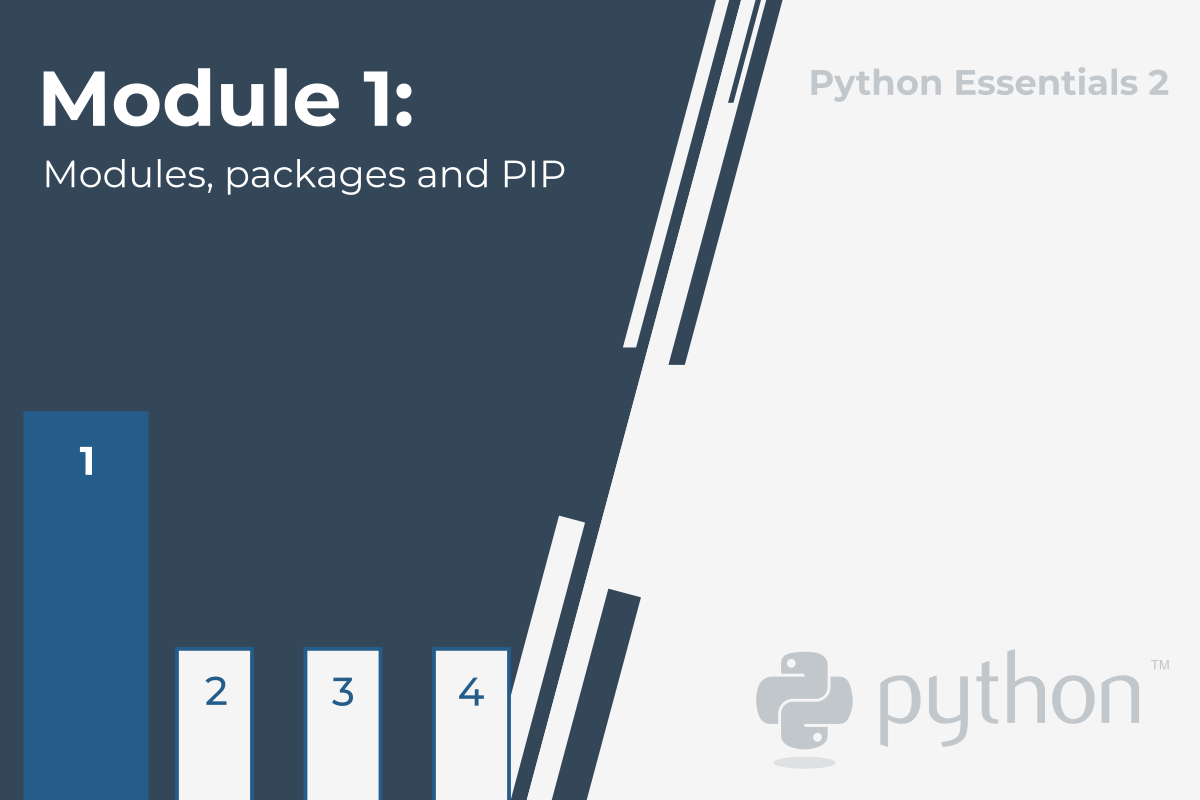
**Python Essentials 2:  
Módulo 1**

**Módulos, Pacotes e PIP**

Neste módulo, aprenderá sobre:

* importação e utilização de módulos Python;
* utilização de alguns dos módulos de biblioteca padrão Python mais úteis;
* construção e utilização de pacotes Python;
* PIP (Python Installation Package) e como utilizá-lo para instalar e desinstalar pacotes prontos a usar do PyPI.



**O que é um módulo?**

O código informático tem uma tendência para crescer. Podemos dizer que o código que não cresce é provavelmente completamente inutilizável ou abandonado. Um código real, procurado e amplamente utilizado desenvolve-se continuamente, à medida que tanto as exigências dos utilizadores como as expetativas dos utilizadores se desenvolvem nos seus próprios ritmos.

Um código que não seja capaz de responder às necessidades dos utilizadores será rapidamente esquecido, e imediatamente substituído por um código novo, melhor e mais flexível. Esteja preparado para isto, e nunca pense que algum dos seus programas acabe por ser concluído. A conclusão é um estado de transição e normalmente passa rapidamente, após o primeiro relatório de bug. O próprio Python é um bom exemplo de como a regra atua.

Código em crescimento é, de facto, um problema em crescimento. Um código maior significa sempre uma manutenção mais dura. A procura de bugs é sempre mais fácil onde o código é mais pequeno (tal como encontrar uma quebra mecânica é mais simples quando a maquinaria é mais simples e pequena).

Além disso, quando se espera que o código a ser criado seja realmente grande (pode usar um número total de source lines como uma medida útil, mas não muito precisa, do tamanho de um código) pode querer (ou melhor, será forçado a) dividi-lo em muitas partes, implementado em paralelo por alguns, uma dúzia, várias dúzias, ou mesmo várias centenas de programadores individuais.

É claro que isto não pode ser feito utilizando um grande source file, que é editado por todos os programadores ao mesmo tempo. Isto conduziria certamente a um desastre espetacular.

Se deseja que tal projeto de software seja concluído com êxito, tem de ter os meios que lhe permitam fazê-lo:

* dividir todas as tarefas entre os programadores;
* juntar todas as partes criadas num só todo funcional.

Por exemplo, um determinado projeto pode ser dividido em duas partes principais:

* a interface do utilizador (a parte que comunica com o utilizador utilizando widgets e um ecrã gráfico)
* a lógica (a parte que processa dados e produz resultados)

Cada uma destas partes pode ser (muito provavelmente) dividida em partes mais pequenas, e assim por diante. Tal processo é muitas vezes chamado **decomposição**.

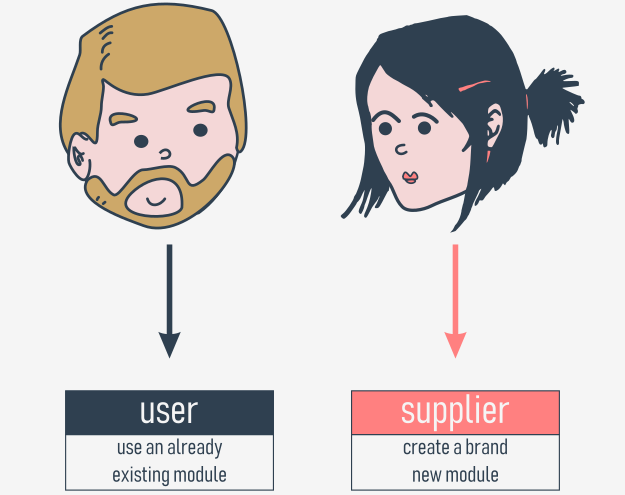
Por exemplo, se lhe pedissem para organizar um casamento, não faria tudo sozinho - encontraria um número de profissionais e dividiria a tarefa entre todos eles.

Como se divide um software em partes separadas mas cooperantes? Esta é a questão. Os **módulos** são a resposta.

**Como fazer uso de um módulo?**

Então, o que é um módulo? O [Tutorial Python](https://docs.python.org/3/tutorial/modules.html) define-o como **um ficheiro contendo definições e declarações Python**, que pode ser posteriormente importado e utilizado quando necessário.

A manipulação de módulos consiste em duas questões diferentes:



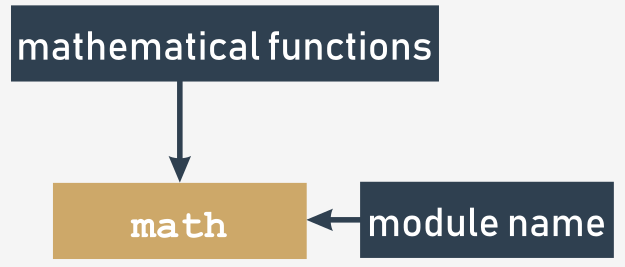
* o primeiro (provavelmente o mais comum) acontece quando se pretende utilizar um módulo já existente, escrito por outra pessoa, ou criado por si próprio durante o seu trabalho nalgum projeto complexo - neste caso você é o **utilizador** do módulo;
* o segundo ocorre quando se pretende criar um novo módulo, quer para uso próprio, quer para facilitar a vida a outros programadores - você é o **fornecedor** do módulo.

Vamos discuti-los separadamente.

Em primeiro lugar, um módulo é identificado pelo seu **nome**. Se quiser utilizar qualquer módulo, precisa de saber o nome. Um número (bastante grande) de módulos é entregue juntamente com o próprio Python. Pode pensar neles como uma espécie de “equipamento extra de Python”.

Todos estes módulos, juntamente com as funções integradas, formam a **Python standard library** - um tipo especial de biblioteca onde os módulos desempenham o papel de livros (podemos mesmo dizer que as pastas desempenham o papel de prateleiras). Se quiser ver a lista completa de todos os “volumes” recolhidos nessa biblioteca, pode encontrá-la aqui: <https://docs.python.org/3/library/index.html>.

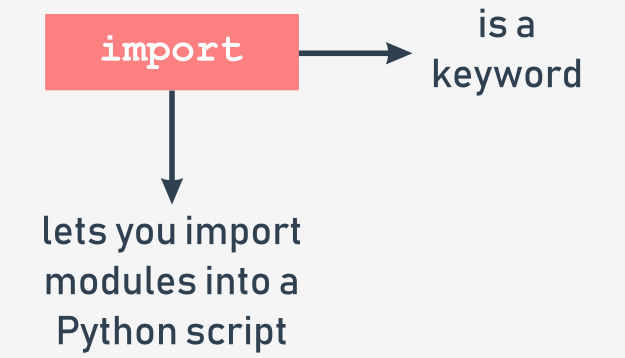
Cada módulo consiste em entidades (como um livro consiste em capítulos). Estas entidades podem ser funções, variáveis, constantes, classes e objetos. Se souber como aceder a um determinado módulo, pode fazer uso de qualquer uma das entidades que ele armazena.



Vamos começar a discussão com um dos módulos mais usados, chamado math. O seu nome fala por si - o módulo contém uma rica coleção de entidades (não apenas funções) que permitem a um programador implementar eficazmente cálculos que exijam o uso de funções matemáticas, como *sin()* ou *log()*.

**Importar um módulo**

Para tornar um módulo utilizável, deve **importá-lo** (pense nisto como tirar um livro da prateleira). Importar um módulo é feito por uma instrução chamada import. Nota: import é também uma keyword (com todas as consequências deste facto).



Vamos supor que deseja utilizar duas entidades fornecidas pelo módulo math :

* um símbolo (constante) representando um valor preciso (o mais preciso possível usando aritmética de duplo floating-point) de π (embora a utilização de uma letra grega para nomear uma variável seja totalmente possível em Python, o símbolo é nomeado **pi** - é uma solução mais conveniente, especialmente para aquela parte do mundo que não tem nem vai usar um teclado grego)
* uma função chamada sin() (o equivalente informático da função *seno* matemática)

Ambas as entidades estão disponíveis através do módulo math , mas a forma como pode utilizá-los depende fortemente de como a importação foi feita.

A maneira mais simples de importar um módulo específico é utilizar a instrução de importação da seguinte forma:

import math

A cláusula contém:

* a keyword import ;
* o **nome do módulo** que está sujeito a importação.

A instrução pode estar localizada em qualquer parte do seu código, mas deve ser colocada **antes da primeira utilização de qualquer uma das entidades do módulo**.

Se quiser (ou tiver de) importar mais do que um módulo, pode fazê-lo repetindo a cláusula import (de preferência):

import math

import sys

ou listando os módulos após a keyword import , como aqui:

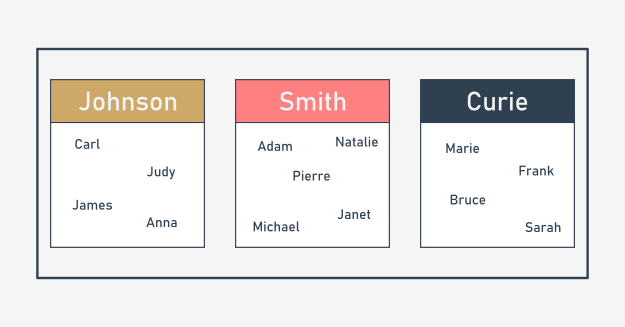
import math, sys

A instrução importa dois módulos, primeiro o chamado math e, em seguida, o segundo chamado sys.

A lista dos módulos pode ser arbitrariamente longa.

Para continuar, é preciso familiarizar-se com um termo importante: **namespace**. Não se preocupe, não entraremos em grandes detalhes - esta explicação será tão curta quanto possível.

Um **namespace** é um espaço (entendido num contexto não físico) no qual alguns nomes existem, e os nomes não entram em conflito entre si (ou seja, não existem dois objetos diferentes com o mesmo nome). Podemos dizer que cada grupo social é um namespace - o grupo tende a nomear cada um dos seus membros de uma forma única (por exemplo, os pais não dão aos seus filhos os mesmos nomes próprios).

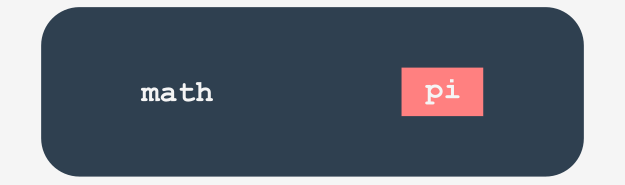


Esta singularidade pode ser conseguida de várias maneiras, por exemplo, utilizando alcunhas juntamente com os nomes próprios (funcionará dentro de um pequeno grupo como uma turma numa escola) ou atribuindo identificadores especiais a todos os membros do grupo (o Número de Segurança Social é um bom exemplo de tal prática).

**Dentro de um determinado namespace, cada nome deve permanecer exclusivo**. Isto pode significar que alguns nomes podem desaparecer, quando qualquer outra entidade de um nome já conhecido entra no namespace. Vamos mostrar-lhe como funciona e como controlá-lo, mas primeiro, vamos voltar às importações.

Se o módulo de um nome especificado **existir e for acessível** (um módulo é, na verdade, um **source file de Python**), o Python importa o seu conteúdo, ou seja, **todos os nomes definidos no módulo tornam-se conhecidos**, mas não entram no namespace do seu código.

Isto significa que pode ter as suas próprias entidades nomeadas sin ou pi e não serão de forma alguma afetadas pela importação.



A este ponto, poderá estar a perguntar-se como pode aceder à pi vinda do módulo math .

Para fazer isso, tem de qualificar o pi com o nome do seu módulo original.

Olhe para o snippet em baixo. Esta é a forma como se qualifica os nomes de pi e sin com o nome do seu módulo de origem:

math.pi

math.sin

É simples, coloca:

* o **nome do módulo** (por exemplo, math)
* um **ponto** (i.e., .)
* o **nome da entidade** (por exemplo, pi)

Tal forma indica claramente o namespace em que o nome existe.

Nota: a utilização desta qualificação é **obrigatória** se um módulo tiver sido importado pela instrução do módulo import . Não importa se algum dos nomes do seu código e do namespace do módulo está ou não em conflito.

Este primeiro exemplo não será muito avançado - queremos apenas imprimir o valor de **sen(½π)**.

Veja o código no editor. É assim que o testamos.

import math

print(math.sin(math.pi/2))

O código faz output do valor esperado: 1.0.

Nota: a remoção de qualquer uma das duas qualificações tornará o código errado. Não há outra maneira de introduzir mathdo namespace se fez o seguinte:

import math

Agora vamos mostrar-lhe como os dois namespaces (o seu e o do módulo) podem coexistir.

Dê uma vista de olhos no exemplo na janela do editor.

Nós definimos o nosso pi e sin aqui.

Execute o programa. O código deve produzir o seguinte output:

0.99999999

1.0

**output**

Como pode ver, as entidades não se afetam umas às outras.

No segundo método, a sintaxe de importindica precisamente qual a entidade (ou entidades) do módulo que é aceitável no código:

from math import pi

A instrução consiste nos seguintes elementos:

* a keyword from ;
* O **nome do módulo** a importar (seletivamente);
* a keyword import ;
* o **nome ou lista de nomes da entidade/entidades** que estão a ser importadas para o namespace.

A instrução tem este efeito:

* As entidades listadas (e apenas essas) são **importadas do módulo indicado**;
* Os nomes das entidades importadas são **acessíveis sem qualificação**.

Nota: nenhuma outra entidade é importada. Além disso, não se pode importar entidades adicionais utilizando uma qualificação - uma linha como esta:

print(math.e)

causará um erro (e é o número de Euler: 2.71828...)

Vamos reescrever o script anterior para incorporar a nova técnica.

Aqui está:

from math import sin, pi

print(sin(pi/2))

O output deve ser o mesmo que anteriormente, pois de facto já utilizámos as mesmas entidades de antes: 1.0. Copie o código, cole-o no editor e execute o programa.

O código parece mais simples? Talvez, mas o aspeto não é o único efeito deste tipo de importação. Vamos mostrar-lho.

Veja o código no editor. Analise-o com cuidado:

* linha 1: efetuar a importação seletiva;
* linha 3: fazer uso das entidades importadas e obter o resultado esperado (1.0)
* linhas 5 a 12: redefinir o significado de pi e sin - com efeito, substituem as definições originais (importadas) dentro do namespace do código;
* linha 15: obter 0.99999999, que confirma as nossas conclusões.

Vamos fazer outro teste. Veja o código em baixo:

pi = 3.14

def sin(x):

if 2 \* x == pi:

return 0.99999999

else:

return None

print(sin(pi / 2))

from math import sin, pi

print(sin(pi / 2))

Aqui, revertemos a sequência das operações do código:

* linhas 1 a 8: definir o nosso próprio pi e sin;
* linha 11: fazer uso deles (0.99999999 aparece no ecrã)
* linha 13: realizar a importação - os símbolos importados **substituem as suas definições anteriores dentro do namespace**;
* linha 15: obter 1.0 como resultado.

# Importar um módulo: \*

No terceiro método, a sintaxe de importé uma forma mais agressiva da apresentada anteriormente:

from module import \*

Como pode ver, o nome de uma entidade (ou a lista de nomes das entidades) é substituído por um único asterisco (\*).

Tal instrução **importa todas as entidades do módulo indicado**.

Isto é conveniente? Sim, é, pois liberta-o do dever de enumerar todos os nomes de que necessita.

É inseguro? Sim, é - a menos que saiba todos os nomes fornecidos pelo módulo, talvez **não consiga evitar conflito de nomes**. Trate isto como uma solução temporária, e tente não a utilizar em código normal.

## Importação de um módulo: a keyword as .

Se utilizar a variante do módulo de importação e não gostar do nome de um determinado módulo (por exemplo, é o mesmo que uma das suas entidades já definidas, pelo que a qualificação se torna problemática) pode dar-lhe o nome que quiser - a isto chama-se **aliasing**.

Aliasing faz com que o módulo seja identificado com um nome diferente do original. Isto também pode encurtar os nomes qualificados.

A criação de um alias é feita em conjunto com a importação do módulo, e exige a seguinte forma da instrução de importação:

import module as alias

O “módulo” identifica o nome do módulo original enquanto o “alias” é o nome que deseja utilizar em vez do original.

Nota: as é uma keyword.

Se precisar de mudar a palavra math, pode introduzir o seu próprio nome, tal como no exemplo:

import math as m

print(m.sin(m.pi/2))

Nota: após a execução bem sucedida de uma importação de um alias (pseudónimo), o **nome original do módulo torna-se inacessível** e não deve ser utilizado.

Por sua vez, quando utiliza a variante from module import name e precisa de mudar o nome da entidade, faz um alias para a entidade. Isto fará com que o nome seja substituído pelo alias que escolher.

É assim que pode ser feito:

from module import name as alias

Como anteriormente, o nome original (unaliased) torna-se inacessível.

A frase name as alias pode ser repetida - use vírgulas para separar as frases multiplicadas, como esta:

from module import n as a, m as b, o as c

O exemplo pode parecer um pouco estranho, mas funciona:

from math import pi as PI, sin as sine

print(sine(PI/2))

Agora está familiarizado com as noções básicas de utilização de módulos. Deixe-nos mostrar-lhe alguns módulos e algumas das suas entidades úteis.

# Key takeaways

1. Se quiser importar um módulo como um todo, pode fazê-lo usando a declaração import module\_name . É-lhe permitido importar mais do que um módulo de uma só vez utilizando uma lista separada por vírgulas. Por exemplo:

import mod1

import mod2, mod3, mod4

embora esta última forma não seja recomendada devido a razões estilísticas, e é melhor e mais bonito expressar a mesma intenção de uma forma mais verbosa e explícita, como por exemplo:

import mod2

import mod3

import mod4

2. Se um módulo for importado da forma acima referida e quiser aceder a qualquer uma das suas entidades, terá de prefixar o nome da entidade utilizando a **notação de ponto**. Por exemplo:

import my\_module

result = my\_module.my\_function(my\_module.my\_data)

O snippet faz uso de duas entidades provenientes do módulo my\_module : uma função chamada my\_function() e uma variável chamada my\_data. Ambos os nomes **devem ser prefixados por** my\_module. Nenhum dos nomes das entidades importadas entra em conflito com os nomes idênticos existentes no espaço de nomes do seu código.

3. É-lhe permitido não só importar um módulo como um todo, mas também importar apenas entidades individuais a partir dele. Neste caso, as entidades importadas **não devem ser** prefixadas quando utilizadas. Por exemplo:

from module import my\_function, my\_data

result = my\_function(my\_data)

A forma acima - apesar da sua atratividade - não é recomendada devido ao perigo de causar conflitos com nomes derivados da importação do namespace do código.

4. A forma mais geral da declaração acima permite importar **todas as entidades** oferecidas por um módulo:

from my\_module import \*

result = my\_function(my\_data)

**Nota**: a variante desta importação não é recomendada pelas mesmas razões que anteriormente (a ameaça de um conflito de nomes é ainda mais perigosa aqui).

5. Pode alterar o nome da entidade importada "on the fly", utilizando a frase as do import. Por exemplo:

from module import my\_function as fun, my\_data as dat

result = fun(dat)

**Exercício 1**

Pretende invocar a função make\_money() contida no módulo denominado mint. O seu código começa com a seguinte linha:

import mint

Qual é a forma adequada de invocação da função?  
  
**Exercício 2**

Pretende invocar a função make\_money() contida no módulo denominado mint. O seu código começa com a seguinte linha:

from mint import make\_money

Qual é a forma adequada de invocação da função?  
  
**Exercício 3**

Escreveu uma função chamada make\_money por sua conta. É necessário importar uma função com o mesmo nome do módulo mint e não quer renomear nenhum dos seus nomes previamente definidos. Qual a variante da declaração import que o pode ajudar com a questão?  
  
**Exercício 4**

Que forma de invocação da função make\_money é válida se o seu código começar com a seguinte linha?

from mint import \*

# Trabalhar com módulos padrão

Antes de começarmos a passar por alguns módulos Python padrão, queremos introduzir-lhe a função dir() . Não tem nada a ver com o comando dir que conhece de consolas Windows e Unix, visto dir() não mostrar o conteúdo de uma diretoria/pasta de disco, mas não há como negar que faz algo realmente semelhante - é capaz de revelar todos os nomes fornecidos através de um determinado módulo.

Há uma condição: o módulo tem de ter sido previamente importado como um todo (isto é, utilizar a import module instrução - from module não é suficiente).

A função devolve uma **lista ordenada alfabeticamente** contendo todos os nomes de entidades disponíveis no módulo identificados por um nome passado para a função como argumento:

dir(module)

Nota: se o nome do módulo tiver sido aliased, deve usar o alias, não o nome original.

A utilização da função dentro de um script regular não faz muito sentido, mas ainda assim é possível.

Por exemplo, pode executar o seguinte código para imprimir os nomes de todas as entidades dentro do math módulo:

import math

for name in dir(math):

print(name, end="\t")

O código de exemplo deve produzir o seguinte output:

\_\_doc\_\_ \_\_loader\_\_ \_\_name\_\_ \_\_package\_\_ \_\_spec\_\_ acos acosh asin asinh atan atan2 atanh ceil copysign cos cosh degrees e erf erfc exp expm1 fabs factorial floor fmod frexp fsum gamma hypot isfinite isinf isnan ldexp lgamma log log10 log1p log2 modf pi pow radians sin sinh sqrt tan tanh trunc

**output**

Já reparou nestes estranhos nomes que começam com \_\_ no topo da lista? Falaremos mais sobre eles quando falarmos sobre as questões relacionadas com a escrita dos seus próprios módulos.

Alguns dos nomes podem trazer memórias de lições de matemática, e provavelmente não terá quaisquer problemas em adivinhar o seu significado.

Utilizar a função dir() dentro de um código pode não parecer muito útil - normalmente quer saber o conteúdo de um determinado módulo antes de escrever e executar o código.

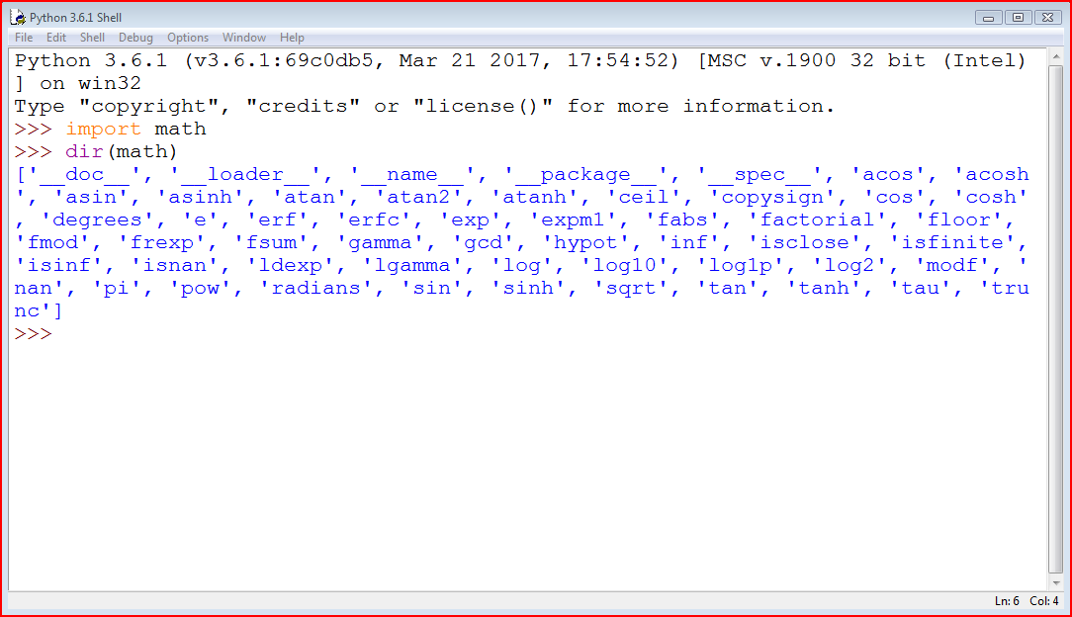
Felizmente, pode executar a função **diretamente na consola Python** (IDLE), sem necessidade de escrever e executar um script separado.

É assim que pode ser feito:

import math

dir(math)

Deve-se ver algo semelhante a isto:



**Funções selecionadas a partir do módulo math .**

Vamos começar com uma rápida pré-visualização de algumas das funções fornecidas pelo módulo math .

Escolhemo-los arbitrariamente, mas isso não significa que as funções que não mencionamos aqui sejam menos significativas. Mergulhe você mesmo nas profundezas dos módulos - não temos aqui espaço nem tempo para falar de tudo em pormenor.

O primeiro grupo das funções de mathestão relacionadas com **trigonometria**:

* sin(x) → o seno de x;
* cos(x) → o cosseno de x;
* tan(x) → a tangente de x.

Todas estas funções tomam um argumento (uma medição angular expressa em radianos) e devolvem o resultado apropriado (tenha cuidado com tan() - nem todos os argumentos são aceites).

Naturalmente, existem também as suas versões invertidas:

* asin(x) → o arco-seno de x;
* acos(x) → o arco-cosseno de x;
* atan(x) → o arco-tangente de x.

Estas funções tomam um argumento (atenção aos domínios) e devolvem uma medida de um ângulo em radianos.

Para operar eficazmente em medições de ângulos, o módulo math fornece-lhe as seguintes entidades:

* pi → uma constante com um valor que é uma aproximação de π;
* radians(x) → uma função que converte x de graus para radianos;
* degrees(x) → atuando na outra direção (dos radianos aos graus)

Agora veja o código no editor. O programa de exemplo não é muito sofisticado, mas consegue prever os seus resultados?

from math import pi, radians, degrees, sin, cos, tan, asin

ad = 90

ar = radians(ad)

ad = degrees(ar)

print(ad == 90.)

print(ar == pi / 2.)

print(sin(ar) / cos(ar) == tan(ar))

print(asin(sin(ar)) == ar)

Além das funções circulares (listadas acima) o módulo math também contém um conjunto dos seus **análogos hiperbólicos**:

* sinh(x) → o seno hiperbólico;
* cosh(x) → o cosseno hiperbólico;
* tanh(x) → a tangente hiperbólica;
* asinh(x) → o arco-seno hiperbólico;
* acosh(x) → o arco-cosseno hiperbólico;
* atanh(x) → o arco-tangente hiperbólico.

Outro grupo das funções de mathé formado por funções que estão ligadas à **exponenciação**:

* e → uma constante com um valor que é uma aproximação do número de Euler (e)
* exp(x) → encontrar o valor de ex;
* log(x) → o logaritmo natural de x
* log(x, b) → o logaritmo de x para base b
* log10(x) → o logaritmo decimal de x (mais preciso do que log(x, 10))
* log2(x) → o logaritmo binário de x (mais preciso do que log(x, 2))

Nota: a função pow() .

* pow(x, y) → encontrar o valor de xy (atenção aos domínios)

Esta é uma função integrada, e não tem de ser importada.

Veja o código no editor. Consegue prever o seu output?

from math import e, exp, log

print(pow(e, 1) == exp(log(e)))

print(pow(2, 2) == exp(2 \* log(2)))

print(log(e, e) == exp(0))

O último grupo é constituído por algumas funções de caráter geral, como por exemplo:

* ceil(x) → o teto de x (o menor inteiro maior ou igual a x)
* floor(x) → o piso de x (o maior número inteiro menor ou igual a x)
* trunc(x) → o valor de x truncado a um número inteiro (tenha cuidado - não é equivalente a um teto ou piso)
* factorial(x) → devolve x! (x tem de ser um integral e não um negativo)
* hypot(x, y) → devolve o comprimento da hipotenusa de um triângulo de ângulo retângulo com o comprimento das pernas igual a x e y (o mesmo que sqrt(pow(x, 2) + pow(y, 2)) mas mais preciso)

Veja o código no editor. Analise o programa com cuidado.

Demonstra as diferenças fundamentais entre ceil(), floor() e trunc().

Execute o programa e verifique o seu output.

from math import ceil, floor, trunc

x = 1.4

y = 2.6

print(floor(x), floor(y))

print(floor(-x), floor(-y))

print(ceil(x), ceil(y))

print(ceil(-x), ceil(-y))

print(trunc(x), trunc(y))

print(trunc(-x), trunc(-y))

# Existe uma verdadeira aleatoriedade nos computadores?

Outro módulo que vale a pena mencionar é o chamado random.

Fornece alguns mecanismos que lhe permitem operar com números **pseudo-aleatórios**.

  
Note o prefixo **pseudo** - os números gerados pelos módulos podem parecer aleatórios no sentido de que não se pode prever os seus valores subsequentes, mas não se esqueça que todos eles são calculados usando algoritmos muito refinados.

Os algoritmos não são aleatórios - são deterministas e previsíveis. Apenas os processos físicos que estão completamente fora do nosso controlo (como a intensidade da radiação cósmica) podem ser utilizados como fonte de dados aleatórios reais. Os dados produzidos por computadores determinísticos não podem ser aleatórios de forma alguma.

Um gerador de números aleatórios toma um valor chamado **seed**, trata-o como um valor de input, calcula um número "aleatório" com base nele (o método depende de um algoritmo escolhido) e produz um **novo valor de seed**.

O comprimento de um ciclo em que todos os valores de seed são únicos pode ser muito longo, mas não é infinito - mais cedo ou mais tarde os valores de seed começarão a repetir-se, e os valores geradores também se repetirão. Isto é normal. É uma característica, não um erro ou um bug.

O valor inicial da seed, definido durante o início do programa, determina a ordem em que os valores gerados aparecerão.

O fator aleatório do processo pode ser **aumentado através da definição da seed com um número retirado da hora atual** - isto pode assegurar que cada lançamento do programa começará a partir de um valor de seed diferente (ou seja, utilizará números aleatórios diferentes).

Felizmente, tal inicialização é feita pelo Python durante a importação do módulo.

**Funções selecionadas a partir do módulo random .**

**A função random .**

A função mais geral chamada random() (não confundir com o nome do módulo) **produz um número float**x**vindo do intervalo**(0.0, 1.0) - por outras palavras: (0,0 <= x < 1,0).

O programa de exemplo abaixo produzirá cinco valores pseudo-aleatórios - como os seus valores são determinados pelo valor atual (bastante imprevisível) da seed, não se pode adivinhá-los:

from random import random

for i in range(5):

print(random())

Execute o programa. Isto é o que obtivemos:

0.9535768927411208

0.5312710096244534

0.8737691983477731

0.5896799172452125

0.02116716297022092

**output de amostra**

**A seed .**

A função seed() é capaz de **definir diretamente a seed do gerador**. Vamos mostrar-lhe duas das suas variantes:

* seed() - define a seed com a hora atual;
* seed(int\_value) - define a seed com o valor inteiro int\_value.

Modificámos o programa anterior - com efeito, removemos qualquer vestígio de aleatoriedade do código:

from random import random, seed

seed(0)

for i in range(5):

print(random())

Devido ao facto de a seed ser sempre definida com o mesmo valor, a sequência de valores gerados parece sempre a mesma.

Execute o programa. Isto é o que obtivemos:

0.844421851525

0.75795440294

0.420571580831

0.258916750293

0.511274721369

**output de amostra**

E você?

Nota: os seus valores podem ser ligeiramente diferentes dos nossos se o seu sistema utilizar uma aritmética de floating-point mais ou menos precisa, mas a diferença será vista bastante longe do ponto decimal.

**As funções randrange e randint .**

Se quiser valores inteiros aleatórios, uma das seguintes funções encaixar-se-ia melhor:

* randrange(end)
* randrange(beg, end)
* randrange(beg, end, step)
* randint(left, right)

As três primeiras invocações gerarão um número inteiro retirado (pseudo aleatoriamente) do intervalo (respetivamente):

* range(end)
* range(beg, end)
* range(beg, end, step)

Note-se a **exclusão implícita do lado direito**!

A última função é um equivalente a randrange(left, right+1) - gera o valor inteiro i, que cai no intervalo [esquerda, direita] (sem exclusão do lado direito).

Veja o código no editor. Este programa de amostra produzirá consequentemente uma linha composta por três zeros e um zero ou um no quarto lugar.

from random import randrange, randint

print(randrange(1), end=' ')

print(randrange(0, 1), end=' ')

print(randrange(0, 1, 1), end=' ')

print(randint(0, 1))

As funções anteriores têm uma desvantagem importante - podem produzir valores repetitivos mesmo que o número de invocações subsequentes não seja maior do que a largura do intervalo especificado.

Veja o código abaixo - o programa produz muito provavelmente um conjunto de números em que alguns elementos não são únicos:

from random import randint

for i in range(10):

print(randint(1, 10), end=',')

Isto é o que obtivemos num dos lançamentos:

9,4,5,4,5,8,9,4,8,4,

**output de amostra**

**A choice e sample funções**

Como pode ver, esta não é uma boa ferramenta para gerar números numa lotaria. Felizmente, existe uma solução melhor do que escrever o seu próprio código para verificar a singularidade dos números "sorteados".

É uma função chamada de uma maneira muito sugestiva - choice:

* choice(sequence)
* sample(sequence, elements\_to\_choose)

A primeira variante escolhe um elemento "aleatório" a partir da sequência de input e devolve-o.

O segundo constrói uma lista (uma amostra; em inglês, uma sample) que consiste no elemento elements\_to\_choose “sorteado” a partir da sequência de input.

Por outras palavras, a função escolhe (em inglês, chooses) alguns dos elementos de input, devolvendo uma lista com a escolha. Os elementos da amostra são colocados em ordem aleatória. Nota: o elements\_to\_choose não deve ser maior do que o comprimento da sequência de input.

Veja o código abaixo:

from random import choice, sample

my\_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

print(choice(my\_list))

print(sample(my\_list, 5))

print(sample(my\_list, 10))

Novamente, o output do programa não é previsível. Os nossos resultados foram os seguintes:

4

[3, 1, 8, 9, 10]

[10, 8, 5, 1, 6, 4, 3, 9, 7, 2]

**Como saber onde se encontra?**

Por vezes, pode ser necessário descobrir informações não relacionadas com Python. Por exemplo, pode precisar de saber a localização do seu programa dentro do ambiente maior do computador.

Imagine o ambiente do seu programa como uma pirâmide constituída por uma série de camadas ou plataformas.



As camadas são:

* o seu código (em execução) está localizado na parte superior;
* O Python (mais precisamente - o seu ambiente de runtime) encontra-se diretamente abaixo;
* a camada seguinte da pirâmide é preenchida com o SO - o ambiente Python fornece algumas das suas funcionalidades utilizando os serviços do sistema operativo; o Python, embora muito poderoso, não é omnipotente - é forçado a utilizar muitos ajudantes se for processar ficheiros ou comunicar com dispositivos físicos;
* a camada mais inferior é o hardware - o processador (ou processadores), interfaces de rede, dispositivos de interface humana (ratos, teclados, etc.) e toda a outra maquinaria necessária para fazer funcionar o computador; o sistema operativo sabe como conduzi-lo, e usa muitos truques para conduzir todas as peças num ritmo consistente.

Isto significa que algumas das suas ações (ou melhor, do seu programa) têm de percorrer um longo caminho para serem executadas com sucesso - imagine que:

* **o seu código** quer criar um ficheiro, por isso invoca uma das funções de Python;
* O **Python** aceita a ordem, rearranja-a para cumprir os requisitos do SO local (é como colocar o selo "aprovado" no seu pedido) e envia-a (isto pode lembrá-lo de uma cadeia de comando)
* o **SO** verifica se o pedido é razoável e válido (por exemplo, se o nome do ficheiro está em conformidade com algumas regras de sintaxe) e tenta criar o ficheiro; tal operação, aparentemente muito simples, não é atómica - consiste em muitos pequenos passos dados pelo...
* **hardware**, que é responsável pela ativação de dispositivos de armazenamento (disco rígido, dispositivos de estado sólido, etc.) para satisfazer as necessidades do SO.

Normalmente, não está ciente de todo este alarido - quer que o ficheiro seja criado e pronto.

Mas por vezes deseja-se saber mais - por exemplo, o nome do SO que hospeda o Python, e algumas características que descrevem o hardware que hospeda o SO.

Existe um módulo que fornece alguns meios que lhe permitem saber onde está e que componentes trabalham para si. O módulo é chamado **platform**. Mostrar-lhe-emos algumas das funções que ele lhe proporciona.

**Funções selecionadas a partir do módulo platform .**

**A função platform .**

A função platform permite-lhe aceder aos dados da plataforma subjacente, ou seja, hardware, sistema operativo, e informação da versão do intérprete.

Existe uma função que lhe pode mostrar todas as camadas subjacentes num só relance, denominada platform, também. Apenas devolve uma string descrevendo o ambiente; assim, o seu output é mais dirigido ao ser humano do que ao processamento automatizado (vê-lo-á em breve).

É assim que pode invocá-la:

platform(aliased = False, terse = False)

E agora:

* aliased → quando definido para True (ou qualquer valor não nulo) pode fazer com que a função apresente os nomes alternativos das camadas subjacentes em vez dos nomes comuns;
* terse → quando definido para True (ou qualquer valor não nulo) pode convencer a função a apresentar uma forma mais breve do resultado (se possível)

Executámos o nosso programa de amostra utilizando três plataformas diferentes - isto é o que obtivemos:

**Intel x86 + Windows Vista (32 bit)**:

Windows-Vista-6.0.6002-SP2

Windows-Vista-6.0.6002-SP2

Windows-Vista

**output**

**Intel x86 + Gentoo Linux (64 bits)**:

Linux-3.18.62-g6-x86\_64-Intel-R-\_Core-TM-\_i3-2330M\_CPU\_@\_2.20GHz-with-gentoo-2.3

Linux-3.18.62-g6-x86\_64-Intel-R-\_Core-TM-\_i3-2330M\_CPU\_@\_2.20GHz-with-gentoo-2.3

Linux-3.18.62-g6-x86\_64-Intel-R-\_Core-TM-\_i3-2330M\_CPU\_@\_2.20GHz-with-glibc2.3.4

**output**

**Raspberry PI2+ Linux Raspbian (32 bits)**:

Linux-4.4.0-1-rpi2-armv7l-with-debian-9.0

Linux-4.4.0-1-rpi2-armv7l-with-debian-9.0

Linux-4.4.0-1-rpi2-armv7l-with-glibc2.9

**output**

Também pode executar o programa de amostra em IDLE na sua máquina local para verificar qual o output que terá.

**As funções machine .**

Por vezes, pode apenas querer saber o nome genérico do processador que executa o seu SO juntamente com o Python e o seu código - uma função chamada machine() dir-lhe-á isso. Como anteriormente, a função devolve uma string.

Mais uma vez, executámos o programa de amostra em três plataformas diferentes:

**Intel x86 + Windows Vista (32 bit)**:

x86

**output**

**Intel x86 + Gentoo Linux (64 bits)**:

x86\_64

**output**

**Raspberry PI2+ Linux Raspbian (32 bits)**:

armv7l

**As funções processor .**

A função processor() devolve uma string preenchida com o nome do processador real (se possível).

Mais uma vez, executámos o programa de amostra em três plataformas diferentes:

**Intel x86 + Windows Vista (32 bit)**:

x86

**output**

**Intel x86 + Gentoo Linux (64 bits)**:

Intel(R) Core(TM) i3-2330M CPU @ 2.20GHz

**output**

**Raspberry PI2+ Linux Raspbian (32 bits)**:

armv7l

**output**

Teste isto na sua máquina local.

**As funções system .**

Uma função chamada system() devolve o nome genérico do SO como uma cadeia.

As nossas plataformas de exemplo apresentaram-se desta forma:

**Intel x86 + Windows Vista (32 bit)**:

Windows

**output**

**Intel x86 + Gentoo Linux (64 bits)**:

Linux

**output**

**Raspberry PI2+ Linux Raspbian (32 bits)**:

Linux

**output**

**As funções version função**

A versão do SO é fornecida como uma string pela version() função.

Execute o código e verifique o seu output. Isto é o que obtivemos:

**Intel x86 + Windows Vista (32 bit)**:

6.0.6002

**output**

**Intel x86 + Gentoo Linux (64 bits)**:

#1 SMP PREEMPT Fri Jul 21 22:44:37 CEST 2017

**output**

**Raspberry PI2+ Linux Raspbian (32 bits)**:

#1 SMP Debian 4.4.6-1+rpi14 (2016-05-05)

**output**

**As funções python\_implementation e python\_version\_tuple .**

Se precisar de saber qual a versão do Python que está a executar o seu código, pode verificá-la utilizando várias funções dedicadas - aqui estão duas delas:

* python\_implementation() → devolve uma string denotando a implementação do Python (esperar CPython aqui, a menos que decida utilizar qualquer ramo de Python não canónico)

* python\_version\_tuple() → devolve um tuple de três elementos preenchido com:
  + a parte **principal** da versão do Python
  + a parte **secundária**;
  + o número do **patch** level.

O nosso programa de exemplo produziu o seguinte output:

CPython

3

7

7

**output de amostra**

É muito provável que a sua versão do Python seja diferente.

# Index do Módulo Python

Apenas cobrimos aqui as noções básicas dos módulos do Python. Os módulos do Python criam o seu próprio universo, no qual o Python é apenas uma galáxia, e ousaríamos dizer que explorar as profundezas destes módulos pode levar significativamente mais tempo do que conhecer o Python "puro".

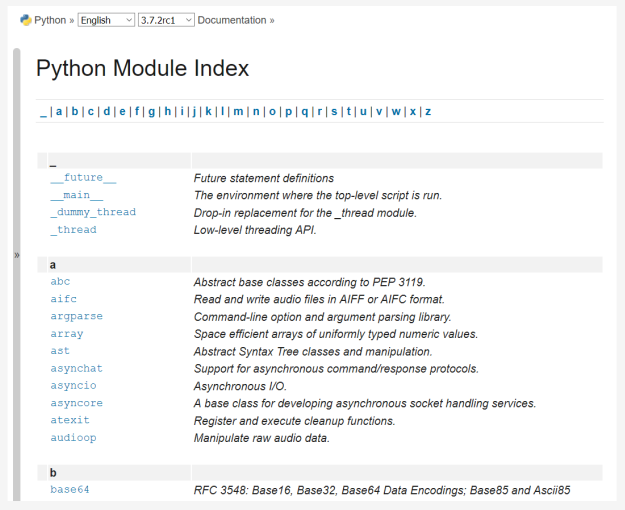
Além disso, a comunidade Python em todo o mundo cria e mantém centenas de módulos adicionais utilizados em aplicações de nicho como a genética, a psicologia, ou mesmo a astrologia.

Estes módulos não são (nem serão) distribuídos juntamente com o Python, nem através dos canais oficiais, o que torna o universo Python mais amplo - quase infinito.

Pode ler sobre todos os módulos Python padrão aqui: <https://docs.python.org/3/py-modindex.html>.

Não se preocupe - não vai precisar de todos estes módulos. Muitos deles são muito específicos.

Tudo o que precisa de fazer é encontrar os módulos que deseja, e ensinar-se a usá-los. É fácil.

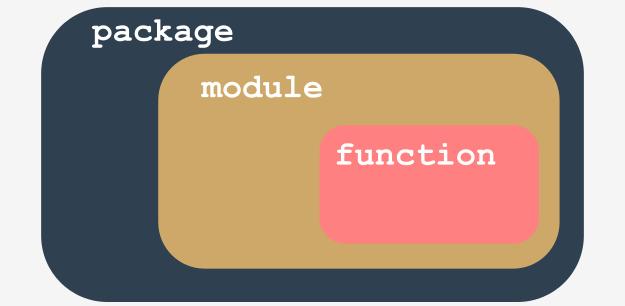


Na secção seguinte vamos dar uma vista de olhos a outra coisa. Vamos mostrar-lhe como escrever o seu próprio módulo.

**O que é um pacote?**

Escrever os seus próprios módulos não difere muito da escrita de guiões normais.

Há alguns aspetos específicos de que deve estar ciente, mas definitivamente não é ciência de foguetões. Verá isto em breve.



Vamos resumir algumas questões importantes:

* um **módulo é uma espécie de recipiente cheio de funções** - pode embalar tantas funções quantas quiser num único módulo e distribuí-lo pelo mundo inteiro;
* claro, é geralmente uma boa ideia não misturar funções com diferentes áreas de aplicação dentro de um módulo (tal como numa biblioteca - ninguém espera que as obras científicas sejam colocadas entre os livros de banda desenhada), por isso, agrupe as suas funções cuidadosamente e nomeie o módulo que as contém de uma forma clara e intuitiva (por exemplo, não dê o nome arcade\_games a um módulo contendo funções destinadas a particionar e formatar discos rígidos)
* fazer muitos módulos pode causar uma pequena confusão - mais cedo ou mais tarde vai querer **agrupar os seus módulos** exatamente da mesma forma que anteriormente agrupou funções - existe um recipiente mais geral do que um módulo?
* sim, existe - é um **pacote**; no mundo dos módulos, um pacote desempenha um papel semelhante a uma pasta/diretoria no mundo dos ficheiros.

# O seu primeiro módulo: passo 1

Nesta secção vai estar a trabalhar localmente na sua máquina. Vamos começar do zero. Crie um ficheiro vazio, tal como este:



module.py

Serão necessários dois ficheiros para repetir estas experiências. O primeiro deles será o próprio módulo. Está vazio por agora. Não se preocupe, vai preenchê-lo em breve com código real.

Nomeámos o ficheiro module.py. Não é muito criativo, mas é simples e claro.

# O seu primeiro módulo: passo 2

O segundo ficheiro contém o código utilizando o novo módulo. O seu nome é main.py. O seu conteúdo é muito breve até agora:



import module

main.py

Nota: **ambos os ficheiros têm de estar localizados na mesma pasta**. Encorajamo-lo vivamente a criar uma pasta nova, e vazia, para ambos os ficheiros. Algumas coisas serão então mais fáceis.

Inicie o IDLE (ou qualquer outro IDE que prefira) e execute o main.py . O que vê?

Não deveria ver nada. Isto significa que o Python importou com sucesso o conteúdo do module.py .

Não importa que o módulo esteja vazio por agora. O primeiro passo já foi dado, mas antes de dar o passo seguinte, queremos que dê uma vista de olhos à pasta na qual ambos os ficheiros existem.

Repara em algo interessante?

Apareceu uma nova subpasta - consegue vê-la? O seu nome é \_\_pycache\_\_. Dê uma vista de olhos no interior. O que vê?

Há um ficheiro com o nome (mais ou menos) module.cpython-xy.pyc onde x e y são dígitos derivados da sua versão de Python (por exemplo, eles serão 3 e 8 se utilizar o Python 3.8).

O nome do ficheiro é o mesmo que o nome do seu módulo (módulo aqui). A parte após o primeiro ponto diz qual a implementação Python que criou o ficheiro (CPython aqui) e o seu número de versão. A última parte (pyc) vem das palavras *Python* e *compiled*.

Pode ver o interior do ficheiro - o conteúdo é completamente ilegível para os seres humanos. Tem de ser assim, uma vez que o ficheiro se destina apenas à utilização do Python.

Quando o Python importa um módulo pela primeira vez, ele **traduz o seu conteúdo numa forma algo compilada**.

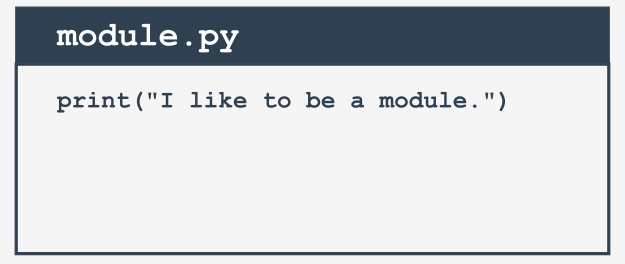
O ficheiro não contém machine code - é código interno Python **semi-compilado**, pronto para ser executado pelo intérprete Python. Como tal, um ficheiro não requer muitas das verificações necessárias para um source file puro, a execução começa mais rapidamente, e corre também mais rapidamente.

Graças a isso, cada importação subsequente será mais rápida do que a interpretação do source text a partir do zero.

Python é capaz de verificar se o source file do módulo foi modificado (neste caso, o ficheiro pyc será reconstruído) ou não (quando o ficheiro pyc pode ser executado de uma só vez). Como este processo é totalmente automático e transparente, não precisa de o ter em mente.

# O seu primeiro módulo: passo 3

Agora pusemos um pouco de algo no ficheiro do módulo:



print("I like to be a module.")

module.py

Consegue notar alguma diferença entre um módulo e um script comum? Não há nenhuma até agora.

É possível executar este ficheiro como qualquer outro script. Experimente você mesmo.

O que acontece? Deverá ver a seguinte linha dentro da sua consola:

I like to be a module.

**output**

# O seu primeiro módulo: passo 4

Vamos voltar ao ficheiro main.py modificado.



import module

main.py

Execute-o. O que vê? Esperamos que veja algo como isto:

I like to be a module.

**output**

O que significa realmente?

Quando um módulo é importado, o seu conteúdo é **implicitamente executado pelo Python**. Dá ao módulo a oportunidade de inicializar alguns dos seus aspetos internos (por exemplo, pode atribuir algumas variáveis com valores úteis).

Nota: **a inicialização ocorre apenas uma vez**, quando ocorre a primeira importação, pelo que as atribuições feitas pelo módulo não são repetidas desnecessariamente.

Imagine o seguinte contexto:

* há um módulo chamado mod1;
* há um módulo chamado mod2 que contém a instrução import mod1 ;
* há um ficheiro principal contendo as instruções import mod1 e import mod2 .

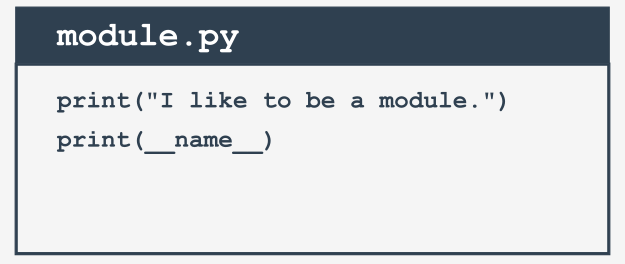
À primeira vista, pode pensar que o mod1 será importado duas vezes - felizmente, **apenas a primeira importação ocorre**. O Python lembra-se dos módulos importados e silenciosamente omite todas as importações subsequentes.

# O seu primeiro módulo: passo 5

O Python pode fazer muito mais. Ele também cria uma variável chamada \_\_name\_\_.

Além disso, cada source file utiliza a sua própria versão separada da variável - não é partilhada entre módulos.

Vamos mostrar-lhe como usá-la. Modifique um pouco o módulo:



print("I like to be a module.")

print(\_\_name\_\_)

module.py

Agora execute o ficheiro module.py . Deve ver as seguintes linhas:

I like to be a module

\_\_main\_\_

**output**

Agora execute o ficheiro main.py . E? Vê o mesmo que nós?

I like to be a module

module

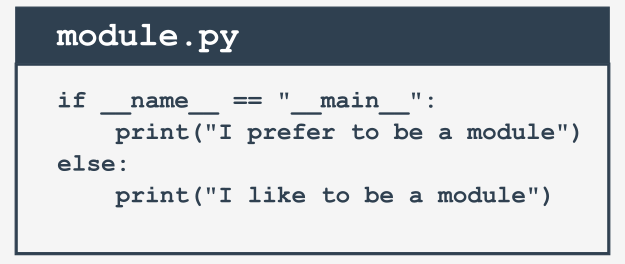
**output**

Podemos dizer que:

* quando se executa diretamente um ficheiro, a sua variável \_\_name\_\_ é definida como \_\_main\_\_;
* quando um ficheiro é importado como um módulo, a sua variável \_\_name\_\_ é definida como o nome do ficheiro (excluindo .py)

# O seu primeiro módulo: passo 6

É assim que pode fazer uso da variável \_\_main\_\_ para detetar o contexto em que o seu código foi ativado:



if \_\_name\_\_ == "\_\_main\_\_":

print("I prefer to be a module.")

else:

print("I like to be a module.")

module.py

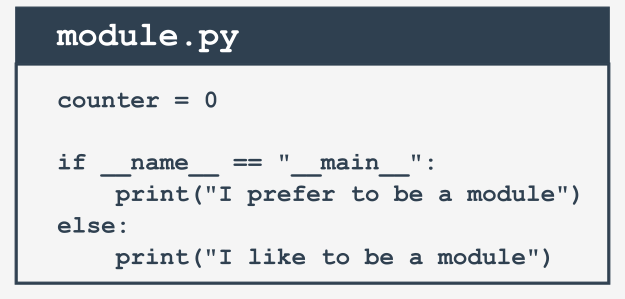
No entanto, há uma forma mais inteligente de utilizar a variável. Se escrever um módulo preenchido com várias funções complexas, pode utilizá-lo para colocar uma série de testes para verificar se as funções funcionam corretamente.

Cada vez que modificar qualquer uma destas funções, pode simplesmente executar o módulo para se certificar de que as suas alterações não estragaram o código. Estes testes serão omitidos quando o código for importado como um módulo.

# O seu primeiro módulo: passo 7

Este módulo conterá duas funções simples, e se quiser saber quantas vezes as funções foram invocadas, precisa de um contador inicializado a zero quando o módulo estiver a ser importado.

Pode fazê-lo desta maneira:



counter = 0

if \_\_name\_\_ == "\_\_main\_\_":

print("I prefer to be a module.")

else:

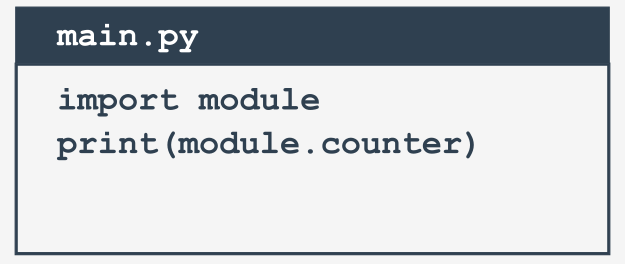
print("I like to be a module.")

module.py

# O seu primeiro módulo: passo 8

A introdução de tal variável é absolutamente correta, mas pode causar **efeitos secundários** importantes de que deve estar ciente.

Dê uma vista de olhos ao ficheiro main.py modificado.



import module

print(module.counter)

main.py

Como se pode ver, o ficheiro principal tenta aceder à variável counter do módulo. Isto é legal? Sim, é. É utilizável? Pode ser muito utilizável. É seguro?

Depende - se confiar nos utilizadores do seu módulo, não há problema; no entanto, pode não querer que o resto do mundo veja a sua **variável pessoal/privada**.

Ao contrário de muitas outras linguagens de programação, o Python não tem meios de lhe permitir esconder tais variáveis dos olhos dos utilizadores do módulo.

Apenas pode informar os seus utilizadores que esta é a sua variável, que podem lê-la, mas que não a devem modificar em circunstância alguma.

Isto é feito precedendo o nome da variável com \_ (um underscore) ou \_\_ (dois underscores), mas lembre-se, é apenas uma **convenção**. Os utilizadores do seu módulo podem obedecer-lhe ou não.

Claro, seguiremos a convenção. Agora vamos colocar duas funções no módulo - elas vão avaliar a soma e o produto dos números recolhidos numa lista.

Além disso, vamos acrescentar alguns ornamentos e remover quaisquer restos supérfluos.

# O seu primeiro módulo: passo 9

Muito bem. Vamos escrever um código novinho em folha no nosso ficheiro module.py . O módulo atualizado está pronto aqui:

#!/usr/bin/env python3

""" module.py - an example of a Python module """

\_\_counter = 0

def suml(the\_list):

global \_\_counter

\_\_counter += 1

the\_sum = 0

for element in the\_list:

the\_sum += element

return the\_sum

def prodl(the\_list):

global \_\_counter

\_\_counter += 1

prod = 1

for element in the\_list:

prod \*= element

return prod

if \_\_name\_\_ == "\_\_main\_\_":

print("I prefer to be a module, but I can do some tests for you.")

my\_list = [i+1 for i in range(5)]

print(suml(my\_list) == 15)

print(prodl(my\_list) == 120)

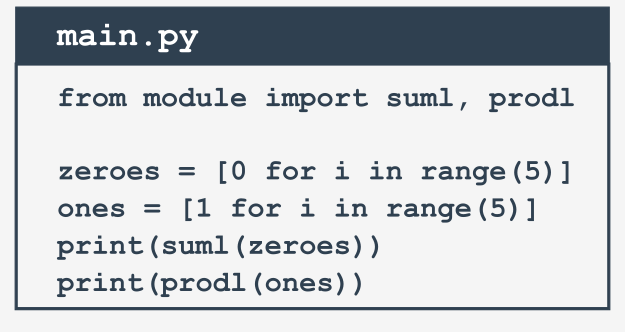
module.py

Alguns elementos precisam de alguma explicação, pensamos nós:

* a linha começando com #! tem muitos nomes - pode ser chamada *shabang*, *shebang*, *hashbang*, *poundbang* ou até mesmo *hashpling* (não nos pergunte porquê). O nome em si não significa nada aqui - o seu papel é mais importante. Do ponto de vista do Python, é apenas um **comentário** visto começar com #. Para Unix e Unix-like SOs (incluindo MacOS) tal linha **instrui o SO a como executar o conteúdo do ficheiro** (por outras palavras, que programa precisa de ser lançado para interpretar o texto). Em alguns ambientes (especialmente aqueles ligados a servidores web) a ausência dessa linha causará problemas;
* uma string (talvez uma multilinha) colocada antes de qualquer instrução de módulo (incluindo importações) é chamada a **doc-string**, e deve explicar brevemente a finalidade e o conteúdo do módulo;
* as funções definidas dentro do módulo (suml() e prodl()) estão disponíveis para importação;
* utilizámos a variável \_\_name\_\_ para detetar quando o ficheiro é executado de forma autónoma, e aproveitou esta oportunidade para realizar alguns testes simples.

# O seu primeiro módulo: passo 10

Agora é possível usar o módulo atualizado - esta é uma maneira:



from module import suml, prodl

zeroes = [0 for i in range(5)]

ones = [1 for i in range(5)]

print(suml(zeroes))

print(prodl(ones))

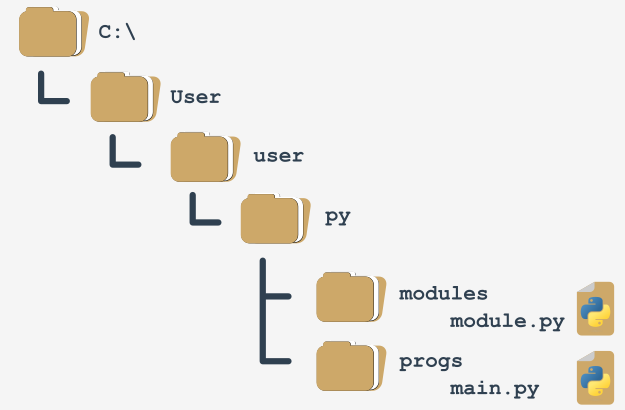
main.py

# O seu primeiro módulo: passo 11

É altura de tornar o nosso exemplo mais complicado - até agora assumimos que o ficheiro Python principal está localizado na mesma pasta/diretoria que o módulo a ser importado.

Vamos desistir desta suposição e realizar a seguinte experiência de pensamento:

* estamos a utilizar Windows ® OS (esta suposição é importante, pois a forma do nome do ficheiro depende disso)
* o script Python principal está em C:\Users\user\py\progs e é chamado main.py
* o módulo a ser importado está localizado em C:\Users\user\py\modules



Como lidar com isto?

Para responder a esta pergunta, temos de falar sobre **a forma como o Python procura módulos**. Há uma variável especial (na realidade uma lista) que armazena todos os locais (pastas/diretorias) que são pesquisados a fim de encontrar um módulo que tenha sido solicitado pela instrução de importação.

O Python navega por estas pastas pela ordem em que estão listadas na lista - se o módulo não puder ser encontrado em nenhuma destas diretorias, a importação falha.

Caso contrário, a primeira pasta contendo um módulo com o nome desejado será tomada em consideração (se alguma das restantes pastas contiver um módulo com esse nome, será ignorada).

A variável é chamada path, e é acessível através do módulo chamado sys. É assim que se pode verificar o seu valor regular:

import sys

for p in sys.path:

print(p)

Lançámos o código dentro da pasta C:\User\user , e isto é o que obtivemos:

C:\Users\user

C:\Users\user\AppData\Local\Programs\Python\Python36-32\python36.zip

C:\Users\user\AppData\Local\Programs\Python\Python36-32\DLLs

C:\Users\user\AppData\Local\Programs\Python\Python36-32\lib

C:\Users\user\AppData\Local\Programs\Python\Python36-32

C:\Users\user\AppData\Local\Programs\Python\Python36-32\lib\site-packages

**output de amostra**

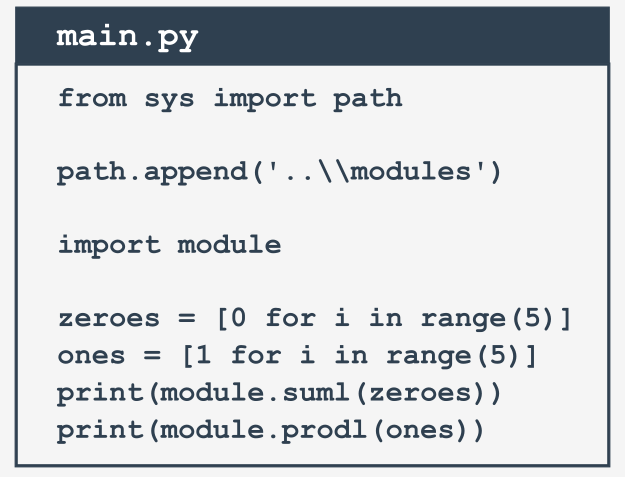
Nota: a pasta em que a execução começa é listada no **elemento do primeiro caminho**.

Nota mais uma vez: há um ficheiro zip listado como um dos elementos do caminho - não é um erro. O Python é capaz de tratar ficheiros zip como pastas normais - isto pode poupar muito armazenamento.

Consegue descobrir como podemos resolver o nosso problema agora? Podemos adicionar uma pasta contendo o módulo à variável path (é totalmente modificável).

# O seu primeiro módulo: passo 12

Uma das várias soluções possíveis parece-se com esta:



from sys import path

path.append('..\\modules')

import module

zeroes = [0 for i in range(5)]

ones = [1 for i in range(5)]

print(module.suml(zeroes))

print(module.prodl(ones))

main.py

Nota:

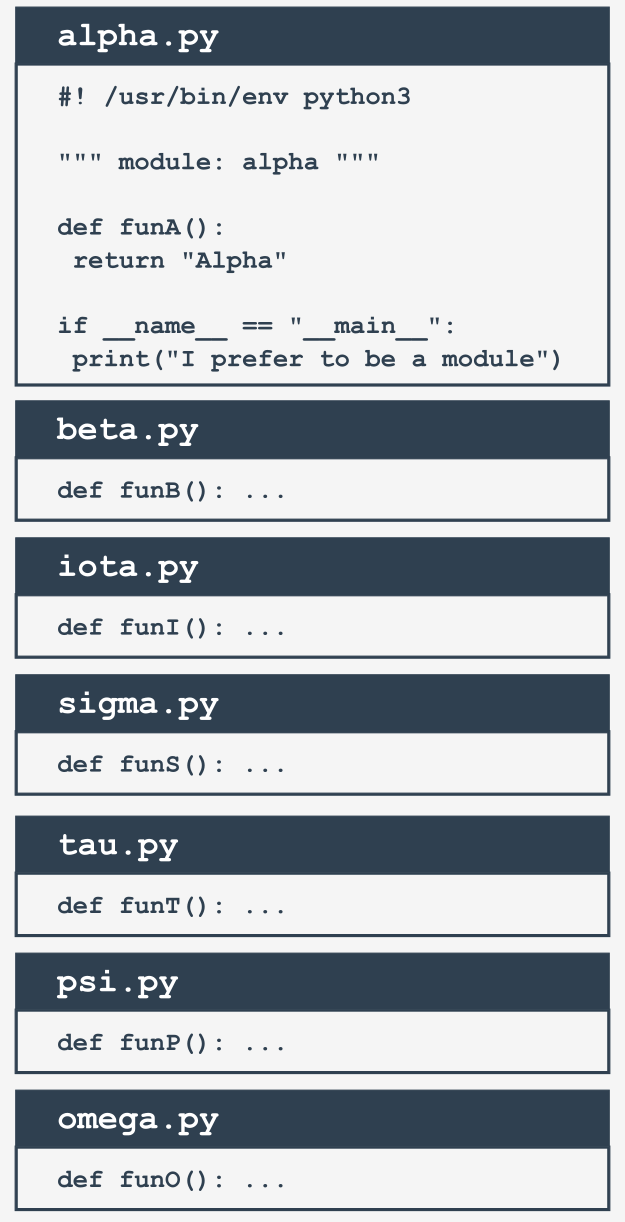
* duplicámos o \ nome da pasta interna - sabe porquê?  
    
  Verifique
* usámos o nome relativo da pasta - isto funcionará se iniciar o ficheiro main.py diretamente da sua home folder, e não funcionará se a diretoria atual não se ajustar ao caminho relativo; pode sempre usar um caminho absoluto, como este:  
    
  path.append('C:\\Users\\user\\py\\modules')

* utilizámos o método append() - com efeito, o novo caminho ocupará o último elemento da lista de caminhos; se não gostar da ideia, pode usar insert() em vez disso.

# O seu primeiro pacote: passo 1

Imagine que num futuro não tão distante, você e os seus associados escrevem um grande número de funções Python.

A sua equipa decide agrupar as funções em módulos separados, e este é o resultado final da ordenação:



#! /usr/bin/env python3

""" module: alpha """

def funA():

return "Alpha"

if \_\_name\_\_ == "\_\_main\_\_":

print("I prefer to be a module.")

alpha.py

Nota: apresentámos todo o conteúdo apenas para o módulo alpha.py - assumir que todos os módulos são semelhantes (eles contêm uma função chamada funX, onde *X* é a primeira letra do nome do módulo).

# O seu primeiro pacote: passo 2

De repente, alguém repara que estes módulos formam a sua própria hierarquia, pelo que colocá-los todos numa estrutura plana não será uma boa ideia.

Após alguma discussão, a equipa chega à conclusão de que os módulos têm de ser agrupados. Todos os participantes concordam que a seguinte estrutura em árvore reflete perfeitamente as relações mútuas entre os módulos:



Vamos rever isto de baixo para cima:

* o grupo ugly contém dois módulos: psi e omega;
* o grupo best contém dois módulos: sigma e tau;
* o grupo good contém dois módulos (alfa e beta) e um subgrupo (best)
* o grupo extra contém dois subgrupos (good e bad) e um módulo (iota)

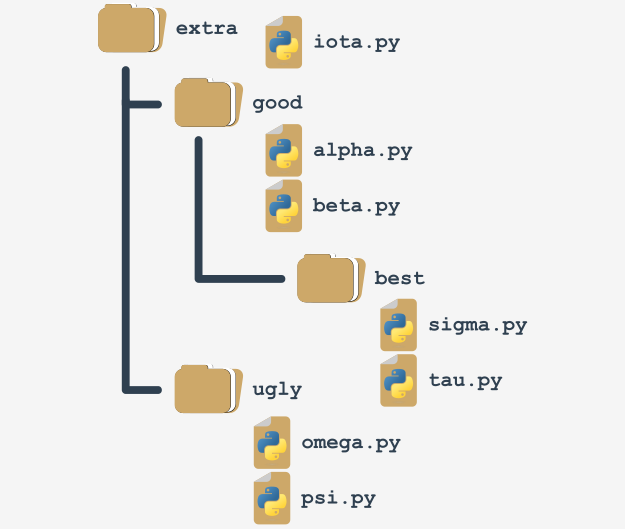
Tem mau aspecto? De modo algum - analise cuidadosamente a estrutura. Assemelha-se a algo, não é?

Parece ser uma **estrutura de diretoria**.

Vamos construir uma árvore refletindo as dependências projetadas entre os módulos.

# O seu primeiro pacote: passo 3

Esta é a aparência atual da árvore:



Tal estrutura é quase um pacote (no sentido Python do termo). Falta-lhe o detalhe de ser ao mesmo tempo funcional e operacional. Vamos completá-lo num momento.

Se assumir que extra é o nome de um **pacote recém-criado** (pense nele como a **raiz do pacote**), ele imporá uma regra de nomenclatura que lhe permite nomear claramente cada entidade da árvore.

Por exemplo:

* a localização de uma função chamada funT() a partir do pacote tau pode ser descrito como:
* extra.good.best.tau.funT()
* uma função marcada como:
* extra.ugly.psi.funP()

vem do módulo psi a ser armazenado no ugly subpacote do pacote extra .

# O seu primeiro pacote: passo 4

Há duas perguntas a responder:

* **como** transforma tal árvore (na realidade, uma subárvore) num **pacote** Python real (por outras palavras, como convencer o Python de que tal árvore não é apenas um monte de junk files, mas um conjunto de módulos)?
* **onde** coloca a subárvore para a tornar acessível ao Python?

A primeira pergunta tem uma resposta surpreendente: **os pacotes, como os módulos, podem precisar de inicialização**.

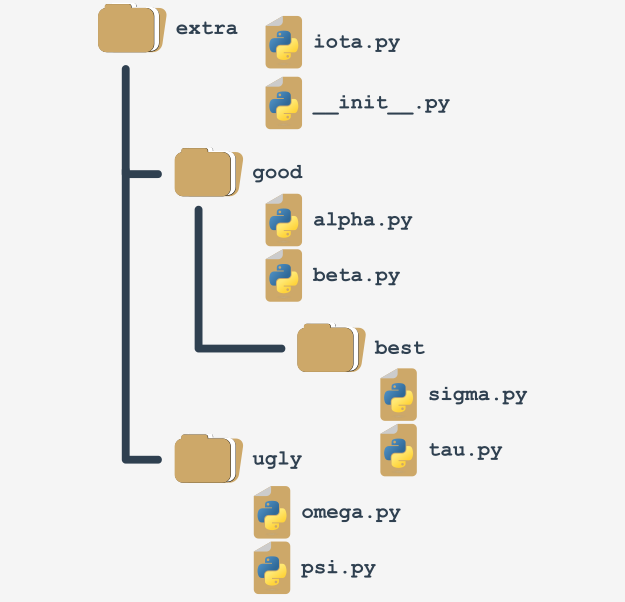
A inicialização de um módulo é feita por um *código unbound (não vinculado) (não faz parte de nenhuma função) localizado dentro do ficheiro do módulo. Como um pacote não é um ficheiro, esta técnica é inútil para inicializar pacotes.*

*Em vez disso, é preciso usar um truque diferente - o Python espera que haja um ficheiro com um nome muito único dentro da pasta do pacote: \_\_init\_\_.py.*

*O conteúdo do ficheiro é executado quando qualquer um dos módulos do pacote é****importado****. Se não quiser inicializações especiais, pode deixar o ficheiro vazio, mas não o deve omitir.*

# O seu primeiro pacote: passo 5

Lembre-se: **a presença do ficheiro \_\_init.py\_\_ finalmente compõe o pacote**.



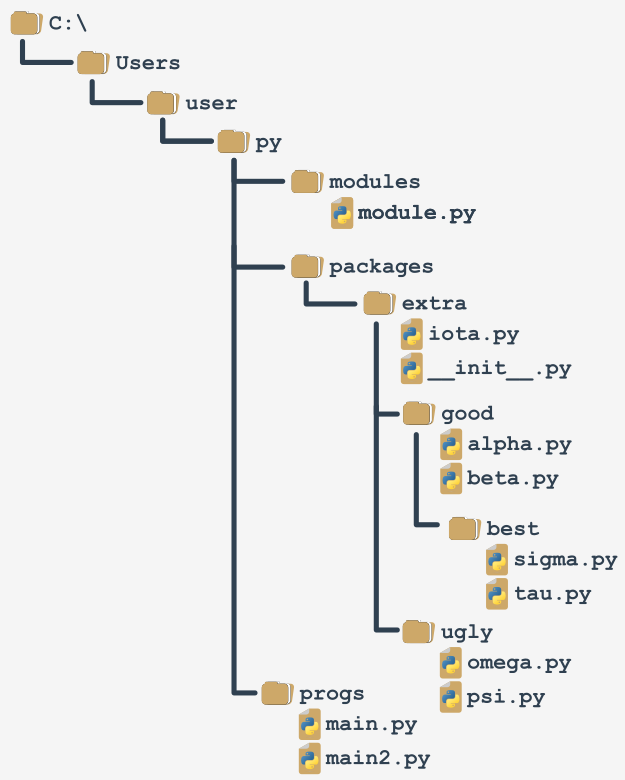
Nota: não é apenas a *root* file que pode conter o ficheiro \_\_init.py\_\_ - também pode colocá-lo dentro de qualquer uma das suas subpastas (subpacotes). Pode ser útil se alguns dos subpacotes requerem tratamento individual e tipos especiais de inicialização.

Agora está na altura de responder à segunda pergunta - a resposta é simples: **em qualquer lugar**. Só precisa de garantir que o Python esteja ciente da localização do pacote. Já sabe como fazê-lo.

Está pronto para fazer uso do seu primeiro pacote.

# O seu primeiro pacote: passo 6

Vamos supor que o ambiente de trabalho tem o seguinte aspeto:



Preparámos um ficheiro zip contendo todos os ficheiros do ramo de pacotes. Pode descarregá-lo e usá-lo nas suas próprias experiências, mas **lembre-se de descompactá-lo na pasta apresentada no esquema**, caso contrário ele não estará acessível ao código a partir do main file.

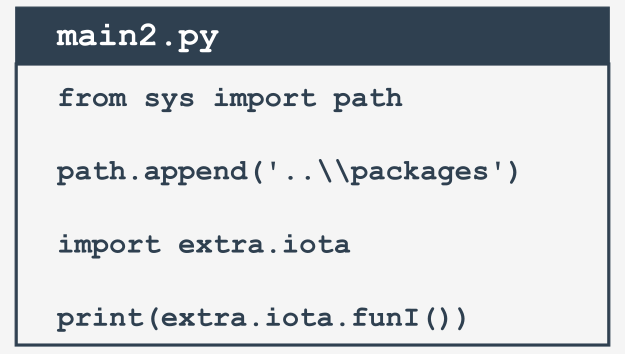
**DOWNLOAD** [Modules and Packages ZIP file](https://edube.org/uploads/media/default/0001/01/39a03067f85a9b39c0c94ca50b5d445b7df18a04.zip)

Continuará as suas experiências utilizando o main2.py ficheiro.

**O seu primeiro pacote: passo 7**

Vamos aceder à função funI() do módulo iota a partir do topo do pacote extra . Isto obriga-nos a utilizar nomes de pacotes qualificados (associe isto com a nomeação de pastas e subpastas - as convenções são muito semelhantes).

Esta é a forma de o fazer:



from sys import path

path.append('..\\packages')

import extra.iota

print(extra.iota.funI())

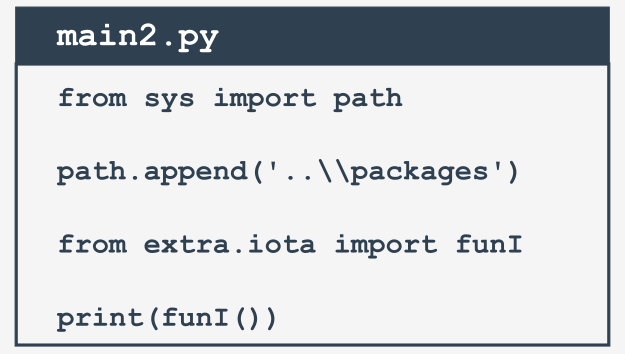
main2.py

Nota:

* modificámos a variável path para a tornar acessível ao Python;
* o ramo import não aponta diretamente para o módulo, mas especifica o caminho totalmente qualificado a partir do topo do pacote;

substituir import extra.iota por import iota causará um erro.

A seguinte variante também é válida:



from sys import path

path.append('..\\packages')

from extra.iota import funI

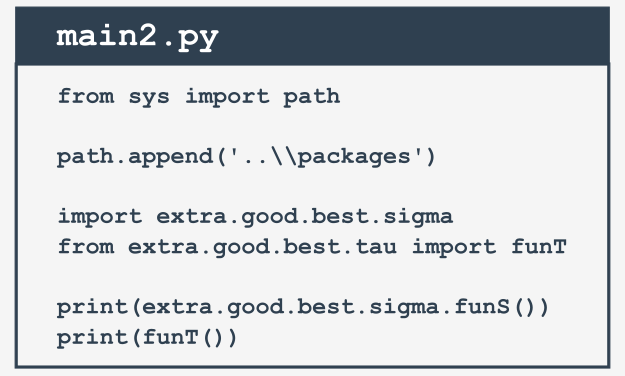
print(funI())

main2.py

Observe o nome qualificado do módulo iota .

# O seu primeiro pacote: passo 8

Vamos agora chegar até ao fundo da árvore - é assim que se pode ter acesso aos módulos sigma e tau :



from sys import path

path.append('..\\packages')

import extra.good.best.sigma

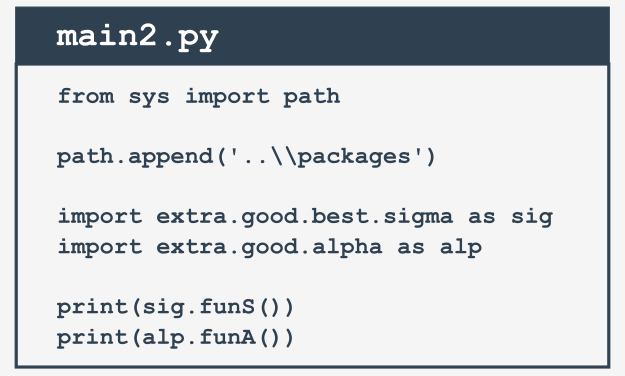
from extra.good.best.tau import funT

print(extra.good.best.sigma.funS())

print(funT())

main2.py

Pode tornar a sua vida mais fácil através da utilização de aliasing:



from sys import path

path.append('..\\packages')

import extra.good.best.sigma as sig

import extra.good.alpha as alp

print(sig.funS())

print(alp.funA())

main2.py

# O seu primeiro pacote: passo 9

Vamos assumir que zipámos toda a subdiretoria, a partir da pasta extra (incluindo a pasta), e vamos obter um ficheiro com o nome extrapack.zip. A seguir, colocamos o ficheiro dentro da pasta packages .

Agora podemos utilizar o ficheiro zip com o papel de pacotes:

from sys import path

path.append('..\\packages\\extrapack.zip')

import extra.good.best.sigma as sig

import extra.good.alpha as alp

from extra.iota import funI

from extra.good.beta import funB

print(sig.funS())

print(alp.funA())

print(funI())

print(funB())

main2.py

Se quiser realizar as suas próprias experiências com o pacote que criámos, pode descarregá-lo em baixo. Encorajamo-lo a fazê-lo.

**DOWNLOAD** [Extrapack ZIP file](https://edube.org/uploads/media/default/0001/01/d9df38daa0410952c4cbf85b47892954c45b9215.zip" \t "_blank)

Agora pode criar módulos e combiná-los em pacotes. É tempo de iniciar uma discussão completamente diferente - sobre erros, falhas e crashes.