

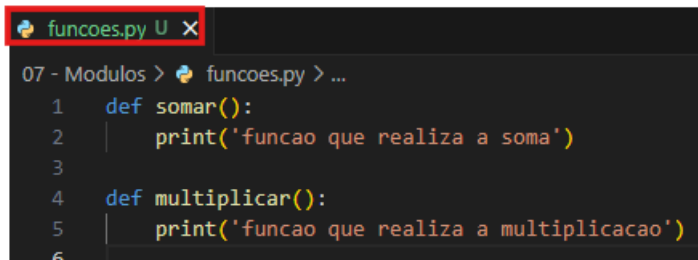
Modulos

Vamos agora organizar melhor o nosso código!! Pois até agora criamos as nossas funções diretamente pelo script principal do **python**, mas agora vamos organizar utilizando o recurso de **Módulos**!!

E como já foi citado anteriormente no capítulo <https://github.com/fspelling/python-study/tree/main/03%20-%20Funcoes/01%20-%20De%20Function%20a%20Library>, **módulos** se trata de um arquivo ou vários arquivos, onde queremos **agrupar** as nossas **funções**, para que seja reutilizada em nossa aplicação, ou ate mesmo em outras aplicações caso seja exposto em uma **library**.

Criando o primeiro modulo

E para exemplificar, vamos criar o nosso primeiro arquivo de **modulo**, que nesse caso terá algumas funções de operações matemáticas.



```
07 - Modulos > funcoes.py > ...
1  def somar():
2      print('funcao que realiza a soma')
3
4  def multiplicar():
5      print('funcao que realiza a multiplicacao')
6
```

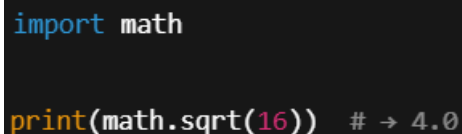
Pronto, já temos as nossas funções agrupadas em um modulo, que nesse caso é referente ao arquivo **funções.py**.

Importando o modulo com FROM e IMPORT

Após a criação do modulo, precisamos importá-lo em nosso código para utilizá-lo!! E para isso temos 2 formas de realizar, que são:

- **import**: importa o modulo inteiro, ou seja, importa **TODAS** as funções e objetos do modulo na sua aplicação!!

- Exemplo:



```
import math

print(math.sqrt(16))  # -> 4.0
```

Veja que ao importar pelo **import**, acessamos as funções pela referência da modulo importado!!

- **from [...] import:** importa funções e objetos específicos do módulo!!

- Exemplo:

```
from math import sqrt

print(sqrt(16))  # → 4.0
```

Veja que ao importar pelo **from ... import**, acessamos as funções diretamente pela própria função do módulo importado, sem precisar referência o nome do módulo sem si!!

Dito isso, qual devo usar então??

- Use **import** modulo quando quiser **evitar conflitos** e manter o código mais explícito na importação.
- Use **from ... import** quando quiser deixar o código mais **enxuto** ou só precisa de **itens específicos** do módulo.

E para exemplificar a importação das 2 maneiras, veja como que ficaria!

→ Usando o **import**

```
07 - Modulos > main.py
1  import funcoes
2  funcoes.somar()
3
```

→ Usando o **form ... import**

```
07 - Modulos > main.py
1  from funcoes import somar
2  somar()
3
```

Obs: Com o uso do **from ... import**, até podemos importar todas as funções daquele módulo, sem precisar especificar as funções específicas, porém isso **NÃO DEVE SER FEITO!!** Pois se tiver funções de módulos diferentes com o mesmo nome, um modulo pode impactar e sobrescrever a função do outro modulo!!

```
from math import *
from cmath import *  # também tem sqrt()

print(sqrt(-1))  # Vai usar cmath.sqrt e retornar (0+1j)
```

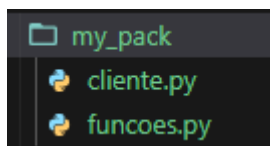
Criando e importando Packages

Podemos melhorar ainda mais a nossa organização de código, pois já vimos como colocar várias das nossas funções em **módulos** diferentes, mas se não tomar cuidado esses **módulos** podem crescer com tempo e ficar difícil de manter!

É aí que entra os **Packages**!!

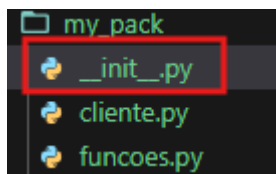
Pois com os **packages** conseguimos organizar nossos módulos em diretórios diferentes, facilitando na importação dos módulos, indicando de qual **package** que aqueles módulos pertencem!

Dito isso, vamos criar um diretório de nome **“my_pack”**, onde ficara os nossos módulos criados.



Vale destacar que nas versões mais recentes do Python, só de criar um diretório para os nossos **módulos** o **Python** já vai saber que se trata de um **package** e tudo funcionara ao realizar a importação dos **módulos** por esse **package**!!

Porém a recomendação é de criar um arquivo chamado **__init__.py** no diretório do **package**, para que o Python saiba que esse diretório se trata de um **package** e evitar qualquer tipo de problema!



Com o **package** criado, chegou a hora de importar os módulos a partir do **package**, e para isso basta definir o nome do **package** na importação pelo script principal, que nesse caso é o **main.py**.

```
07 - Modulos > main.py
1  from my_pack.funcoes import somar
2  from my_pack.cliente import get_cliente
```

Obs: Um detalhe importante que ao rodar o **script main.py**, foi gerado um diretório chamado **__pycache__** /, pois quando temos arquivos **.py** que nesse caso seria nossos **módulos**, que o script necessita, o **Python** irá **armazenar** uma compilação para **bytecode** (mais rápido de interpretar), e isso acaba melhorando a performance da aplicação!! Pois na próxima vez que for executado o script e esses módulos não forem alterados, o Python não precisara realizar a compilação desses arquivos novamente!!

Exportando uma Library

E se quisermos expor nossos **packages** para outros devs utilizarem o nosso código também?? Como fazemos?? E a resposta disso é realizando essa exportação através de **libraries**!!

E para exportar nossos **packages** através de uma **library** é bem simples!! Basta criarmos um arquivo **setup.py**, onde será definido todas as informações sobre a nossa biblioteca, como podemos ver a seguir:

```
07 - Modulos > setup.py
1  from setuptools import setup, find_packages
2
3  setup(
4      name="my_library",
5      version="1.0",
6      packages=find_packages(),
7      description="Biblioteca de exemplo",
8      author="Fernando Spelling",
9      author_email="fernando.spelling@gmail.com",
10 )
11
```

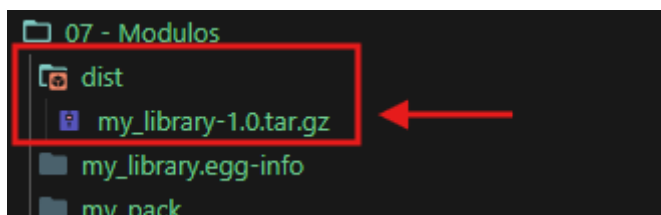
Veja que precisamos importar a **lib** do **Python** chamado **setuptools**, e é com ela que realizamos toda a configuração da nossa **library**!!

Após realizado a configuração, é só executar a **linha de comando** para que seja realizado o empacotamento do pacote referente a **library**:

python setup.py sdist

```
C:\Projetos\pocs\python-study\07 - Modulos (main -> origin)
λ python setup.py sdist
running sdist
running egg_info
creating my_library.egg-info
writing my_library.egg-info\PKG-INFO
writing dependency_links to my_library.egg-info\dependency_links.txt
writing top-level names to my_library.egg-info\top_level.txt
writing manifest file 'my_library.egg-info\SOURCES.txt'
reading manifest file 'my_library.egg-info\SOURCES.txt'
writing manifest file 'my_library.egg-info\SOURCES.txt'
warning: sdist: standard file not found: should have one of README, README.d
```

Feito isso, veja que foi gerado o arquivo desse empacotamento da nossa **library** dentro do diretório **dist/**.



Pronto!! Agora é só os outros Devs instalarem o nosso pacote com o comando:

pip install ./dist/minha_library-0.1.tar.gz

Fazendo isso, poderão utilizar as funções e objetos compartilhados na nossa **library** que foi empacotado!!!