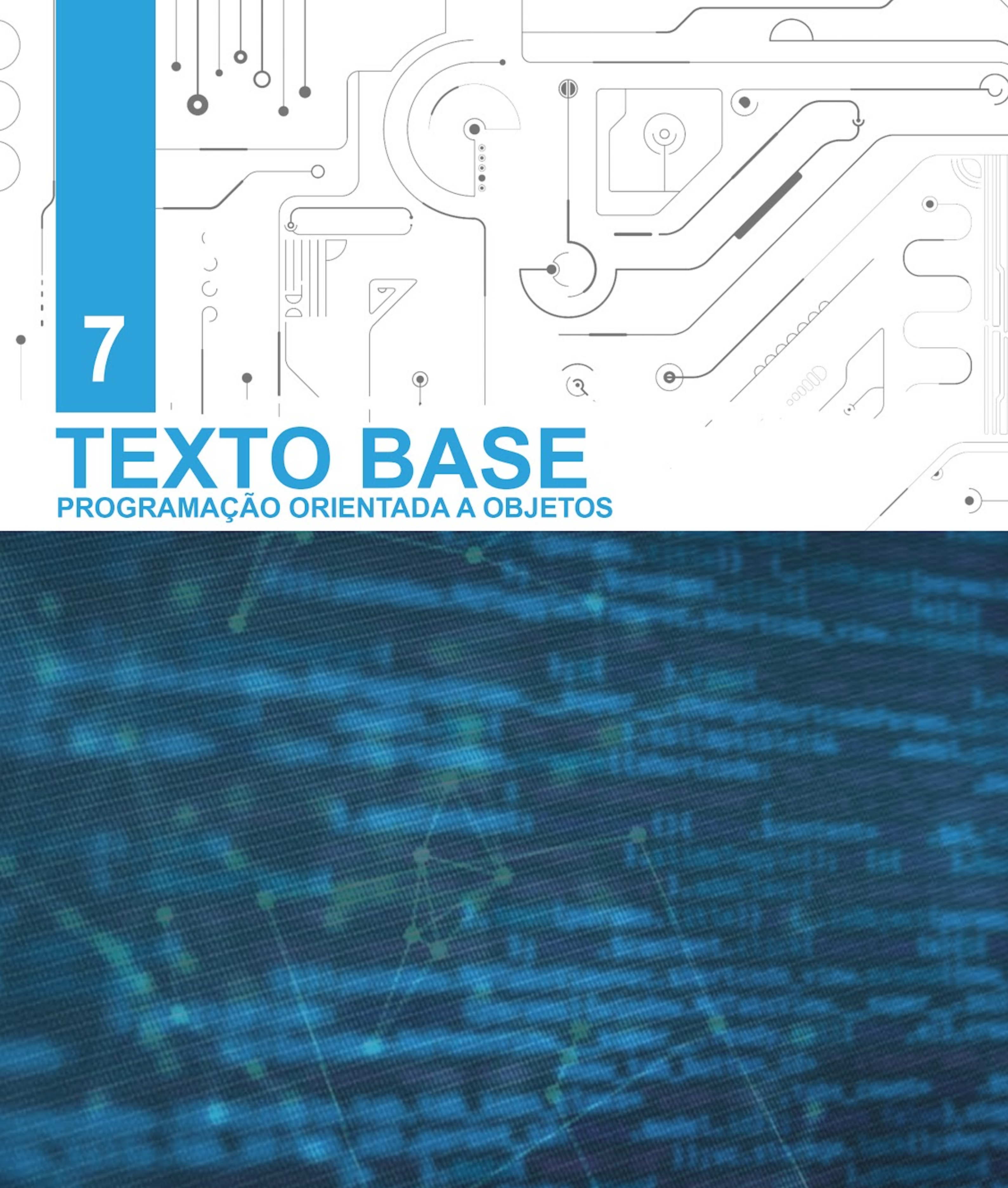


TEXTO BASE

PROGRAMAÇÃO ORIENTADA A OBJETOS

Faculdade IMPACTA

Texto base

7

Módulos e pacotes

Prof. MSc. Rafael Maximo Carreira Ribeiro

Resumo

Neste capítulo iremos aprender sobre a criação e utilização de módulos e pacotes em Python. Veremos a diferença entre os termos script, módulo e pacote, aprenderemos o que é a variável __name__ em um módulo Python e qual a sua finalidade, e estudaremos as diferentes formas de se importar um módulo completa ou parcialmente e como organizar nossos arquivos em projetos maiores.

7.1. Introdução

Já vimos que em Python, todo arquivo "**.py*" é automaticamente um módulo, que pode ser importado em outros arquivos "**.py*" ou diretamente em uma *Shell* do Python, mas pouco foi falado sobre como podemos usar isso para melhor organizar nosso código.

Quando iniciamos o aprendizado de uma linguagem de programação, é comum utilizarmos um interpretador interativo para testar os recursos da linguagem em tempo real e aprender sobre os diferentes tipos de dados, mas isso não nos permite salvar nosso progresso, pois a cada vez que fechamos o programa, tudo é perdido e precisamos começar do zero na próxima vez.

Para resolver isso, usamos um editor de texto para salvar as instruções em pequenos trechos de código, e então executamos o código a partir deste arquivo em um interpretador. A esses arquivos damos o nome de *scripts*.

Conforme avançamos nos estudos e no desenvolvimento de um projeto, esse arquivo cresce e começa a ficar responsável por muitas tarefas, o que dificulta a manutenção do código. Então é comum dividirmos esse *script* em diversos arquivos,

para facilitar a organização e manutenção do nosso projeto. A esses arquivos damos o nome de *módulos*.

Dependendo do tamanho do projeto, pode ser necessário um número muito grande de arquivos, então é comum dividirmos estes arquivos em pastas para melhor organizar o que cada conjunto de arquivos é responsável por fazer. A estas pastas damos os nomes de pacotes[1]. Um pacote pode conter quantos módulos e sub-pacotes forem necessários.

Observe que do ponto de vista prático, não existe uma diferença real entre um *script* e um módulo, a não ser pela forma como se pretende utilizá-los. Quando dividimos nosso código em módulos, podemos importar o conteúdo de um módulo para outro, o que facilita a reutilização de código, sem necessidade de copiá-lo.

Se construirmos os módulos de maneira que eles sejam o mais independentes possível, isto é, dependentes apenas de interfaces bem definidas para as funções e classes, mas agnósticos à sua implementação, nosso projeto será mais fácil de testar e manter, já que alterações em um módulo causam pouco (idealmente nenhum) impacto em outros módulos.

7.2. Módulos em Python

Podemos separar os módulos do Python em 3 tipos, de acordo com a sua origem, mas vale ressaltar que do ponto de vista do Python só existe um tipo de módulo.

- módulos integrados;
- módulos de terceiros; e
- módulos próprios.

Os primeiros módulos que utilizamos são aqueles que já vêm integrados à linguagem, como por exemplo os módulos `time` e `math`. Em Python, os módulos integrados à implementação padrão[2] da linguagem podem ser escritos tanto em C quanto em Python.

O módulo de matemática é um exemplo de módulo escrito em C, cujo código fonte pode ser visto no github oficial do Python (PSF, 2021a). Já o módulo `turtle`, cuja demo pode ser acessada a partir do menu de ajuda do IDLE, é escrito inteiramente em Python e seu código fonte pode ser visto no diretório de instalação do Python. Na *Shell* do Python, digite o código da Codificação 7.1 em uma *Shell* do Python e navegue até a pasta que for exibida para visualizar o código fonte do módulo turtle.

[1] O termo pacote normalmente se refere à forma como o código é organizado para ser distribuído, mas em um contexto mais genérico, é comum usarmos os termos biblioteca, pacote ou módulo, de maneira intercambiável, para indicar um conjunto de ferramentas que lida com um tipo de problema ou domínio.
[2] CPython é a implementação padrão do Python, escrita em C.

```
>>> import turtle
>>> print(turtle.__file__)
C:\Program Files\Python39\lib\turtle.py
```

Codificação 7.1: Visualização do diretório do módulo *turtle* no Python

Além dos módulos integrados, podemos também instalar módulos ou pacotes desenvolvidos por terceiros para resolver um determinado problema, como por exemplo, criar interfaces gráficas, ler e escrever dados em planilhas, criar aplicações web e API's, fazer análises estatísticas, manipular matrizes n-dimensionais, criar jogos, entre muitas outras possibilidades.

A principal forma de se obter tais módulos é através do PyPI[3], repositório oficial da linguagem Python, e a ferramenta mais popular e recomendada de se instalar tais pacotes é o `pip`[4]. O guia completo de utilização do pip pode ser visto na página do grupo responsável por manter o PyPI (PyPA[5], 2021).

VOCÊ SABIA?

Em Python, uma das principais ferramentas para trabalhar com vetores e matrizes é a biblioteca Numpy, desenvolvida para realizar com maior eficiência cálculos complexos com diferentes tipos de sequências. É uma biblioteca utilizada em áreas como ciência de dados, engenharia, matemática aplicada, estatística, economia, entre outras.

Essa biblioteca, como outras do CPython, foi desenvolvida em C, mas sua interface é Python, possibilitando a eficiência dos códigos em C, com a simplicidade da sintaxe Python. Portanto, escreve-se um código mais fácil de ler e entender, sem abrir mão do desempenho necessário para aplicações que processam enorme quantidade de dados.

O Numpy, juntamente com outras bibliotecas do Python que dependem dele, como Scikit-image, SciPy, MatPlotLib e Pandas, foi usado em duas situações com repercussão mundial: a geração da primeira imagem de um buraco negro (Numpy, 2020a) e a detecção de ondas gravitacionais (Numpy, 2020b) pelos cientistas do Observatório de Ondas Gravitacionais por Interferômetro Laser (LIGO).

Em ambos os casos o Python foi usado para coletar, tratar, analisar e gerar visualizações processando terabytes de dados diariamente. São aplicações do Python em problemas complexos, com muitos dados e com exigência de excelente desempenho computacional.

[3] Python Package Index
[4] Package Installer for Python
[5] Python Packaging Authority

E por fim, podemos ainda utilizar nossos próprios módulos em nossos projetos, ou seja, podemos subdividir nosso projeto em diversos arquivos e importar as classes e funções de um lugar para outro conforme necessário. então veremos agora como organizar nossos arquivos em módulos e como o Python trabalha com diferentes arquivos em um mesmo projeto.

7.2.1. Espaço de nomes de um módulo

O primeiro ponto a ser observado é o que podemos chamar de escopo global do módulo. Quando aprendemos a definir funções, estudamos a diferença entre o escopo global e o escopo local da função. Cada módulo em Python define um escopo global, no qual há uma tabela que relaciona os identificadores existentes naquele módulo com os objetos na memória.

Então, ao importarmos um módulo, o Python adiciona uma entrada neste escopo com uma referência para o módulo que foi importado. Isso pode ser verificado com o uso da função `dir`, que retorna uma lista dos nomes existentes em um dado escopo. Se chamada sem argumento, retorna os nomes do escopo atual. Veja o exemplo da Codificação 7.2.

Codificação 7.2: Utilização da função *dir*

```
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__',
'__name__', '__package__', '__spec__']
>>> import math
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__',
'__name__', '__package__', '__spec__', 'math']
```

Fonte: do autor, 2021

Podemos observar que o nome math foi adicionado ao escopo atual, e podemos então utilizar as funções que este módulo traz. Para ver a lista completa: `dir(math)`.

7.2.2. A variável `__name__`

Todo arquivo Python contendo definições e declarações é um módulo, e o nome do módulo é o próprio nome do arquivo sem a extensão. No interior do módulo, esse nome fica disponível, como *string*, na variável global `__name__`. Vamos criar um módulo de exemplo para praticar.

1) Crie um arquivo Python vazio com o nome `meu_modulo.py`;
2) Abra um terminal no próprio VSCode ou externamente (cmd ou powershell, se estiver no Windows);

3) Se necessário, navegue até a pasta do arquivo que você criou, com o comando: `cd caminho\para\pasta`;
4) Abra o interpretador do Python, com o comando: `py` no Windows, e `python` ou `python3` no Linux/Mac.

Na Codificação 7.3 importamos nosso módulo e inspecionamos o conteúdo da variável `__name__`.

Codificação 7.3: Inspeção do nome de um módulo importado na *Shell*

```
>>> import meu_modulo
>>> meu_modulo.__name__
'meu_modulo'
```

Fonte: do autor, 2021

Quando importamos um módulo, o Python cria um nome no escopo global para o qual o módulo foi importado e associa este nome ao objeto do módulo na memória (veja o que retorna a função `dir()` após a importação de `meu_modulo`). Dessa forma, evitamos qualquer conflito de nomes entre as variáveis globais do módulo atual, que está sendo executado, e dos módulos que são importados. Isso garante que podemos desenvolver nossos módulos sem nos preocupar com a unicidade dos nomes das variáveis, classes e funções, em relação a outros módulos.

Quando executamos um módulo diretamente, seja pressionando F5 na IDLE, usando o modo de depuração do VSCode, executando o interpretador do Python na linha de comando ou usando qualquer outra IDE, dizemos que ele é o módulo (ou *script*) principal. Nesse caso, a variável especial `__name__` recebe o como nome a string `'__main__'` .

Ao importarmos um módulo, o interpretador do Python executa todo o seu conteúdo uma única vez, criando um objeto na memória que contém internamente todos os objetos criados pelo módulo. Vamos então incluir a seguinte linha no nosso arquivo `meu_modulo.py`: `print(f'O nome deste módulo é: {__name__!r}')`.[6]

Agora podemos executar o arquivo na linha de comando:

Codificação 7.4: Execução do arquivo meu_modulo.py na linha de comando

```
E:\POO\aula07> py meu_modulo.py
O nome deste módulo é: '__main__'
```

Fonte: do autor, 2021

E podemos também importar esse arquivo para uma Shell do Python:

[6] O modificador `!r` força a utilização da função `repr`, que gera uma representação do objeto em *string*, como `__name__` já é uma *string*, a sua representação na tela inclui as aspas, evidenciando que é uma *string*, já que a função `print`, por padrão, exibe sempre o conteúdo da *string*, sem as aspas.

Codificação 7.5: Importação do módulo meu_modulo na *Shell* do Python

```
E:\POO\aula07> py
Python 3.9.1
>>> import meu_modulo
O nome deste módulo é: 'meu_modulo'
```

Fonte: do autor, 2021

Observe que quando o módulo é executado diretamente, o nome da variável especial `__name__` é `'__main__'`, já quando o módulo é importado, ou seja, não é o módulo principal sendo executado, essa variável guarda o nome do próprio módulo `meu_modulo`. No segundo exemplo, o “módulo” principal é a execução atual da *Shell* do Python, que define um espaço de nomes e também possui a variável especial `__name__`. Podemos confirmar isso executando os comando da Codificação 7.6 na mesma *Shell*:

Codificação 7.6: Inspeção dos nomes do módulo e do escopo atual

```
>>> __name__
'__main__'
>>> meu_modulo.__name__
'meu_modulo'
```

Fonte: do autor, 2021

Você deve ter observado que a instrução para exibir o nome do módulo foi executada automaticamente no momento da importação. Em geral, esse é um comportamento que queremos evitar ao importar um módulo, pois na grande maioria das vezes, queremos apenas carregar suas definições (classes, funções, constantes, etc.) para podermos utilizá-las conforme necessário. Vamos trocar a exibição direta para a definição de uma função que quando chamada exibe o nome do módulo.

Edite o arquivo `meu_modulo.py` para corresponder à Codificação 7.7.

Codificação 7.7: Conteúdo do arquivo meu_modulo.py

```
def exibe_nome():
    print(f'O nome deste módulo é: {__name__!r}')

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

Fonte: do autor, 2021

Antes de seguir para o próximo teste, feito na Codificação 7.8, feche a *Shell* que utilizamos para o teste anterior, com o comando `exit()`, ou abra um novo terminal. Isso é necessário pois o Python importa os módulos apenas uma vez, ao tentarmos

importar um módulo que já está importado, o Python identifica que aquele módulo já existe e ignora o comando de importação[7].

Codificação 7.8: Comparação entre a execução no terminal e a importação na Shell do arquivo da Codificação 7.7

```
E:\POO\aula07> py meu_modulo.py
O nome deste módulo é: '__main__'
E:\POO\aula07> py
Python 3.9.1
>>> import meu_modulo
>>> meu_modulo.exibe_nome()
O nome deste módulo é: 'meu_modulo'
```

Fonte: do autor, 2021

Para que nosso módulo continue funcionando também como um *script*, o que pode ser útil durante o desenvolvimento, para realização de testes por exemplo, podemos adicionar uma verificação do conteúdo da variável especial `__name__`, se o módulo estiver sendo executado como módulo principal, fazemos a chamada à função que exibe o nome, mas quando importamos o módulo, esse código não é executado.

Observe que para chamar a função que exibe o nome do módulo, usamos a notação de ponto, indicando que queremos executar a função `exibe_nome` que pertence ao módulo `meu_modulo`.

7.2.3. Formas de importar um módulo

Além da forma padrão que vimos até agora: `import modulo`, existem algumas outras formas de se importar um módulo, ou parte dele, para o escopo atual, como veremos agora.

7.2.3.1. O caminho de busca de um módulo

Ao executarmos um comando de importação de um módulo, o Python irá seguir as seguintes regras sobre onde procurar pelo arquivo do módulo:

- Antes de realizar qualquer busca, o Python verifica se aquele módulo já foi importado naquela sessão, se sim, ele reutiliza a mesma referência para o módulo já existente em memória e interrompe o processo.

[7] Para forçar a reimportação de um módulo, é necessário importar a biblioteca `importlib` e usar o método `reload()`, passando para ele o módulo a ser recarregado.

```
>>> import meu_modulo                    # importação inicial
>>> import importlib
>>> importlib.reload(meu_modulo)         # módulo recarregado
```

- Se o módulo ainda não foi importado, o Python irá fazer uma busca a partir de uma lista de diretórios que pode ser vista na variável `sys.path`. Esta lista é composta por 3 partes:
    1) O primeiro lugar buscado é a partir do diretório em que se encontra o arquivo de entrada, que está executando a importação, ou do diretório atual se não for especificado nenhum arquivo (como em uma *Shell*, por exemplo);
    2) Em seguida, ele busca a lista de diretórios especificada em uma variável de ambiente com caminhos padrão do Python; e
    3) Por último, há uma lista de caminhos padrão que depende da instalação e do sistema operacional.

É possível visualizar e alterar em tempo de execução essa lista para incluir ou excluir diretórios de busca conforme a necessidade, através da variável `sys.path`, utilizada como no exemplo da Codificação 7.9.

Codificação 7.9: Visualização da variável sys.path em uma instalação do Python 3.8 no Ubuntu

```
>>> import sys
>>> sys.path
['',
 '/usr/lib/python38.zip',
 '/usr/lib/python3.8',
 '/usr/lib/python3.8/lib-dynload',
 '/usr/local/lib/python3.8/dist-packages',
 '/usr/lib/python3/dist-packages']
```

Fonte: do autor, 2021

Essa variável é uma lista comum do Python, e portanto podemos usar todos os métodos de lista para alterá-la. As *strings* com cada caminho devem seguir o padrão do sistema operacional em questão, e tais alterações devem ser feitas com cautela, pois uma alteração incorreta poderá fazer com que a importação de módulos pare de funcionar ou tenha efeitos inesperados.

7.2.3.2. Importação com a criação de um alias para o módulo

Para criarmos um alias, ou apelido, para um módulo, usamos a sintaxe a seguir:

Codificação 7.10: Sintaxe para importação de um módulo atribuído a um novo nome (apelido)

```
import <nome do módulo> as <apelido>
```

Fonte: do autor, 2021

Isso irá associar o módulo importado ao nome definido por <apelido>. Traduzindo para o português, podemos ler a instrução acima como: “importe o módulo

<nome do módulo> como <apelido>". Veja o exemplo com módulo `math` na Codificação 7.11.

Codificação 7.11: Exemplo de utilização da importação com apelido

```
>>> import math as mat
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__',
'__name__', '__package__', '__spec__', 'mat']
>>> mat
<module 'math' (built-in)>
>>> mat.pi
3.141592653589793
```

Fonte: do autor, 2021

Observe que no escopo atual, foi criado o nome `mat`, e devemos utilizá-lo para acessar o objeto do módulo de matemática que contém todas as funções que este módulo nos traz. O nome do módulo (que pode ser acessado em `mat.__name__`) continua o mesmo, mas sua referência está associada a um novo nome no escopo atual.

7.2.3.3. Importação de elementos do módulo

É possível importar apenas um sub-pacote, classe ou função específica a partir de um módulo, com a seguinte sintaxe:

Codificação 7.12: Sintaxe da importação parcial de um elemento do módulo

```
from <nome do módulo> import <nome do elemento>
```

Fonte: do autor, 2021

Isso irá importar apenas o elemento para o escopo atual, e não o módulo inteiro. Em geral utilizamos essa abordagem quando não queremos carregar o módulo todo pois só utilizaremos poucas de suas funções ou quando queremos simplificar o código do escopo atual, pois agora não precisamos mais da notação de ponto para chegar até um determinado elemento do módulo. Traduzindo para o português, podemos ler a instrução acima como: "a partir do módulo <nome do módulo> importe o <elemento>". Veja novamente o exemplo com módulo de matemática na Codificação 7.13.

Codificação 7.13: Exemplo de importação parcial de um elemento do módulo

```
>>> from math import pi
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__',
'__name__', '__package__', '__spec__', 'pi']
>>> pi
3.141592653589793
```

Fonte: do autor, 2021

Note que podemos utilizar o nome `pi` diretamente, sem o nome do módulo, mas ao mesmo tempo, não temos acesso a nenhuma outro nome definido pelo módulo de matemática, como por exemplo a função `log`, `sqrt`, etc. Devemos também estar atentos a um eventual conflito que pode ocorrer caso nosso código atual também defina uma variável chamada `pi`.

É possível importar mais de um elemento de um mesmo módulo separando-os por vírgula, como no exemplo da Codificação 7.14, no qual importamos as funções de seno, cosseno e tangente, além da constante **π**.

Codificação 7.14: Exemplo de importação parcial de múltiplos elementos de um módulo

```
>>> from math import cos, pi, sin, tan
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__',
'__name__', '__package__', '__spec__', 'cos', 'pi', 'sin', 'tan']
```

Fonte: do autor, 2021

7.2.3.4. Importação de elemento com a criação de alias

É possível também unir as duas situações que acabamos de ver, e criarmos um apelido no escopo atual para o elemento que foi importado, com a seguinte sintaxe:

Codificação 7.15: Sintaxe da importação parcial de um elemento do módulo atribuído a um novo nome (apelido)

```
from <nome do módulo> import <nome do elemento> as <apelido>
```

Fonte: do autor, 2021

Podemos ler esse comando como “a partir do módulo <nome do módulo> importe o elemento <nome do elemento> como <apelido>”. Voltando ao exemplo do módulo de matemática, veja a Codificação 7.16.

Codificação 7.16: Exemplo de utilização conjunta da importação parcial com apelido

```
>>> from math import sin as seno
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__',
'__name__', '__package__', '__spec__', 'seno']
```

Fonte: do autor, 2021

E podemos ainda fazer a importação de mais de um elemento usando a atribuição de apelidos, como mostra a Codificação 7.17.

Codificação 7.17: Exemplo de utilização conjunta da importação parcial de múltiplos elementos do módulo com apelidos

```
>>> from math import cos as cosseno, pi, sin as seno, tan as tangente
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__',
'__name__', '__package__', '__spec__', 'cosseno', 'pi', 'seno',
'tangente']
```

Fonte: do autor, 2021

7.2.4. Organização de módulos em pacotes e sub-pacotes

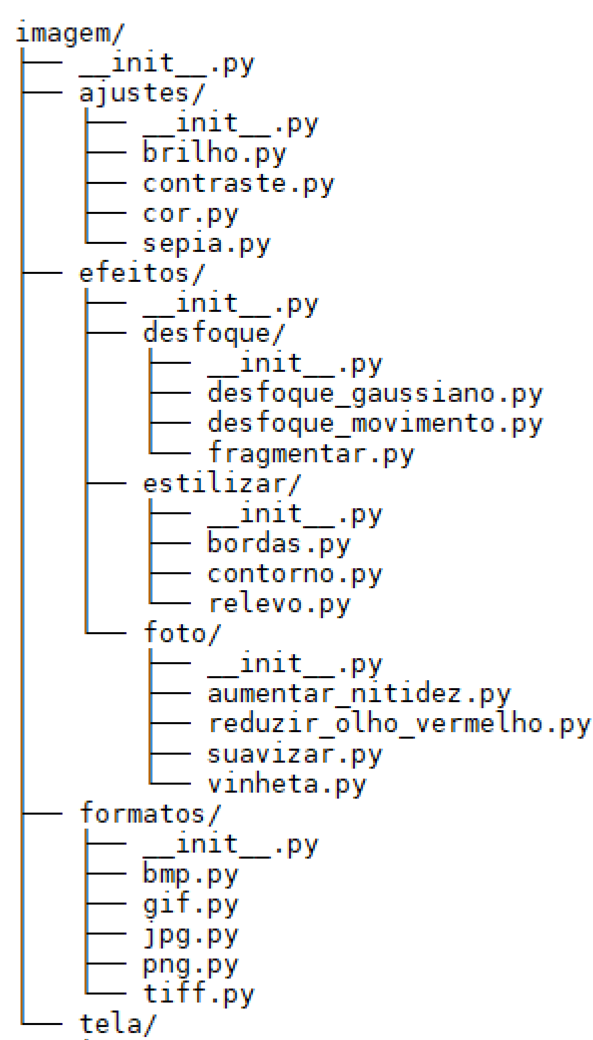
Para entender como podemos criar e organizar módulos em pacotes e sub-pacotes, veremos um exemplo de uma ferramenta para edição de imagens, cuja estrutura de diretórios é mostrada na Figura 7.1. Vale lembrar que o objetivo desta disciplina não é discutir a arquitetura do projeto, apenas mostrar um exemplo simplificado dessa organização, para entendermos como funciona a importação e utilização de tais módulos.

Neste exemplo, não estamos preocupados com o conteúdo de cada arquivo Python, mas imagine que em uma situação real eles teriam as classes e funções necessárias para realizar o processamento dos dados das imagens e a leitura e escrita em disco dos arquivos.

Com essa estrutura, podemos desenvolver cada módulo isoladamente, sem nos preocupar com os demais módulos nem com possíveis conflitos de nomes entre os módulos, pois como vimos, cada módulo possui seu próprio escopo global, no qual são definidos os nomes de variáveis, classes e funções.

Para acessar as funções e classes de um módulo, podemos utilizar a notação de ponto para chegar até o módulo que for necessário e importá-lo.

Figura 7.1: Exemplo de estrutura de um módulo para tratamento de imagens em Python.



```
imagem/
├── __init__.py
├── ajustes/
│   ├── __init__.py
│   ├── brilho.py
│   ├── contraste.py
│   ├── cor.py
│   └── sepia.py
├── efeitos/
│   ├── __init__.py
│   ├── desfoque/
│   │   ├── __init__.py
│   │   ├── desfoque_gaussiano.py
│   │   ├── desfoque_movimento.py
│   │   └── fragmentar.py
│   ├── estilizar/
│   │   ├── __init__.py
│   │   ├── bordas.py
│   │   ├── contorno.py
│   │   └── relevo.py
│   └── foto/
│       ├── __init__.py
│       ├── aumentar_nitidez.py
│       ├── reduzir_olho_vermelho.py
│       ├── suavizar.py
│       └── vinheta.py
├── formatos/
│   ├── __init__.py
│   ├── bmp.py
│   ├── gif.py
│   ├── jpg.py
│   ├── png.py
│   └── tiff.py
└── tela/
    ├── __init__.py
    ├── girar.py
    ├── inverter.py
    └── redimensionar.py
```

Fonte: do autor, 2021

7.2.4.1. O arquivo __init__.py

Você deve ter observado a presença de um arquivo com o nome `__init__.py` em cada uma das pastas. Desde a versão 3.3 do Python, esse arquivo é opcional e, se estiver presente, será executado quando o módulo for importado. Podemos deixar o arquivo vazio, apenas para indicar ao Python que aquela pasta deve ser interpretada como um pacote, ou colocar algum código de inicialização do nosso módulo, como por exemplo fazer a importação de outros nomes para o escopo atual.

Outra vantagem de colocar esse arquivo `__init__.py` é evitar que pastas com nomes comuns ao Python, como **string** por exemplo, acabem escondendo outros pacotes mais adiante no caminho de busca de pacotes.

Aqui estamos usando a palavra módulo para nos referenciar também ao pacote. Acontece que em python, todo pacote é também um módulo, que pode ser importado e define um espaço de nomes, mas nem todo módulo é um pacote (PSF, 2021b). Por exemplo, a maioria dos módulos são arquivos que contém as funções e classes que desejamos reutilizar.

Se deixarmos os arquivos de inicialização vazios, podemos utilizar esse módulo externamente sem nenhum problema, importando cada um dos submódulos que precisarmos.

No da Codificação 7.18, podemos importar cada módulo necessário e acessá-los utilizando a notação de ponto. Alternativamente, podemos fazer uma importação parcial apenas das funções que iremos precisar, como mostra na codificação 7.19.

Codificação 7.18: Exemplo 1 de utilização do módulo *imagem*

```
import imagem.ajustes.brilho
import imagem.efeitos.foto.vinheta
import imagem.formatos.jpg
import imagem.formatos.png
import imagem.tela.redimensionar

arquivo = input('Digite o nome do arquivo jpg: ')

dados = imagem.formatos.jpg.carregar(arquivo)
dados = imagem.tela.redimensionar.pixels(dados, 600, 400)
dados = imagem.ajustes.brilho.ajustar_brilho(dados, 0.7)
dados = imagem.efeitos.foto.vinheta.aplicar(dados, 0.8, '0032af')

nome = input('Salvar como png: ')
imagem.formatos.png.salvar(dados, f'{nome}.png')
```

Fonte: do autor, 2021

Codificação 7.19: Exemplo 2 de utilização do módulo *imagem*

```
from imagem.ajustes.brilho import ajustar_brilho
from imagem.efeitos.foto.vinheta import aplicar
from imagem.formatos.jpg import carregar
from imagem.formatos.png import salvar
from imagem.tela.redimensionar import pixels

arquivo = input('Digite o nome do arquivo jpg: ')

dados = carregar(arquivo)
dados = pixels(dados, 600, 400)
dados = ajustar_brilho(dados, 0.7)
dados = aplicar(dados, 0.8, '0032af')

nome = input('Salvar como png: ')
salvar(dados, f'{nome}.png')
```

Fonte: do autor, 2021

A forma da codificação 7.19 deixa o código mais simples, mas ao mesmo tempo perdemos parte da semântica trazida ao lermos o caminho completo de importação de cada função utilizada. Portanto, a decisão entre qual delas usar irá depender do contexto e neste exemplo, é razoável pensar que haverá mais de uma função *carregar* ou *salvar*, então poderia fazer mais sentido a importação apresentada na Codificação 7.20.

Codificação 7.20: Exemplo 3 de utilização do módulo *imagem*

```
from imagem.ajustes.brilho import ajustar_brilho
from imagem.efeitos.foto import vinheta
from imagem.formatos import jpg, png
from imagem.tela.redimensionar import pixels

arquivo = input('Digite o nome do arquivo jpg: ')

dados = jpg.carregar(arquivo)
dados = pixels(dados, 600, 400)
dados = ajustar_brilho(dados, 0.7)
dados = vinheta.aplicar(dados, 0.8, '0032af')

nome = input('Salvar como png: ')
png.salvar(dados, f'{nome}.png')
```

Codificação 7.20: Exemplo 3 de utilização do módulo *imagem*

Neste terceiro exemplo, continuamos com um código mais simples e evitamos um possível conflito de nomes entre a função salvar do formato jpg e png, por exemplo, além de deixar o código mais expressivo e mais fácil de ler.

7.2.4.2. Inicialização dos módulos

Ao deixarmos os arquivos de inicialização vazios, os submódulos não são automaticamente importados ao importarmos simplesmentes o módulo principal, isto é, ao executar o comando `import imagem`, mas ao realizar este comando, o arquivo `imagem/__init__.py` será executado, então podemos incluir nele o código da Codificação 7.21.

Codificação 7.21: Código do arquivo de inicialização do módulo imagem (imagem/__init__.py)

```
import imagem.ajustes
import imagem.efeitos
import imagem.formatos
import imagem.tela
```

Fonte: do autor, 2021

Tais importações irão executar os arquivos de inicialização de cada um destes submódulos, então se repetirmos esse procedimento para todos os arquivos de inicialização, importando em cada um deles os submódulos ali presentes, teremos todo o módulo de imagem carregado para o escopo atual ao fazermos simplesmente o comando: `import imagem`. Dessa forma, podemos simplificar a importação do módulo como mostra a Codificação 7.22.

Codificação 7.20: Exemplo 4 de utilização do módulo *imagem*

```
import imagem

arquivo = input('Digite o nome do arquivo jpg: ')

dados = imagem.formatos.jpg.carregar(arquivo)
dados = imagem.tela.redimensionar.pixels(dados, 600, 400)
dados = imagem.ajustes.brilho.ajustar_brilho(dados, 0.7)
dados = imagem.efeitos.foto.vinheta.aplicar(dados, 0.8, '0032af')

nome = input('Salvar como png: ')

imagem.formatos.png.salvar(dados, f'{nome}.png')
```

Fonte: do autor, 2021

7.3. Nota sobre a criação de pacotes em Python

A criação de pacotes em Python é um assunto muito mais complexo do que o que foi apresentado neste capítulo, e o seu entendimento por completo se faz necessário apenas ao precisarmos criar um pacote Python para distribuição.

Para os exemplos que veremos ao longo do curso, é suficiente deixar os arquivos na mesma pasta, sem a necessidade de criação do arquivo de inicialização. E podemos utilizar tanto a importação completa (`import <módulo>`) quanto a parcial (`from <módulo> import <elemento>`[8]), conforme desejarmos, que isso irá funcionar.

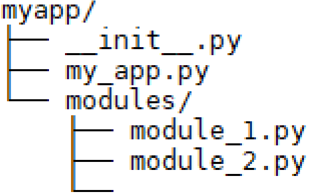
Para continuar seus estudos a respeito dos módulos em Python, veja o tutorial da documentação oficial (PSF, 2021c).

7.4. PEP 8

A recomendação da PEP 8 em relação à nomenclatura de pacotes e módulos, cujo exemplo pode ser visto na Figura 7.2, é a seguinte:

- Módulos (arquivos) devem ter nomes curtos e com todas as letras minúsculas, e as palavras podem ser separadas por sublinhado se isso melhorar a legibilidade.
- Pacotes (pastas) devem seguir a mesma regra, nomes curtos com letras minúsculas, mas sem a separação com sublinhados.

Figura 7.2: Exemplo de nomenclatura de módulos e pacotes segundo a PEP 8.



Fonte: Elaborado pelo autor

[8] <elemento> é qualquer nome definido no escopo do módulo, pode ser um submódulo, classe, função ou variável global.

Bibliografia

Case study: first image of a black hole. Numpy, 2020a. Disponível em: <https://numpy.org/case-studies/blackhole-image/>. Acesso em: 20 mar. 2021.

Case study: discovery of gravitational waves. Numpy, 2020b. Disponível em: <https://numpy.org/case-studies/gw-discov/>. Acesso em: 20 mar. 2021.

PSF. **Repositório oficial do python:** código fonte do módulo *math*. 2021a. Disponível em: <https://github.com/python/cpython/blob/master/Modules/mathmodule.c>. Acesso em: 31 mar. 2021.

PSF. **The import system:** packages. 2021b. Disponível em: <https://docs.python.org/3/reference/import.html#packages>. Acesso em: 3 abr. 2021.

PSF. **O tutorial python:** Módulos. 2021c. Disponível em: <https://docs.python.org/pt-br/3/tutorial/modules.html>. Acesso em: 3 abr. 2021.

PyPA. **PIP documentation:** user guide. 2021. Disponível em: <https://pip.pypa.io/en/stable/user_guide/>. Acesso em: 31 mar. 2021.