.'
```

4. Esse arquivo irá reunir todas as funções que precisaremos. Agora, na mesma pasta, criaremos outro arquivo **.py**. Vamos chamá-lo de **main.py**.
5. Nesse novo arquivo **main.py**, iremos importar o arquivo **equacoes.py** através do comando `import equacoes` para fazer o cálculo da equação do 1º grau. Para chamar a função, é necessário chamar o nome do arquivo antes do nome da função, dessa forma `equacoes.calcular_1_grau(a, b)`. Veja abaixo:

In [ ]:
```python
# importa o arquivo de funções
import equacoes

# programa principal
a = 14
b = -2

print(f'Equação do 1º grau: {equacoes.calcular_1_grau(a, b)}')
```

```
Equação do 1º grau: 0.14285714285714285
```

6. No exemplo acima, estamos importando todo o arquivo desnecesssariamente, pois estamos utilizando apenas a função que calcula a equação do 1º grau. Podemos importar apenas a função que vamos usar, trocando o `import` por `from equacoes import calcular_1_grau`:

In [ ]:
```python
from equacoes import calcular_1_grau

a = 2
b = 5

print(f'Equação do 1º grau: {calcular_1_grau(a, b)}')
```

```
Equação do 1º grau: -2.5
```

7. Porém, queremos usar também a equação do 2º grau, mas podemos importar as duas funções. Veja:

In [ ]:
```python
from equacoes import calcular_1_grau, calcular_2_grau

a = 2
b = 5
c = 3
equacao_2_grau = calcular_2_grau(a, b, c)

print(f'Equação do 1º grau: {calcular_1_grau(a, b)}')

for x in equacao_2_grau:
    print(f'Equação do 2º grau: {x}')
```

```
Equação do 1º grau: -2.5
Equação do 2º grau: x1 = -1.0
Equação do 2º grau: x2 = -1.5
```

O if name main do Python

Quando temos um projeto com mais de um arquivo **.py**, é comum que o arquivo que contenha o código principal (geralmente é o arquivo chamado **main.py**) contenha uma estrutura iniciando com o seguinte comando: `if __name__ == '__main__:'`. É dentro dessa estrutura que está o algoritmo principal, que irá chamar as funções e exibir o resultado na tela. Essa estrutura existe para evitar que um outro arquivo importe essa estrutura para dentro dele. Ou seja, o `if __name__ == '__main__':` só vai ser executado quando você executar o código diretamente.

Não é algo complicado, é só você pensar na própria estrutura do If no Python, quando você executa um arquivo diretamente dentro dele essa variável name vai receber o valor main, caso contrário não recebe esse valor. Então quando for fazer a comparação dentro do If, se estiver rodando o código principal essa verificação é verdadeira e ele executa o conteúdo, caso contrário a verificação é falsa e o código é ignorado. Pense assim, o código que possui essa estrutura if name main está sendo executado à partir dele mesmo? Se sim, vamos executar o conteúdo dessa estrutura.Caso contrário ele estará sendo executado a partir de outro código, portanto a estrutura if name main não será executada. Veja no exemplo abaixo. Digamos que ele é um arquivo chamado **main.py**:

In [ ]:
```python
# função
def calcular_retangulo(b, a):
    retangulo = b * a
    return retangulo

# algoritmo principal
if __name__ == "__main__":
    b = int(input('Informe o valor da base do retângulo: '))
    a = int(input('Informe o valor da altura do retângulo: '))

    print(f'Área do retângulo: {calcular_retangulo(b, a)}')
```

```
Área do retângulo: 15
```

Dessa forma, se eu for importar o arquivo **main.py** para outro arquivo, ele irá importar `calcular_retangulo`, mas não conseguirá importar o que está dentro de `if __name__ == '__main__':`.

Obs: o ideal é sempre colocar essa estrutura no seu código-fonte principal.

Como funciona?

Ao rodar qualquer arquivo em Python ele automaticamente, antes de tudo, define a variável **name**, que é uma variável reservada do Python. Então quando fazemos uma execução direta, executando o próprio arquivo, ele atribui o valor "**main**" para essa variável. Isso quer dizer que, estamos executando o arquivo de forma direta, ou seja, abrimos o arquivo e estamos executando ele. Dessa forma a variável **name** recebe o valor **main**. No entanto, quando importamos esse arquivos e executamos através de um segundo arquivo, essa variável recebe outra informação.

Então ao fazer a comparação de **name** = "**main**" ela será falsa e, portanto, não vai executar o conteúdo dessa estrutura. Por esse motivo quando executamos o código diretamente no arquivo ele lê todo o conteúdo da estrutura e quando executamos através da importação do arquivo ele não lê o que está na estrutura. Dessa forma o valor atribuído a variável **name** é diferente de "**main**"! Isso é interessante quando você vai fazer algum teste no seu arquivo principal, mas não quer que ele seja executado em arquivos secundários, então pode colocar alguma função que só funcione no arquivo principal.

Fonte

- https://www.hashtagtreinamentos.com/if-name-main-no-python?gad_source=1&gclid=CjwKCAjw88yxBhBWEiwA7cm6pYljtaawzEmxW4Q79CX7apbBq18j-DNKsNDqM-0OKQvFBbGGaEKp3hoCbBcQAvD_BwE

Importação de pacotes externos

Um dos procedimentos mais comuns em programação com Python é a importação de pacotes externos, ou seja, de bibliotecas que não vem instaladas no Python por padrão, exigindo a instalação deles na máquina do desenvolvedor. Existem n bibliotecas muito famosas para Python disponíveis na Internet, muitas deles extremamente famosas e muito exigidas no mercado de trabalho, e para todo tipo de tarefa que imaginar. Entre as mais famosas, podemos citar o pandas (análise de dados), o django (desenvolvimento web), o scikit Learn (machine learn), a open cv (detecção de imagens), o pytorch (redes neurais), flet (para aplicativos web, desktop e mobile), entre outros.

Nesses casos, a biblioteca precisa ser instalada no PC antes de ser usada. Mas não precisamos acessar sites, baixar instaladores ou arquivos compactados e abrir executáveis, nem descompactar nada dentro de pasta alguma: quando instalamos o Python lá no início do curso (aula 03), foi instalado junto um gerenciador de pacotes chamado **pip**. Ele permite baixarmos e instalarmos o pacote que quisermos diretamente do **CLI** (linhas de comando do terminal/prompt). Precisamos para isso apenas do comando de instalação do Python `pip install nome-da-biblioteca` no terminal/prompt na pasta do projeto.

Vamos testar o `pip install` em um novo projeto. Desta vez, iremos montar uma simples barra de progresso. Para isso, precisaremos da biblioteca `tqdm`:

1. Abra o terminal na pasta do seu projeto.
2. No terminal, digite `pip install tqdm`, e aguarde a instalação.
3. Crie um arquivo **.py** e digite o código-fonte abaixo:

In [ ]:
```
# importa a biblioteca externa tqdm e a interna time
from tqdm import tqdm
import time

# exibe a barra
for i in tqdm(range(100)):
    time.sleep(0.05)
```

```
100%|██████████| 100/100 [00:05<00:00, 19.63it/s]
```

Obs: lembra da aula 04, onde aprendemos a configurar um **ambiente virtual Python** criando uma pasta ***.venv***? Um dos motivos para a criação desse ambiente virtual é a aula atual. O pacote externo será instalado no ambiente virtual configurado para o projeto. Sem isso, ele não conseguirá instalar o pacote, ou usará o Python global (o que foi instalado no PC) para a instalação do pacote, fazendo com que seu computador fique mais pesado que o necessário para o projeto.

Interessante né?! Que tal criarmos um programa com uma finalidade um pouco mais importante?! Vamos criar um programa que recebe um valor de uma moeda e converte para outra, utilizando a cotação do dia. Para isso, vamos utilizar a biblioteca `forex_python`, que possui uma função chamada `CurrencyRates()`, utilizada exatamente para este fim. Crie um arquivo com o código-fonte abaixo:

In [ ]:
```
# importa a biblioteca de conversão de valores que contém a cotação atual de vár
from forex_python.converter import CurrencyRates

# recebe o valor da moeda a ser convertida
valor = str(input('Informe o valor a ser convertido ou Enter para encerrar o pro
valor = float(valor)

moeda_origem = input('Digite a moeda de origem (exemplo: USD, EUR, BRL): ').uppe
moeda_destino = input('Digite a moeda de destino (exemplo: USD, EUR, BRL): ').up

# faz a conversão
resultado = CurrencyRates().convert(moeda_origem, moeda_destino, valor)

# exibe o resultado na tela
print(f'$ {valor:,.2f} {moeda_origem} = $ {resultado:,.2f} {moeda_destino}.')
```

```
$ 1.00 USD = $ 4.97 BRL.
```

Alias

Alias (ou apelidos) são nomes que podem substituir o nome dos pacotes, quando estes ficam muito grandes no código. Não é obrigatório, mas facilita na hora de codificar.

Por exemplo, vamos pegar a biblioteca ***numpy***, utilizada para trabalhar com *arrays* e *matrizes*, e criar um pequeno código apenas para verificar a versão do numpy instalada. Para isso, iremos importar a biblioteca através do comando `import numpy as np`. Caso essa biblioteca não esteja instalada, podemos instalar através do `pip install numpy`.

In [ ]:
```
# importando uma biblioteca e atribuindo a ela um apelido
import numpy as np

# chamando o comando da biblioteca pelo apelido
print(np.__version__)
```

```
1.26.4
```

Como pode observar, essa biblioteca pode ter seu nome trocado nos comandos para facilitar a digitação do comando. Nesse caso, trocamos o nome `numpy` por `np`.

Requirements.txt

Pode haver a necessidade de mover o projeto da sua máquina atual para outra. Se isso acontecer, obviamente os pacotes instalados não moverão junto, o que o obrigará a

instalar os pacotes de novo. Porém, há uma forma de salvar os pacotes instalados e movê-los junto com o seu projeto:

1. Abra o terminal na pasta do projeto.
2. No terminal, digite o seguinte comando: `pip freeze > requirements.txt`. Isso fará com que o Python salve na pasta do seu projeto um arquivo **.txt** com a lista de todos os pacotes instalados e a versão de cada pacote.
3. Mova o seu projeto junto com esse arquivo.
4. Quando o seu projeto já estiver na nova máquina, configure o ambiente virtual Python novamente, como se fosse um novo projeto.
5. Abra o terminal na pasta do seu projeto na nova máquina, e digite o comando: `pip install -r requirements.txt`. Isso fará com que todos os pacotes instalados na máquina original sejam instalados na pasta do projeto na nova máquina.