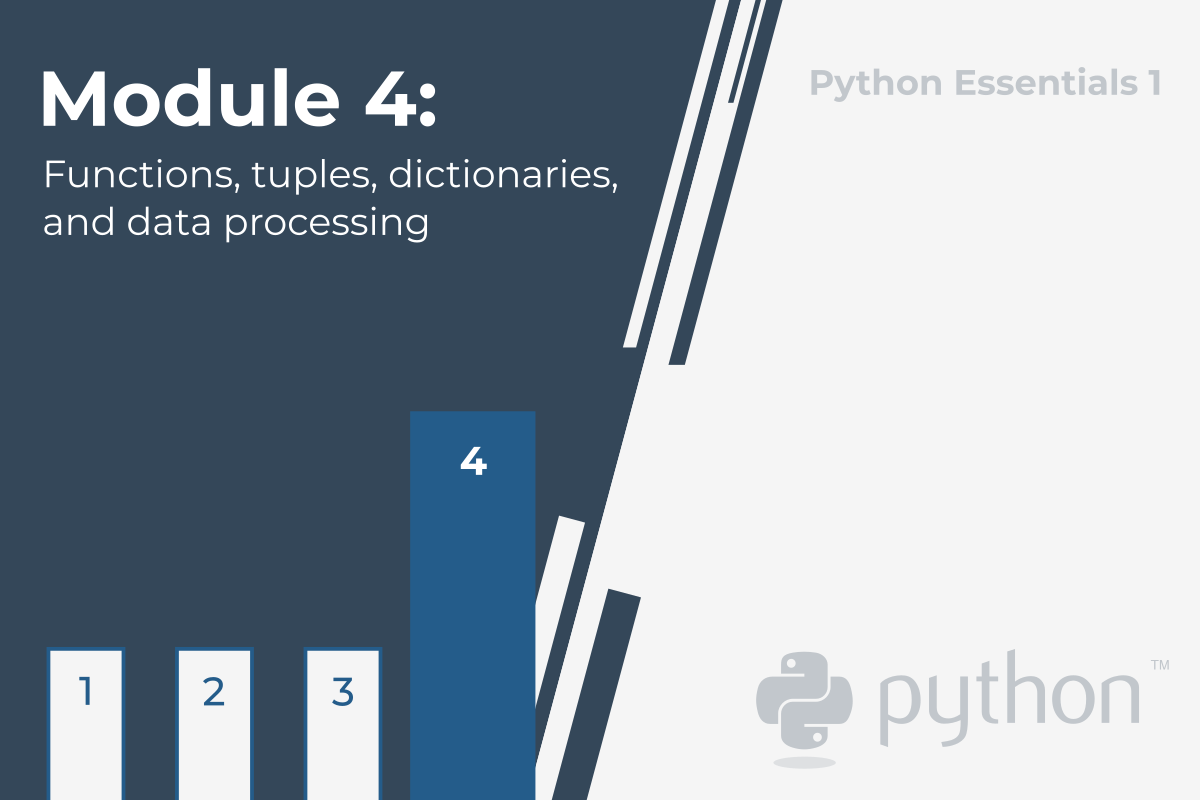
**Python Essentials 1:  
Módulo 4**

**Funções, Tuples, Dicionários e Processamento de Dados**

Neste módulo, cobrirá os seguintes tópicos:

* estruturação de códigos e o conceito de função;
* invocação de função e devolução do resultado de uma função;
* scopes do nome e sombreamento variável;
* tuples e o seu propósito, construção e utilização de tuples;
* dicionários e o seu propósito, construção e utilização de dicionários.



# Porque é que precisamos de funções?

Já se deparou com **funções** muitas vezes até agora, mas a opinião sobre os seus méritos que lhe demos tem sido bastante unilateral. Apenas invocou as funções utilizando-as como ferramentas para facilitar a vida, e para simplificar tarefas demoradas e enfadonhas.

Quando se pretende que alguns dados sejam impressos na consola, utiliza-se print(). Quando se quer ler o valor de uma variável, usa input(), juntamente com int() ou float().

Também fez uso de alguns **métodos**, que de facto funcionam, mas que são declarados de uma forma muito específica.

Agora aprenderá a escrever as suas próprias funções, e como utilizá-las. Escreveremos em conjunto várias funções, desde as muito simples até às bastante complexas, o que exigirá a sua concentração e atenção.

Acontece frequentemente que uma determinada peça de código é **repetida muitas vezes no seu programa**. Repete-se literalmente, ou com apenas algumas pequenas modificações, consistindo na utilização de outras variáveis no mesmo algoritmo. Acontece também que um programador não resiste a simplificar o trabalho, e começa a clonar tais peças de código utilizando as operações de clipboard e de copiar-colar.

Pode acabar por ser tão frustrante quando de repente se descobre que houve um erro no código clonado. O programador terá muito trabalho para encontrar todos os sítios que necessitam de correções. Há também um risco elevado de as correções causarem erros.

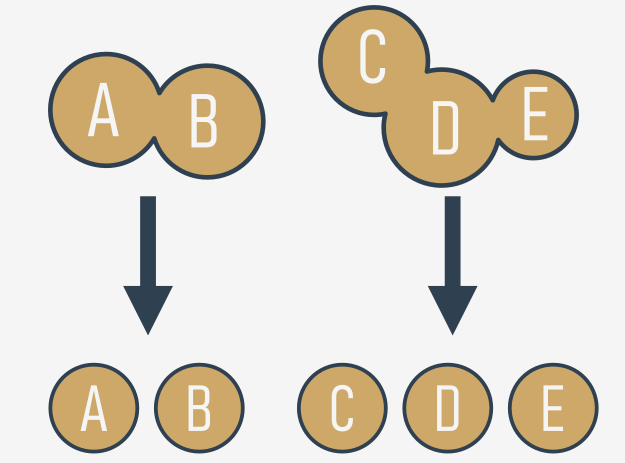
Podemos agora definir a primeira condição que pode ajudá-lo a decidir quando começar a escrever as suas próprias funções: **se um determinado fragmento do código começar a aparecer em mais do que um lugar, considerar a possibilidade de o isolar sob a forma de uma função** invocada a partir dos pontos onde o código original foi colocado anteriormente.

Pode acontecer que o algoritmo que vai implementar seja tão complexo que o seu código comece a crescer de uma forma descontrolada, e de repente note que já não é capaz de navegar através dele tão facilmente.

Pode tentar lidar com a questão comentando extensivamente o código, mas em breve descobrirá que isto piora dramaticamente a sua situação - **demasiados comentários tornam o código maior e mais difícil de ler**. Alguns dizem que **uma função bem escrita deve ser vista inteiramente num só relance**.

Um bom e atento programador **divide o código** (ou mais precisamente: o problema) em peças bem isoladas, e **codifica cada uma delas sob a forma de uma função**.

Isto simplifica consideravelmente o trabalho do programa, porque cada peça de código pode ser codificada separadamente, e testada separadamente. O processo aqui descrito é muitas vezes chamado **decomposição**.

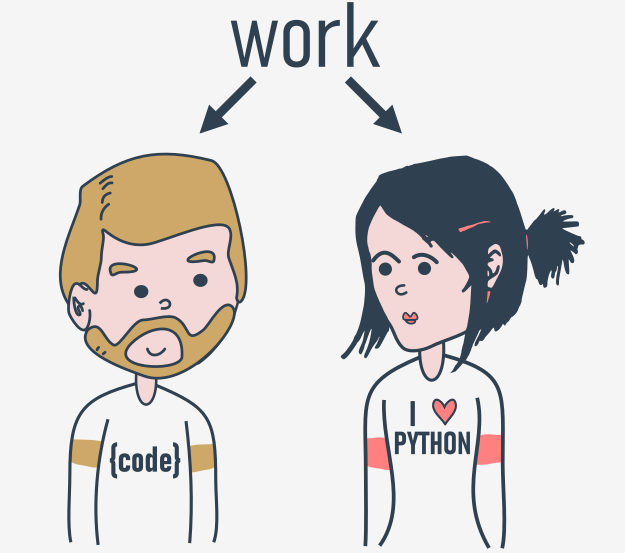


Podemos agora afirmar a segunda condição: **se um pedaço de código se torna tão grande que a sua leitura e compreensão podem causar um problema, considere dividi-lo em problemas separados, mais pequenos, e implemente cada um deles sob a forma de uma função separada**.

Esta decomposição continua até se obter um conjunto de funções curtas, fáceis de compreender e de testar.

# Decomposição

Acontece frequentemente que o problema é tão grande e complexo que não pode ser atribuído a um único programador, e uma **equipa de programadores** tem de trabalhar sobre ele. O problema deve ser dividido entre vários programadores de forma a assegurar a sua cooperação eficiente e sem descontinuidades.



Parece inconcebível que mais do que um programador escreva a mesma peça de código ao mesmo tempo, pelo que o trabalho tem de ser disperso entre todos os membros da equipa.

Este tipo de decomposição tem um propósito diferente do descrito anteriormente - não se trata apenas de **partilhar o trabalho**, mas também de **partilhar a responsabilidade** entre város programadores.

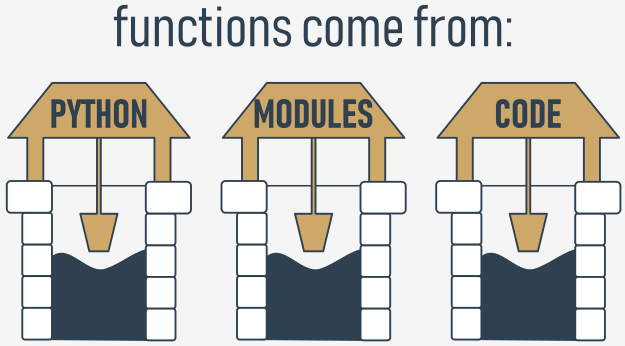
Cada um deles escreve um conjunto claramente definido e descrito de funções, que quando **combinado no módulo** (falaremos sobre isto um pouco mais tarde) dará o produto final.

Isto leva-nos diretamente à terceira condição: se vai dividir o trabalho entre vários programadores, **decomponha o problema para permitir que o produto seja implementado como um conjunto de funções escritas em separado, embaladas em diferentes módulos**.

## De onde vêm as funções?

Em geral, as funções vêm de pelo menos três lugares:

* do próprio Python - inúmeras funções (como print()) são **parte integrante do Python**, e estão sempre disponíveis sem qualquer esforço adicional em nome do programador; chamamos a estas funções **funções integradas**;
* dos **módulos pré-instalados** de Python - muitas funções, muito úteis, mas utilizadas com menos frequência do que as integradas, estão disponíveis em vários módulos instalados juntamente com o Python; a utilização destas funções requer alguns passos adicionais do programador para as tornar totalmente acessíveis (falaremos sobre isto dentro de algum tempo);
* **diretamente do seu código** - pode escrever as suas próprias funções, colocá-las dentro do seu código, e utilizá-las livremente;
* existe uma outra possibilidade, mas está ligada às classes, por isso vamos omiti-la por agora.



# A sua primeira função

Veja o snippet no editor.

print("Enter a value: ")

a = int(input())

print("Enter a value: ")

b = int(input())

print("Enter a value: ")

c = int(input())

É bastante simples, mas queremos apenas que seja um exemplo de **transformação de uma parte repetida de um código numa função**.

As mensagens enviadas para a consola pela função print() são sempre as mesmas. Claro que não há nada de muito mau num código destes, mas tente imaginar o que teria de fazer se o seu chefe lhe pedisse para alterar a mensagem para a tornar mais educada, por exemplo, para a iniciar com a frase "Please,".

Parece que teria de passar algum tempo a alterar todas as ocorrências da mensagem (utilizaria uma área de transferência, claro, mas isso não facilitaria muito a sua vida). É óbvio que provavelmente cometeria alguns erros durante o processo de emenda, e você (e o seu chefe) ficariam um pouco frustrados.

É possível separar uma parte do código que pode ser *repetida*, nomeá-la e torná-la reutilizável? Significaria que **uma mudança feita uma vez num só local seria propagada a todos os locais onde é utilizada**.

É claro que tal código só deve funcionar quando é explicitamente lançado.

Sim, é possível. É exatamente para isto que servem as funções.

## **Como se faz tal função?**

É preciso **defini-la**. A palavra *definir* é significativa aqui.

Este é o aspeto da definição mais simples da função:

def function\_name():

function\_body

* Começa sempre com a **keyword**def ( para *definir*)
* a seguir a def vai o **nome da função** (as regras para nomear funções são exatamente as mesmas que para nomear variáveis)
* após o nome da função, há um lugar para um par de **parêntesis** (não contêm nada aqui, mas isso irá mudar em breve)
* a linha tem de ser terminada com **dois pontos**;
* a linha imediatamente a seguir a def começa o **corpo da função** - um par (pelo menos um) de **instruções necessariamente nested**, que serão executadas sempre que a função for invocada; nota: **a função termina onde termina o nesting**, por isso é preciso ter cuidado.

Estamos prontos para definir a nossa função de **prompting**. Vamos nomeá-la message - aqui está:

def message():

print("Enter a value: ")

A função é extremamente simples, mas totalmente **utilizável**. Demos-lhe o nome message, mas pode rotulá-la a seu gosto. Vamos usá-la.

O nosso código contém agora a definição da função:

def message():

print("Enter a value: ")

print("We start here.")

print("We end here.")

Nota: não utilizamos de todo a função - não há nenhuma **invocação** dentro do código.

Quando a executa, vê o seguinte output:

We start here.

We end here.

**output**

Isto significa que o Python lê as definições da função e lembra-se delas, mas não as lança sem a sua permissão.

Modificámos o código agora - inserimos a **invocação da função** entre as mensagens de início e fim:

def message():

print("Enter a value: ")

print("We start here.")

message()

print("We end here.")

O output parece diferente agora:

We start here.

Enter a value:

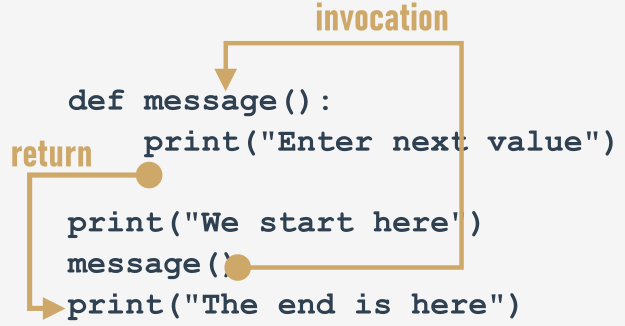
We end here.

**output**

Teste o código, modifique-o, experimente com ele.

**Como funcionam as funções**

Veja a imagem abaixo:



Ela tenta mostrar-lhe todo o processo:

* quando se **invoca** uma função, o Python lembra-se do local onde aconteceu e *salta* para a função invocada;
* o corpo da função é então **executado**;
* chegar ao fim da função força o Python a **regressar** ao local diretamente após o ponto de invocação.

Há dois, muito importantes, senão. Aqui está o primeiro deles:

**Não se deve invocar uma função que não seja conhecida no momento da invocação.**

Lembre-se - o Python lê o seu código de cima para baixo. Não vai olhar em frente para encontrar uma função que se esqueceu de colocar no lugar certo ("certo" significa "antes da invocação").

Inserimos um erro neste código - consegue ver a diferença?

print("We start here.")

message()

print("We end here.")

def message():

print("Enter a value: ")

Movemos a função para o final do código. O Python é capaz de encontrá-la quando a execução atinge a invocação?

Não, não é. A mensagem de erro lerá:

NameError: name 'message' is not defined

**output**

Não tente forçar o Python a procurar por funções que não entregou na altura certa.

O segundo senão soa um pouco mais simples:

**Não se deve ter uma função e uma variável com o mesmo nome**.

O seguinte snippet está errado:

def message():

print("Enter a value: ")

message = 1

Atribuir um valor à mensagem do nome faz com que o Python esqueça o seu papel anterior. A função chamada message torna-se indisponível.

Felizmente, é livre de **misturar o seu código com funções** - não é obrigado a colocar todas as suas funções no topo do seu source file.

Veja o snippet:

print("We start here.")

def message():

print("Enter a value: ")

message()

print("We end here.")

Pode parecer estranho, mas está completamente correto, e funciona como pretendido.

Voltemos ao nosso exemplo principal, e utilizemos a função para o trabalho certo, como aqui:

def message():

print("Enter a value: ")

message()

a = int(input())

message()

b = int(input())

message()

c = int(input())

Modificar a mensagem de solicitação é agora fácil e claro - pode fazê-lo **alterando o código num só lugar** - dentro do corpo da função.

Abra a sandbox e tente fazê-lo você mesmo.

**Key takeaways**

1. Uma **função** é um bloco de código que executa uma tarefa específica quando a função é chamada (invocada). Pode utilizar funções para tornar o seu código reutilizável, mais bem organizado e mais legível. As funções podem ter parâmetros e valores de retorno.

2. Existem pelo menos quatro tipos básicos de funções em Python:

* **funções incorporadas** que são parte integrante do Python (tais como a função print() ). Pode ver uma lista completa de funções Python incorporadas em <https://docs.python.org/3/library/functions.html>.
* as que vêm de **módulos pré-instalados** (aprenderá sobre eles no curso *Python Essentials 2*)
* **funções definidas pelo utilizador** que são escritas por utilizadores para utilizadores - pode escrever as suas próprias funções e utilizá-las livremente no seu código,
* as funções lambda (aprenderá sobre elas no curso *Python Essentials 2*.)

3. Pode definir a sua própria função usando a keyword def e a seguinte sintaxe:

def your\_function(optional parameters):

# the body of the function

Pode definir uma função que não aceita quaisquer argumentos, por exemplo:

def message(): # defining a function

print("Hello") # body of the function

message() # calling the function

Pode definir uma função que também aceita argumentos, tal como a função de um parámetro abaixo:

def hello(name): # defining a function

print("Hello,", name) # body of the function

name = input("Enter your name: ")

hello(name) # calling the function

Na próxima secção, falaremos mais sobre as funções parametrizadas. Não se preocupe.

**Exercício 1**

A função input() é um exemplo de uma:  
  
a) função definida pelo utilizador  
b) função incorporada

Verifique

**Exercício 2**

O que acontece quando se tenta invocar uma função antes de a definir? Exemplo:

hi()

def hi():

print("hi!")

Verifique

**Exercício 3**

O que acontecerá quando executar o código abaixo?

def hi():

print("hi")

hi(5)

Verifique

**Funções parametrizadas**

Todo o potencial da função é revelado quando esta pode ser equipada com uma interface capaz de aceitar dados fornecidos pelo invocador. Tais dados podem modificar o comportamento da função, tornando-a mais flexível e adaptável às condições em mudança.

Um parâmetro é na verdade uma variável, mas há dois fatores importantes que tornam os parâmetros diferentes e especiais:

* **os parâmetros existem apenas dentro das funções em que foram definidos**, e o único lugar onde o parâmetro pode ser definido é um espaço entre um par de parêntesis na declaração def ;
* **atribuir um valor ao parâmetro é feito no momento da invocação da função**, especificando o argumento correspondente.

def function(parameter):

###

Não se esqueça:

* **os parâmetros vivem dentro de funções** (este é o seu ambiente natural)
* **os argumentos existem fora de funções**, e são portadores de valores passados para parâmetros correspondentes.

Existe uma fronteira clara e inequívoca entre estes dois mundos.

Vamos enriquecer a função acima com apenas um parâmetro - vamos usá-lo para mostrar ao utilizador o número de um valor que a função pede.

Temos de reconstruir a declaração def - é assim que se parece agora:

def message(number):

###

A definição especifica que a nossa função opera apenas num parâmetro chamado number. Pode utilizá-lo como uma variável comum, mas **apenas dentro da função** - ele não é visível em qualquer outro lugar.

Vamos agora melhorar o corpo da função:

def message(number):

print("Enter a number:", number)

Fizemos uso do parâmetro. Nota: não atribuímos o parâmetro com qualquer valor. Está correto?

Sim, está.

Um valor para o parâmetro chegará a partir do ambiente da função.

Lembre-se: **especificar um ou mais parâmetros na definição de uma função** também é um requisito, e tem de o cumprir durante a invocação. Deve **fornecer tantos argumentos quantos os parâmetros definidos**.

Não o fazer causará um erro.

Tente executar o código no editor.

def message(number):

print("Enter a number:", number)

message()

Isto é o que verá na consola:

TypeError: message() missing 1 required positional argument: 'number'

**output**

Isto parece melhor, com certeza:

def message(number):

print("Enter a number:", number)

message(1)

Além disso, comporta-se melhor. O código irá produzir o seguinte output:

Enter a number: 1

**output**

Vê como funciona? O valor do argumento utilizado durante a invocação (1) foi passado para a função, definindo o valor inicial do parâmetro chamado number.

Temos de o tornar sensível a uma circunstância importante.

É válido, e possível, ter uma **variável com o mesmo nome que o parâmetro de uma função**.

O snippet ilustra o fenómeno:

def message(number):

print("Enter a number:", number)

number = 1234

message(1)

print(number)

Uma situação como esta ativa um mecanismo chamado **sombreamento**:

* parâmetro x sombreia qualquer variável do mesmo nome, mas...
* ... apenas dentro da função que define o parâmetro.

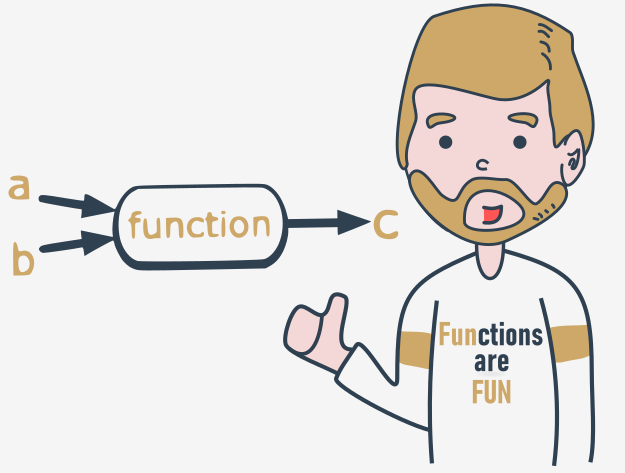
O parâmetro chamado number é uma entidade completamente diferente da variável chamada number.

Isto significa que o snippet acima irá produzir o seguinte output:

Enter a number: 1

1234

Uma função pode ter **tantos parâmetros quantos quiser**, mas quanto mais parâmetros tiver, mais difícil é memorizar os seus papéis e propósitos.



Vamos modificar a função - tem **dois parâmetros** agora:

def message(what, number):

print("Enter", what, "number", number)

Isto também significa que invocar a função irá requerer **dois argumentos**.

O primeiro parâmetro novo destina-se a levar o nome do valor desejado.

Aqui está:

def message(what, number):

print("Enter", what, "number", number)

message("telephone", 11)

message("price", 5)

message("number", "number")

Esta é o output que está prestes a ver:

Enter telephone number 11

Enter price number 5

Enter number number number

**output**

Execute o código, modifique-o, adicione mais parâmetros e veja como isso afeta o output.

# Passagem de parâmetros posicionais

Uma técnica que atribui o iésimo (primeiro, segundo, e assim por diante) argumento ao iésimo (primeiro, segundo, e assim por diante) parâmetro de função é chamada **passagem de parâmetro posicional**, enquanto os argumentos passados desta forma são denominados **argumentos posicionais**.

Já o usou, mas o Python pode oferecer muito mais. Vamos agora falar-lhe sobre isso.

def my\_function(a, b, c):

print(a, b, c)

my\_function(1, 2, 3)

Nota: a passagem de parâmetros posicionais é intuitivamente utilizada pelas pessoas em muitas ocasiões sociais. Por exemplo, pode ser geralmente aceite que quando nos apresentamos mencionemos o(s) nosso(s) nome(s) próprio(s) antes do nosso apelido, por exemplo, "O meu nome é Miguel Matos".

A propósito, os húngaros fazem-no pela ordem inversa.

Vamos implementar esse costume social em Python. A seguinte função será responsável pela introdução de alguém:

def introduction(first\_name, last\_name):

print("Hello, my name is", first\_name, last\_name)

introduction("Luke", "Skywalker")

introduction("Jesse", "Quick")

introduction("Clark", "Kent")

Consegue adivinhar o output? Execute o código e descubra se estava certo.

Agora imagine que a mesma função está a ser utilizada na Hungria. Nesse caso, o código ficaria assim:

def introduction(first\_name, last\_name):

print("Hello, my name is", first\_name, last\_name)

introduction("Skywalker", "Luke")

introduction("Quick", "Jesse")

introduction("Kent", "Clark")

O output terá um aspeto diferente. Consegue adivinhá-lo?

Execute o código para ver se também estava certo aqui. Está surpreendido?

Pode tornar a função mais independente da cultura?

# Passagem de argumentos de keyword

O Python oferece outra convenção para passar argumentos, onde **o significado do argumento é ditado pelo seu nome**, não pela sua posição - chama-se **keyword argument passing**.

Veja o snippet:

def introduction(first\_name, last\_name):

print("Hello, my name is", first\_name, last\_name)

introduction(first\_name = "James", last\_name = "Bond")

introduction(last\_name = "Skywalker", first\_name = "Luke")

O conceito é claro - os valores passados para os parâmetros são precedidos pelos nomes dos parâmetros alvo, seguidos do sinal = .

A posição não importa aqui - o valor de cada argumento conhece o seu destino com base no nome utilizado.

Deverá ser capaz de prever o output. Execute o código para verificar se estava certo.

É claro que **não se deve usar um nome de parâmetro inexistente**.

O snippet seguinte causará um erro de runtime:

def introduction(first\_name, last\_name):

print("Hello, my name is", first\_name, last\_name)

introduction(surname="Skywalker", first\_name="Luke")

Isto é o que Python lhe dirá:

TypeError: introduction() got an unexpected keyword argument 'surname'

**output**

Experimente você mesmo.

**Mistura de argumentos posicionais e de keywords**

Pode misturar ambos os tipos se quiser - há apenas uma regra inquebrável: tem de colocar **argumentos posicionais antes de argumentos de keyword**.

Se pensar por um momento, certamente adivinhará porquê.

Para lhe mostrar como funciona, utilizaremos a seguinte função simples de três parâmetros:

def adding(a, b, c):

print(a, "+", b, "+", c, "=", a + b + c)

O seu objetivo é avaliar e apresentar a soma de todos os seus argumentos.

A função, quando invocada da seguinte maneira:

adding(1, 2, 3)

terá como output:

1 + 2 + 3 = 6

**output**

Foi - como se pode suspeitar - um puro exemplo de **passagem de argumento posicional**.

É claro que se pode substituir tal invocação por uma variante de keyword pura, como esta:

adding(c = 1, a = 2, b = 3)

O nosso programa fará output de uma linha como esta:

2 + 3 + 1 = 6

**output**

Observe a ordem dos valores.

Vamos tentar misturar ambos os estilos agora.

Veja a invocação da função abaixo:

adding(3, c = 1, b = 2)

Vamos analisá-la:

* o argumento (3) para o parâmetro a é passado usando a maneira posicional;
* os argumentos para c e b são especificados como keywords.

Isto é o que verá na consola:

3 + 2 + 1 = 6

**output**

Tenha cuidado, e atenção aos erros. Se tentar passar mais do que um valor para um argumento, tudo o que obterá será um erro de runtime.

Olhe para a invocação em baixo - parece que tentámos fixar a duas vezes:

adding(3, a = 1, b = 2)

A resposta do Python:

TypeError: adding() got multiple values for argument 'a'

**output**

Olhe para o snippet abaixo. Um código como este está totalmente correto, mas não faz muito sentido:

adding(4, 3, c = 2)

Tudo está correto, mas deixar apenas uma keyword parece um pouco estranho - o que acha?

# Funções parametrizadas - mais detalhes

Acontece por vezes que os valores de um determinado parâmetro são utilizados com mais frequência do que outros. Tais argumentos podem ter os seus **valores padrão (predefinidos)** tomados em consideração quando os seus argumentos correspondentes tiverem sido omitidos.

Dizem que o apelido inglês mais popular é *Smith*. Vamos tentar ter isto em conta.

O valor padrão do parâmetro é definido usando uma sintaxe clara e pictórica:

def introduction(first\_name, last\_name="Smith"):

print("Hello, my name is", first\_name, last\_name)

Só tem de estender o nome do parâmetro com o sinal = , seguido pelo valor padrão.

Vamos invocar a função como de costume:

introduction("James", "Doe")

Consegue adivinhar o output do programa? Execute-o e verifique se estava certo.

Então? Tudo parece igual, mas quando se invoca a função de uma forma que à primeira vista parece um pouco suspeita, como esta:

introduction("Henry")

ou esta:

introduction(first\_name="William")

não haverá nenhum erro, ambas as invocações serão bem sucedidas, e a consola mostrará o seguinte output:

Hello, my name is Henry Smith

Hello, my name is William Smith

**output**

Teste-o.

Pode ir mais além se for útil. Ambos os parâmetros têm agora os seus valores padrão, veja o código em baixo:

def introduction(first\_name="John", last\_name="Smith"):

print("Hello, my name is", first\_name, last\_name)

Isto faz com que a seguinte invocação seja absolutamente válida:

introduction()

E este é o output esperado:

Hello, my name is John Smith

**output**

Se utilizar um argumento de keyword, o restante tomará o valor padrão:

introduction(last\_name="Hopkins")

O output é:

Hello, my name is John Hopkins

**output**

Teste-o.

Parabéns - acabou de aprender as formas básicas de comunicar com funções.

**Key takeaways**

1. É possível passar informações para funções utilizando parâmetros. As suas funções podem ter todos os parâmetros que precisar.

Um exemplo de uma função de um parâmetro:

def hi(name):

print("Hi,", name)

hi("Greg")

Um exemplo de uma função de dois parâmetros:

def hi\_all(name\_1, name\_2):

print("Hi,", name\_2)

print("Hi,", name\_1)

hi\_all("Sebastian", "Konrad")

Um exemplo de uma função de três parâmetros:

def address(street, city, postal\_code):

print("Your address is:", street, "St.,", city, postal\_code)

s = input("Street: ")

p\_c = input("Postal Code: ")

c = input("City: ")

address(s, c, p\_c)

2. Pode passar argumentos a uma função utilizando as seguintes técnicas:

* **passagem de argumento posicional** onde a ordem de passagem dos argumentos importa (Ex. 1),
* **passagem de argumento de keyword (nomeada)** onde a ordem dos argumentos passados não importa (Ex. 2),
* uma mistura de passagem de argumentos posicionais e de keyword (Ex. 3).

Ex. 1

def subtra(a, b):

print(a - b)

subtra(5, 2) # outputs: 3

subtra(2, 5) # outputs: -3

Ex. 2

def subtra(a, b):

print(a - b)

subtra(a=5, b=2) # outputs: 3

subtra(b=2, a=5) # outputs: 3

Ex. 3

def subtra(a, b):

print(a - b)

subtra(5, b=2) # outputs: 3

subtra(5, 2) # outputs: 3

É importante lembrar que **os argumentos posicionais não devem seguir os argumentos de keyword**. É por isso que se tentar executar o seguinte snippet:

def subtra(a, b):

print(a - b)

subtra(5, b=2) # outputs: 3

subtra(a=5, 2) # Syntax Error

O Python não o deixará fazê-lo através da sinalização de um SyntaxError.

3. Pode usar a técnica de passagem de argumentos de keyword para **pré-definir** um valor para um determinado argumento:

def name(first\_name, last\_name="Smith"):

print(first\_name, last\_name)

name("Andy") # outputs: Andy Smith

name("Betty", "Johnson") # outputs: Betty Johnson (the keyword argument replaced by "Johnson")

**Exercício 1**

Qual é o output do seguinte snippet?

def intro(a="James Bond", b="Bond"):

print("My name is", b + ".", a + ".")

intro()

Verifique

**Exercício 2**

Qual é o output do seguinte snippet?

def intro(a="James Bond", b="Bond"):

print("My name is", b + ".", a + ".")

intro(b="Sean Connery")

Verifique

**Exercício 3**

Qual é o output do seguinte snippet?

def intro(a, b="Bond"):

print("My name is", b + ".", a + ".")

intro("Susan")

Verifique

**Exercício 4**

Qual é o output do seguinte snippet?

def add\_numbers(a, b=2, c):

print(a + b + c)

add\_numbers(a=1, c=3)

# Efeitos e resultados: a instrução return .

Todas as funções anteriormente apresentadas têm algum tipo de efeito - produzem algum texto e enviam-no para a consola.

Claro que as funções - tal como os seus irmãos matemáticos - podem ter resultados.

Para fazer com que as **funções devolvam um valor** (mas não apenas para este fim) usa-se a instrução return .

Esta palavra dá-lhe uma imagem completa das suas capacidades. Nota: é uma **keyword** Python.

A instrução return tem **duas variantes diferentes** - vamos considerá-las separadamente.

## return sem uma expressão

A primeira consiste na própria keyword, sem nada a seguir.

Quando utilizada dentro de uma função, provoca a **terminação imediata da execução da função, e uma devolução (em inglês, return) imediata (daí o nome) ao ponto de invocação**.

Nota: se uma função não se destina a produzir um resultado, **utilizar a instrução**return**não é obrigatório** - será executada implicitamente no final da função.

De qualquer modo, pode utilizá-la para **terminar as atividades de uma função a pedido**, antes de o controlo chegar à última linha da função.

Consideremos a seguinte função:

def happy\_new\_year(wishes = True):

print("Three...")

print("Two...")

print("One...")

if not wishes:

return

print("Happy New Year!")

Quando invocada sem argumentos:

happy\_new\_year()

A função causa um pequeno ruído - o output terá este aspeto:

Three...

Two...

One...

Happy New Year!

**output**

Fornecer False como um argumento:

happy\_new\_year(False)

irá modificar o comportamento da função - a instrução return causará a sua terminação imediatamente antes dos wishes - este é o resultado atualizado:

Three...

Two...

One...

**output**

## return com uma expressão

A segunda variante return é **estendida com uma expressão**:

def function():

return expression

Há duas consequências da sua utilização:

* provoca a **terminação imediata da execução da função** (nada de novo em comparação com a primeira variante)
* além disso, a função **avaliará o valor da expressão e devolvê-la-á (daí o nome, mais uma vez) como resultado da função**.

Sim, nós já sabemos - este exemplo não é realmente sofisticado:

def boring\_function():

return 123

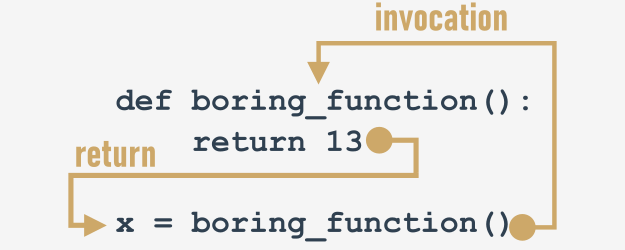
x = boring\_function()

print("The boring\_function has returned its result. It's:", x)

O snippet grava o seguinte texto na consola:

The boring\_function has returned its result. It's: 123

Vamos investigá-lo durante algum tempo.

Analise a figura abaixo: 

A instrução return , enriquecida com a expressão (a expressão é muito simples aqui), "transporta" o valor da expressão para o local onde a função foi invocada.

O resultado pode ser livremente utilizado aqui, por exemplo, para ser atribuído a uma variável.

Também pode ser completamente ignorado e perdido sem deixar rasto.

Note, não estamos a ser muito educados aqui - a função devolve um valor, e ignoramo-lo (não o utilizamos de forma alguma):

def boring\_function():

print("'Boredom Mode' ON.")

return 123

print("This lesson is interesting!")

boring\_function()

print("This lesson is boring...")

O programa produz o seguinte output:

This lesson is interesting!

'Boredom Mode' ON.

This lesson is boring...

**output**

É punível? De modo algum.

A única desvantagem é que o resultado foi irremediavelmente perdido.

Não se esqueça:

* é-lhe sempre **permitido ignorar o resultado da função**, e ficar satisfeito com o efeito da função (se a função tiver algum)
* se uma função se destina a devolver um resultado útil, deve conter a segunda variante da instrução return .

Espere um minuto - isto significa que também há resultados inúteis? Sim - de certa forma.

**Algumas palavras sobre None**

Deixe-nos apresentar-lhe um valor muito curioso (para sermos honestos, um valor nulo), chamado None.

Os seus dados não representam nenhum valor razoável - na verdade, não é um valor de todo; portanto, **não deve tomar parte em nenhuma expressão**.

Por exemplo, um snippet como este:

print(None + 2)

causará um erro de runtime, descrito pela seguinte mensagem de diagnóstico:

TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'

**output**

Nota: None é uma **keyword**.

Existem apenas dois tipos de circunstâncias onde None pode ser utilizado em segurança:

* quando o **atribui a uma variável** (ou o devolve como o **resultado de uma função**)
* quando o **compara com uma variável** para diagnosticar o seu estado interno.

Tal como aqui:

value = None

if value is None:

print("Sorry, you don't carry any value")

Não se esqueça disto: se uma função não devolver um determinado valor usando uma return cláusula de expressão, supõe-se que ela **devolve implicitamente**None.

Vamos testá-lo.

# Algumas palavras sobre None: continuação

Veja o código no editor.

É óbvio que a função strangeFunction devolve True quando o seu argumento é par.

O que devolve de outra forma?

Podemos utilizar o seguinte código para o verificar:

print(strange\_function(2))

print(strange\_function(1))

Isto é o que vemos na consola:

True

None

**output**

Não fique surpreendido da próxima vez que vir None como resultado de uma função - pode ser o sintoma de um erro subtil dentro da função.

# Efeitos e resultados: listas e funções

Há duas questões adicionais que devem ser respondidas aqui.

A primeira é: **pode uma lista ser enviada para uma função como argumento?**

Claro que pode! Qualquer entidade reconhecível pelo Python pode desempenhar o papel de um argumento de função, embora tenha de ter a certeza de que a função é capaz de lidar com ela.

Assim, se passar uma lista a uma função, a função tem de lidar com ela como uma lista.

Uma função como esta aqui:

def list\_sum(lst):

s = 0

for elem in lst:

s += elem

return s

e invocada assim:

print(list\_sum([5, 4, 3]))

irá devolver 12 como resultado, mas deve esperar problemas se a invocar desta forma arriscada:

print(list\_sum(5))

A resposta do Python será inequívoca:

TypeError: 'int' object is not iterable

**output**

Isto é causado pelo facto de que **um único valor inteiro não deve ser iterado pelo loop**for.

# Efeitos e resultados: listas e funções - continuação

A segunda questão é: **pode uma lista ser um resultado de uma função?**

Sim, claro! Qualquer entidade reconhecível por Python pode ser um resultado de uma função.

Veja o código no editor. O output do programa será assim:

[4, 3, 2, 1, 0]

**output**

Agora pode escrever funções com e sem resultados.

Vamos mergulhar um pouco mais fundo nas questões ligadas às variáveis das funções. Isto é essencial para a criação de funções eficazes e seguras.

## LAB - Um ano bissexto.

## Objetivos

Familiarizar o aluno a:

* projetar e escrever funções parametrizadas;
* utilizar a return declaração;
* testar as funções.

## Cenário

A sua tarefa é escrever e testar uma função que leva um argumento (um ano) e devolve True se o ano for um ano *bissexto*, ou False caso contrário.

A seed da função já está semeada no código esqueleto no editor.

Nota: preparámos também um pequeno código de teste, que pode utilizar para testar a sua função.

O código utiliza duas listas - uma com os dados do teste, e a outra com os resultados esperados. O código dir-lhe-á se algum dos seus resultados é inválido.

def is\_year\_leap(year):

#

# put your code here

#

test\_data = [1900, 2000, 2016, 1987]

test\_results = [False, True, True, False]

for i in range(len(test\_data)):

yr = test\_data[i]

print(yr,"->",end="")

result = is\_year\_leap(yr)

if result == test\_results[i]:

print("OK")

else:

print("Failed")

RESPOSTA

def is\_year\_leap(year):

    if year % 4 == 0 :

        if year % 100 == 0 and year % 400 != 0: return False

        else: return True

    else: return False

test\_data = [1900, 2000, 2016, 1987]

test\_results = [False, True, True, False]

for i in range(len(test\_data)):

    yr = test\_data[i]

    print(yr,"->",end="")

    result = is\_year\_leap(yr)

    if result == test\_results[i]:

        print("OK")

    else:

        print("Failed")

## LAB – Quantos dias

## Objetivos

Familiarizar o aluno a:

* projetar e escrever funções parametrizadas;
* utilizar a return declaração;
* utilizar as próprias funções do aluno.

## Cenário

A sua tarefa é escrever e testar uma função que toma dois argumentos (um ano e um mês) e devolve o número de dias para o par mês/ano dado (enquanto apenas fevereiro é sensível ao valor year , a sua função deve ser universal).

A parte inicial da função está pronta. Agora, convença a função a devolver None se os seus argumentos não fizerem sentido.

É claro que pode (e deve) usar a função previamente escrita e testada (LAB 4.3.1.6). Pode ser muito útil. Encorajamo-lo a utilizar uma lista preenchida com a duração dos meses. Pode criá-la dentro da função - este truque irá encurtar significativamente o código.

Preparámos um código de teste. Expanda-o para incluir mais casos de teste.

def is\_year\_leap(year):

#

# Your code from LAB 4.3.1.6.

#

def days\_in\_month(year, month):

#

# Write your new code here.

#

test\_years = [1900, 2000, 2016, 1987]

test\_months = [2, 2, 1, 11]

test\_results = [28, 29, 31, 30]

for i in range(len(test\_years)):

yr = test\_years[i]

mo = test\_months[i]

print(yr, mo, "->", end="")

result = days\_in\_month(yr, mo)

if result == test\_results[i]:

print("OK")

else:

print("Failed")

RESPOSTA

def is\_year\_leap(year):

    if year % 4 == 0 :

        if year % 100 == 0 and year % 400 != 0: return False

        else: return True

    else: return False

def days\_in\_month(year, month):

    months = [31,28,31,30,31,30,31,31,30,31,30,31]

    bissexto = is\_year\_leap(year)

    if month == 2:

        if bissexto == True: x = 29

        else: x = 28

        return x

    else:

        return months[month-1]

test\_years = [1900, 2000, 2016, 1987]

test\_months = [2, 2, 1, 11]

test\_results = [28, 29, 31, 30]

for i in range(len(test\_years)):

    yr = test\_years[i]

    mo = test\_months[i]

    print(yr, mo, "->", end="")

    result = days\_in\_month(yr, mo)

    if result == test\_results[i]:

        print("OK")

    else:

        print("Failed")

## LAB – Dia do ano

## Objetivos

Familiarizar o aluno a:

* projetar e escrever funções parametrizadas;
* utilizar a return declaração;
* construir um conjunto de funções de utilidade;
* utilizar as próprias funções do aluno.

## Cenário

A sua tarefa consiste em escrever e testar uma função que toma três argumentos (um ano, um mês e um dia do mês) e devolve o dia correspondente do ano, ou devolve None se algum dos argumentos for inválido.

Use as funções previamente escritas e testadas. Adicione alguns casos de teste ao código. Este teste é apenas um começo.

def is\_year\_leap(year):

    if year % 4 == 0 :

        if year % 100 == 0 and year % 400 != 0: return False

        else: return True

    else: return False

def days\_in\_month(year, month):

    months = [31,28,31,30,31,30,31,31,30,31,30,31]

    bissexto = is\_year\_leap(year)

    if month == 2:

        if bissexto == True: x = 29

        else: x = 28

        return x

    else:

        return months[month-1]

def day\_of\_year(year, month, day):

    days\_of\_months = [31,28,31,30,31,30,31,31,30,31,30,31]

    if day <= days\_of\_months[month-1]:

        for i in range(month-1):

            day += days\_in\_month(year, month + 1)

        return day

print(day\_of\_year(2000, 2, 28))

## LAB – Números primos

## Objetivos

* familiarizar o aluno com noções e algoritmos clássicos;
* melhorar as habilidades do aluno na definição e utilização de funções.

## Cenário

*Um número natural é****primo****se for maior que 1 e não tiver divisores além de 1 e dele próprio.*

Complicado? De modo algum. Por exemplo, 8 não é um número primo, uma vez que se pode dividi-lo por 2 e 4 (não podemos utilizar divisores iguais a 1 e 8, uma vez que a definição o proíbe).

Por outro lado, 7 é um número primo, uma vez que não conseguimos encontrar quaisquer divisores legais para ele.

A sua tarefa é escrever uma função para verificar se um número é primo ou não.

A função:

* é chamada is\_prime;
* toma um argumento (o valor a verificar)
* devolve True se o argumento for um número primo, e False caso contrário.

Dica: tente dividir o argumento por todos os valores subsequentes (começando por 2) e verifique os restos - se for zero, o seu número não pode ser um primo; pense cuidadosamente sobre quando deve parar o processo.

Se precisar de conhecer a raiz quadrada de qualquer valor, pode utilizar o operador \*\* . Lembre-se: a raiz quadrada de x é o mesmo que x0.5

Complete o código no editor.

Execute o seu código e verifique se o seu output é o mesmo que o nosso.

RESPOSTA

def is\_prime(num):

    #counter = 0

    for counter in range(num):

        counter += 1

        if (num % counter == 0) and (counter != 1) and (num != counter): return False

    return True

for i in range(1, 20):

    if is\_prime(i + 1):

            print(i + 1, end=" ")

print()

## LAB – Conversão de consumo de combustível

## Objetivos

* melhorar as competências do aluno na definição, utilização e teste de funções.

## Cenário

O consumo de combustível de um carro pode ser expresso de várias maneiras. Por exemplo, na Europa, é mostrado como a quantidade de combustível consumido por 100 quilómetros.

Nos EUA, é mostrado como o número de milhas percorridas por um carro utilizando um galão de combustível.

A sua tarefa é escrever um par de funções convertendo l/100 km em mpg, e vice-versa.

As funções:

* são nomeadas liters\_100km\_to\_miles\_gallon e miles\_gallon\_to\_liters\_100km respetivamente;
* tome um argumento (o valor correspondente aos seus nomes)

Complete o código no editor.

Execute o seu código e verifique se o seu output é o mesmo que o nosso.

Aqui estão algumas informações para o ajudar:

* 1 milha americana = 1609,344 metros;
* 1 galão americano = 3,785411784 litros.

60.31143162393162

31.36194444444444

23.52145833333333

3.9007393587617467

7.490910297239916

10.009131205673757

RESPOSTA

kmPmiles = 1.609344

milesPgallon = 3.785411784

def liters\_100km\_to\_miles\_gallon(liters):

    result = ((100 / liters) / kmPmiles) \* milesPgallon

    return result

def miles\_gallon\_to\_liters\_100km(miles):

    result = milesPgallon / (kmPmiles \* miles) \* 100

    return result

print(liters\_100km\_to\_miles\_gallon(3.9))

print(liters\_100km\_to\_miles\_gallon(7.5))

print(liters\_100km\_to\_miles\_gallon(10.))

print(miles\_gallon\_to\_liters\_100km(60.3))

print(miles\_gallon\_to\_liters\_100km(31.4))

print(miles\_gallon\_to\_liters\_100km(23.5))

# Key takeaways

1. Pode utilizar a keyword return para dizer a uma função para devolver algum valor. A declaração return sai da função, por exemplo:

def multiply(a, b):

return a \* b

print(multiply(3, 4)) # outputs: 12

def multiply(a, b):

return

print(multiply(3, 4)) # outputs: None

2. O resultado de uma função pode ser facilmente atribuído a uma variável, por exemplo:

def wishes():

return "Happy Birthday!"

w = wishes()

print(w) # outputs: Happy Birthday!

Veja a diferença no output dos dois exemplos a seguir:

# Example 1

def wishes():

print("My Wishes")

return "Happy Birthday"

wishes() # outputs: My Wishes

# Example 2

def wishes():

print("My Wishes")

return "Happy Birthday"

print(wishes())

# outputs: My Wishes

# Happy Birthday

3. Pode utilizar uma lista como argumento de uma função, por exemplo:

def hi\_everybody(my\_list):

for name in my\_list:

print("Hi,", name)

hi\_everybody(["Adam", "John", "Lucy"])

4. Uma lista também pode ser o resultado de uma função, por exemplo:

def create\_list(n):

my\_list = []

for i in range(n):

my\_list.append(i)

return my\_list

print(create\_list(5))

**Exercício 1**

Qual é o output do seguinte snippet?

def hi():

return

print("Hi!")

hi()

Verifique

**Exercício 2**

Qual é o output do seguinte snippet?

def is\_int(data):

if type(data) == int:

return True

elif type(data) == float:

return False

print(is\_int(5))

print(is\_int(5.0))

print(is\_int("5"))

Verifique

**Exercício 3**

Qual é o output do seguinte snippet?

def even\_num\_lst(ran):

lst = []

for num in range(ran):

if num % 2 == 0:

lst.append(num)

return lst

print(even\_num\_lst(11))

Verifique

**Exercício 4**

Qual é o output do seguinte snippet?

def list\_updater(lst):

upd\_list = []

for elem in lst:

elem \*\*= 2

upd\_list.append(elem)

return upd\_list

foo = [1, 2, 3, 4, 5]

print(list\_updater(foo))

# Funções e scopes

Vamos começar com uma definição:

O **scope de um nome** (por exemplo, um nome de variável) é a parte de um código onde o nome é devidamente reconhecido.

Por exemplo, o scope do parâmetro de uma função é a própria função. O parâmetro é inacessível fora da função.

Vamos verificar. Veja o código no editor. O que acontecerá quando o executar?

def scope\_test():

x = 123

scope\_test()

print(x)

O programa falhará quando executado. A mensagem de erro lerá:

NameError: name 'x' is not defined

**output**

Isto é de se esperar.

Vamos realizar consigo algumas experiências para lhe mostrar como o Python constrói os scopes, e como pode usar os seus hábitos em seu benefício.

Vamos começar por verificar se uma variável criada fora de qualquer função é ou não visível dentro das funções. Por outras palavras, o nome de uma variável propaga-se no corpo de uma função?

def my\_function():

print("Do I know that variable?", var)

var = 1

my\_function()

print(var)

Veja o código no editor. A nossa cobaia está lá.

O resultado do teste é positivo - o código faz os outputs:

Do I know that variable? 1

1

**output**

A resposta é: **uma variável existente fora de uma função tem um scope dentro dos corpos das funções**.

Esta regra tem uma exceção muito importante. Vamos tentar encontrá-la.

Vamos fazer uma pequena alteração no código:

def my\_function():

var = 2

print("Do I know that variable?", var)

var = 1

my\_function()

print(var)

O resultado também mudou - o código produz um output ligeiramente diferente agora:

Do I know that variable? 2

1

**output**

O que aconteceu?

* a variável var criada dentro da função não é a mesmo que quando definida fora dela - parece que existem duas variáveis diferentes com o mesmo nome;
* além disso, a variável da função sombreia a variável proveniente do mundo exterior.

Podemos tornar a regra anterior mais precisa e adequada:

**Uma variável existente fora de uma função tem um scope dentro dos corpos das funções, excluindo aqueles que definem uma variável com o mesmo nome.**

Isto também significa que o **scope de uma variável existente fora de uma função só é suportado quando se obtém o seu valor** (reading). Atribuir um valor força a criação da própria variável da função.

Certifique-se de que entende bem isto e realize as suas próprias experiências.

# a keyword global .

Esperamos que agora tenha chegado à seguinte pergunta: isto significa que uma função não é capaz de modificar uma variável definida fora dela? Isso criaria muito desconforto.

Felizmente, a resposta é *não*.

Existe um método especial Python que pode **alargar o scope de uma variável de forma a incluir os corpos das funções** (mesmo que se pretenda não só ler os valores, mas também modificá-los).

Tal efeito é causado por uma keyword chamada global:

global name

global name1, name2, ...

Utilizar esta keyword dentro de uma função com o nome (ou nomes separados por vírgulas) de uma variável, força o Python a abster-se de criar uma nova variável dentro da função - a que é acessível a partir do exterior será utilizada em vez disso.

Por outras palavras, este nome torna-se global (tem um **scope global**, e não importa se é o assunto de leitura ou atribuição).

Veja o código no editor.

def my\_function():

global var

var = 2

print("Do I know that variable?", var)

var = 1

my\_function()

print(var)

Adicionámos global à função.

O código agora faz o output:

Do I know that variable? 2

2

**output**

Isto deve ser prova suficiente para mostrar que a keyword global faz o que promete.

**Como a função interage com os seus argumentos**

Vamos agora descobrir como a função interage com os seus argumentos.

O código no editor deve ensinar-lhe algo. Como se pode ver, a função altera o valor do seu parâmetro. A alteração afeta o argumento?

Execute o programa e verifique.

def my\_function(n):

print("I got", n)

n += 1

print("I have", n)

var = 1

my\_function(var)

print(var)

O output do código é:

I got 1

I have 2

1

**output**

A conclusão é óbvia - **a alteração do valor do parâmetro não se propaga fora da função** (em qualquer caso, não quando a variável é uma escalar, como no exemplo).

Isto também significa que uma função recebe o **valor do argumento**, não o argumento em si. Isto é verdade para escalares.

Vale a pena verificar como funciona com listas (lembra-se das peculiaridades de atribuir slices de listas versus atribuir listas como um todo?).

O exemplo a seguir irá lançar alguma luz sobre a questão:

def my\_function(my\_list\_1):

print("Print #1:", my\_list\_1)

print("Print #2:", my\_list\_2)

my\_list\_1 = [0, 1]

print("Print #3:", my\_list\_1)

print("Print #4:", my\_list\_2)

my\_list\_2 = [2, 3]

my\_function(my\_list\_2)

print("Print #5:", my\_list\_2)

O output do código é:

Print #1: [2, 3]

Print #2: [2, 3]

Print #3: [0, 1]

Print #4: [2, 3]

Print #5: [2, 3]

**output**

Parece que a regra anterior ainda funciona.

Finalmente, pode ver a diferença no exemplo abaixo:

def my\_function(my\_list\_1):

print("Print #1:", my\_list\_1)

print("Print #2:", my\_list\_2)

del my\_list\_1[0] # Pay attention to this line.

print("Print #3:", my\_list\_1)

print("Print #4:", my\_list\_2)

my\_list\_2 = [2, 3]

my\_function(my\_list\_2)

print("Print #5:", my\_list\_2)

Não alteramos o valor do parâmetro my\_list\_1 (we already know it will not affect the argument), but instead modify the list identified by it.

O output pode ser surpreendente. Execute o código e verifique:

Print #1: [2, 3]

Print #2: [2, 3]

Print #3: [3]

Print #4: [3]

Print #5: [3]

**output**

Consegue explicá-lo?

Vamos tentar:

* se o argumento for uma lista, então alterar o valor do parâmetro correspondente não afeta a lista (lembre-se: as variáveis que contêm listas são armazenadas de uma forma diferente dos escalares),
* mas se alterar uma lista identificada pelo parâmetro (nota: a lista, não o parâmetro!), a lista irá refletir a alteração.

É tempo de escrever algumas funções de exemplo. Fá-lo-á na próxima secção.

# Key takeaways

1. Uma variável que existe fora de uma função tem um scope dentro do corpo da função (Exemplo 1), a menos que a função defina uma variável com o mesmo nome (Exemplo 2, e Exemplo 3), por exemplo:

Exemplo 1:

var = 2

def mult\_by\_var(x):

return x \* var

print(mult\_by\_var(7)) # outputs: 14

Exemplo 2:

def mult(x):

var = 5

return x \* var

print(mult(7)) # outputs: 35

Exemplo 3:

def mult(x):

var = 7

return x \* var

var = 3

print(mult(7)) # outputs: 49

2. Uma variável que existe dentro de uma função tem um scope dentro do corpo da função (Exemplo 4), por exemplo:

Exemplo 4:

def adding(x):

var = 7

return x + var

print(adding(4)) # outputs: 11

print(var) # NameError

3. Pode utilizar a keyword global seguida por um nome de variável para tornar o scope da variável global, por exemplo

var = 2

print(var) # outputs: 2

def return\_var():

global var

var = 5

return var

print(return\_var()) # outputs: 5

print(var) # outputs: 5

**Exercício 1**

O que acontecerá quando tentar executar o seguinte código?

def message():

alt = 1

print("Hello, World!")

print(alt)

Verifique

**Exercício 2**

Qual é o output do seguinte snippet?

a = 1

def fun():

a = 2

print(a)

fun()

print(a)

Verifique

**Exercício 3**

Qual é o output do seguinte snippet?

a = 1

def fun():

global a

a = 2

print(a)

fun()

a = 3

print(a)

Verifique

**Exercício 4**

Qual é o output do seguinte snippet?

a = 1

def fun():

global a

a = 2

print(a)

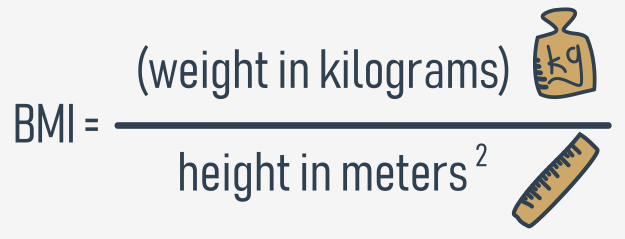
a = 3

fun()

print(a)

**Algumas funções simples: avaliar o IMC**

Vamos começar com uma função para avaliar o Índice de Massa Corporal (IMC).



Como pode ver, a fórmula obtém dois valores:

* peso (originalmente em quilogramas)
* altura (originalmente em metros)

Parece que esta nova função terá **dois parâmetros**. O seu nome será bmi, mas se preferir qualquer outro nome, use-o.

Vamos codificar a função:

def bmi(weight, height):

return weight / height \*\* 2

print(bmi(52.5, 1.65))

O resultado produzido pela invocação da amostra tem o seguinte aspeto:

19.283746556473833

**output**

A função cumpre as nossas expetativas, mas é um pouco simples - assume que os valores de ambos os parâmetros são sempre significativos. Vale definitivamente a pena verificar se são dignos de confiança.

Vamos verificar os dois e devolver None se algum deles parecer suspeito.

# avaliação do IMC (em inglês, BMI) e conversão de unidades imperiais em unidades métricas

Veja o código no editor. Há duas coisas a que temos de prestar atenção.

def bmi(weight, height):

if height < 1.0 or height > 2.5 or \

weight < 20 or weight > 200:

return None

return weight / height \*\* 2

print(bmi(352.5, 1.65))

Primeiro, a invocação de teste assegura que a **proteção** funciona corretamente - o output é:

None

**output**

Em segundo lugar, dê uma vista de olhos na forma como o símbolo da **barra invertida** (\) é usado. Se o utilizar em código Python e terminar uma linha com ele, dirá ao Python para continuar a linha de código na próxima linha de código.

Pode ser particularmente útil quando se tem de lidar com longas linhas de código e se gostaria de melhorar a legibilidade do código.

Está bem, mas há algo que omitimos com demasiada facilidade - as medidas imperiais. Esta função não é muito útil para pessoas acostumadas a libras, pés e polegadas.

O que pode ser feito por elas?

Podemos escrever duas funções simples para **converter unidades imperiais em unidades métricas**. Vamos começar com as libras.

É um facto bem conhecido que 1 lb = 0.45359237 kg. Utilizaremos isto na nossa nova função.

Esta é a nossa função auxiliar, chamada lb\_to\_kg:

def lb\_to\_kg(lb):

return lb \* 0.45359237

print(lb\_to\_kg(1))

O resultado da invocação do teste parece bom:

0.45359237

**output**

E agora é altura para os pés e polegadas: 1 ft = 0.3048 m, e 1 in = 2.54 cm = 0.0254 m.

A função que escrevemos é chamada ft\_and\_inch\_to\_m:

def ft\_and\_inch\_to\_m(ft, inch):

return ft \* 0.3048 + inch \* 0.0254

print(ft\_and\_inch\_to\_m(1, 1))

O resultado de um teste rápido é:

0.3302

**output**

Parece como o esperado.

Nota: queríamos nomear o segundo parâmetro apenas in, não inch, mas não podíamos. Sabe porquê?

in é uma **keyword** Python - não pode ser usada como um nome.

Vamos converter *seis pés* em metros:

print(ft\_and\_inch\_to\_m(6, 0))

E esta é o output:

1.8288000000000002

**output**

É bem possível que por vezes se queira usar apenas os pés sem polegadas. O Python vai ajudá-lo? Claro que vai.

Modificámos um pouco o código:

def ft\_and\_inch\_to\_m(ft, inch = 0.0):

return ft \* 0.3048 + inch \* 0.0254

print(ft\_and\_inch\_to\_m(6))

Agora o parâmetro inch tem o seu valor padrão igual a 0.0.

O código produz o seguinte output - isto é o que é esperado:

1.8288000000000002

**output**

Finalmente, o código é capaz de responder à pergunta: qual é o IMC de uma pessoa com 1,75 m de altura e pesando 176 lbs?

Este é o código que construímos:

def ft\_and\_inch\_to\_m(ft, inch = 0.0):

return ft \* 0.3048 + inch \* 0.0254

def lb\_to\_kg(lb):

return lb \* 0.45359237

def bmi(weight, height):

if height < 1.0 or height > 2.5 or weight < 20 or weight > 200:

return None

return weight / height \*\* 2

print(bmi(weight = lb\_to\_kg(176), height = ft\_and\_inch\_to\_m(5, 7)))

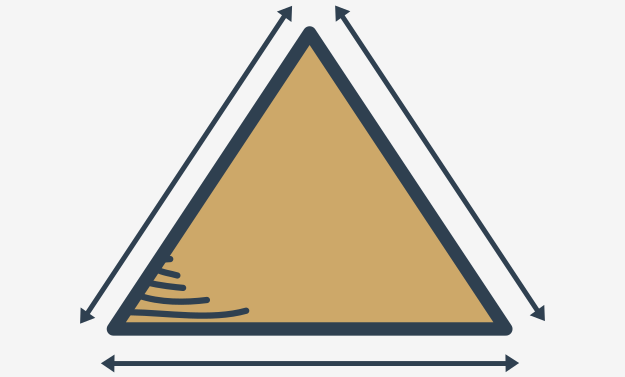
E a resposta é:

27.565214082533313

**output**

Execute o código e teste-o.

Vamos agora brincar com triângulos. Vamos começar com uma função para verificar se três lados de determinados comprimentos podem construir um triângulo.



Sabemos da escola que *a soma de dois lados arbitrários tem de ser mais longa do que o terceiro lado*.

Não será um desafio difícil. A função terá **três parâmetros** - um para cada lado.

Ela vai devolver True se os lados puderem construir um triângulo, e False caso contrário. Neste caso, is\_a\_triangle é um bom nome para tal função.

Veja o código no editor. Pode encontrar aí a nossa função. Execute o programa.

Parece que funciona bem - estes são os resultados:

True

False

**output**

Podemos torná-lo mais compacto? Parece um pouco palavroso.

Esta é uma versão mais compacta:

def is\_a\_triangle(a, b, c):

if a + b <= c or b + c <= a or c + a <= b:

return False

return True

print(is\_a\_triangle(1, 1, 1))

print(is\_a\_triangle(1, 1, 3))

Podemos compactá-lo ainda mais?

Sim, podemos - veja:

def is\_a\_triangle(a, b, c):

return a + b > c and b + c > a and c + a > b

print(is\_a\_triangle(1, 1, 1))

print(is\_a\_triangle(1, 1, 3))

Negámos a condição (invertemos os operadores relacionais e substituímos orpor and, recebendo uma expressão **universal para testar triângulos**).

Vamos instalar a função num programa maior. Vai pedir ao utilizador três valores e fará uso da função.

# triângulos e o teorema de Pitágoras

Veja o código no editor. Ele pede ao utilizador três valores. Em seguida, faz uso da função is\_a\_triangle . O código está pronto a ser executado.

No segundo passo, tentaremos assegurar que um certo triângulo é um **triângulo de ângulo reto**.

Precisamos de utilizar o **teorema de Pitágoras**:

**c2 = a2 + b2**

Como reconhecer qual dos três lados é a hipotenusa?

**A hipotenusa é o lado mais longo**.

Aqui está o código:

def is\_a\_triangle(a, b, c):

return a + b > c and b + c > a and c + a > b

def is\_a\_right\_triangle(a, b, c):

if not is\_a\_triangle(a, b, c):

return False

if c > a and c > b:

return c \*\* 2 == a \*\* 2 + b \*\* 2

if a > b and a > c:

return a \*\* 2 == b \*\* 2 + c \*\* 2

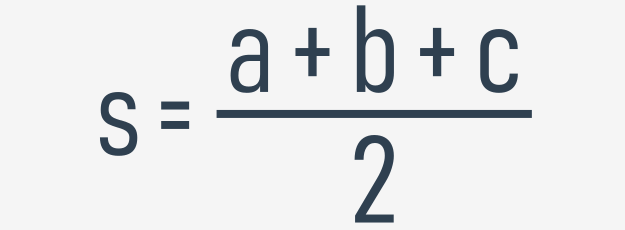
print(is\_a\_right\_triangle(5, 3, 4))

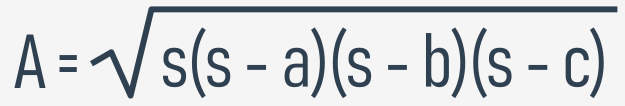
print(is\_a\_right\_triangle(1, 3, 4))

Veja como testamos a relação entre a hipotenusa e os restantes lados - escolhemos o lado mais longo, e aplicamos o **teorema de Pitágoras** para verificar se tudo está certo. Isto requer três verificações no total.

# avaliar a área de um triângulo

Também podemos avaliar a área de um triângulo. A **fórmula de Heron** será útil aqui:





Vamos usar o operador de exponenciação para encontrar a raiz quadrada - pode parecer estranho, mas funciona:



Este é o código resultante:

def is\_a\_triangle(a, b, c):

return a + b > c and b + c > a and c + a > b

def heron(a, b, c):

p = (a + b + c) / 2

return (p \* (p - a) \* (p - b) \* (p - c)) \*\* 0.5

def area\_of\_triangle(a, b, c):

if not is\_a\_triangle(a, b, c):

return None

return heron(a, b, c)

print(area\_of\_triangle(1., 1., 2. \*\* .5))

Tentamo-lo com um triângulo retângulo como metade de um quadrado com um lado igual a 1. Isto significa que a sua área deve ser igual a 0,5.

É estranho - o código produz o seguinte output:

0.49999999999999983

**output**

É muito perto de 0,5, mas não é exatamente 0,5. O que significa isto? É um erro?

Não, não é. Estas são **as especificidades dos cálculos de floating-point**. Em breve, contar-lhe-emos mais sobre o assunto.

# factorials

Outra função que estamos prestes a escrever é a **factorials**. Lembra-se de como um fatorial é definido?

0! = 1 (sim! é verdade)  
1! = 1  
2! = 1 \* 2  
3! = 1 \* 2 \* 3  
4! = 1 \* 2 \* 3 \* 4  
:  
:  
n! = 1 \* 2 \*\* 3 \* 4 \* ... \* n-1 \* n

É marcado com um **ponto de exclamação**, e é igual ao **produto** de todos os números naturais de um até ao seu argumento.

Vamos escrever o nosso código. Vamos criar uma função e chamá-la factorial\_function. Aqui está o código:

def factorial\_function(n):

if n < 0:

return None

if n < 2:

return 1

product = 1

for i in range(2, n + 1):

product \*= i

return product

for n in range(1, 6): # testing

print(n, factorial\_function(n))

Observe como espelhamos passo a passo a definição matemática, e como usamos o loop for para **encontrar o produto**.

Adicionamos um código de teste simples, e estes são os resultados que obtemos:

1 1

2 2

3 6

4 24

5 120

**output**

**números de Fibonacci**

Está familiarizado com os números de **Fibonacci**?

São uma **sequência de números inteiros** construída usando uma regra muito simples:

* o primeiro elemento da sequência é igual a um (**Fib1 = 1**)
* o segundo também é igual a um (**Fib2 = 1**)
* cada número subsequente é a soma dos dois números anteriores:  
  (**Fibi = Fibi-1 + Fibi-2**)

Aqui estão alguns dos primeiros números de Fibonacci:

fib\_1 = 1  
fib\_2 = 1  
fib\_3 = 1 + 1 = 2  
fib\_4 = 1 + 2 = 3  
fib\_5 = 2 + 3 = 5  
fib\_6 = 3 + 5 = 8  
fib\_7 = 5 + 8 = 13

O que pensa da **implementação disto como uma função**?

Vamos criar a nossa função fib e testá-la. Aqui está:

def fib(n):

if n < 1:

return None

if n < 3:

return 1

elem\_1 = elem\_2 = 1

the\_sum = 0

for i in range(3, n + 1):

the\_sum = elem\_1 + elem\_2

elem\_1, elem\_2 = elem\_2, the\_sum

return the\_sum

for n in range(1, 10): # testing

print(n, "->", fib(n))

Analise o corpo do loop for cuidadosamente, e descubra como **movemos as**elem\_1**e**elem\_2**variáveis através dos números de Fibonacci subsequentes**.

A parte de teste do código produz o seguinte output:

1 -> 1

2 -> 1

3 -> 2

4 -> 3

5 -> 5

6 -> 8

7 -> 13

8 -> 21

9 -> 34

**output**

# recursividade

Há mais uma coisa que queremos mostrar-lhe para tornar tudo completo - é a **recursividade**.

Este termo pode descrever muitos conceitos diferentes, mas um deles é especialmente interessante - o que se refere à programação informática.

Neste campo, a recursividade é uma **técnica em que uma função se invoca a si própria**.

Estes dois casos parecem ser os melhores para ilustrar o fenómeno - os números fatorials e Fibonacci. Especialmente o último.

**A definição dos números Fibonacci é um exemplo claro de recursividade**. Já lhe dissemos isso:

**Fibi = Fibi-1 + Fibi-2**

A definição do número i-th refere-se ao número i-1, e assim por diante, até se chegar aos dois primeiros.

Pode ser usado no código? Sim, pode. Pode também tornar o código mais curto e claro.

A segunda versão da nossa função fib() faz uso direto desta definição:

def fib(n):

if n < 1:

return None

if n < 3:

return 1

return fib(n - 1) + fib(n - 2)

O código é agora muito mais claro.

Mas é realmente seguro? Isto implica algum risco?

Sim, há de facto um pequeno risco. **Se se esquecer de considerar as condições que podem parar a cadeia de invocações recursivas, o programa pode entrar num loop infinito**. É preciso ter cuidado.

O fatorial tem também um segundo lado **recursivo**. Veja:

n! = 1 × 2 × 3 ×... × n-1 × n

É óbvio que:

1 × 2 × 3 ×... × n-1 = (n-1)!

Então, finalmente, o resultado é:

n! = (n-1)! × n

De facto, esta é uma receita pronta para a nossa nova solução.

Aqui está:

def factorial\_function(n):

if n < 0:

return None

if n < 2:

return 1

return n \* factorial\_function(n - 1)

Funciona? Sim, funciona. Experimente você mesmo.

A nossa curta viagem *funcional* está quase no fim. A secção seguinte tratará de dois curiosos tipos de dados Python: tuples e dicionários.

# Key takeaways

1. Uma função pode chamar outras funções ou até a ela própria. Quando uma função se chama a si própria, esta situação é conhecida como **recursividade**, e a função que se chama a si própria e contém uma condição de terminação especificada (ou seja, o caso base - uma condição que não diz à função para fazer mais chamadas para essa função) é chamada de função **recursiva**.

2. Pode usar funções recursivas em Python para escrever um **código limpo e elegante, e dividi-lo em pedaços mais pequenos e organizados**. Por outro lado, é preciso ter muito cuidado, pois pode ser **fácil cometer um erro e criar uma função que nunca termina**. Também é preciso lembrar que as **chamadas recursivas consomem muita memória**, e por isso podem por vezes ser ineficientes.

Ao utilizar a recursividade, é necessário ter em consideração todas as suas vantagens e desvantagens.

A função factorial é um exemplo clássico de como o conceito de recursividade pode ser posto em prática:

# Recursive implementation of the factorial function.

def factorial(n):

if n == 1: # The base case (termination condition.)

return 1

else:

return n \* factorial(n - 1)

print(factorial(4)) # 4 \* 3 \* 2 \* 1 = 24

**Exercício 1**

O que acontecerá quando se tentar executar o seguinte snippet e porquê?

def factorial(n):

return n \* factorial(n - 1)

print(factorial(4))

Verifique

**Exercício 2**

Qual é o output do seguinte snippet?

def fun(a):

if a > 30:

return 3

else:

return a + fun(a + 3)

print(fun(25))

# Tipos de sequência e mutabilidade

Antes de começarmos a falar sobre **tuples** e **dicionários**, temos de introduzir dois conceitos importantes: tipos de **sequência** e **mutabilidade**.

Um **tipo de sequência é um tipo de dados em Python que é capaz de armazenar mais de um valor (ou menos de um, pois uma sequência pode estar vazia), e estes valores podem ser pesquisados sequencialmente (daí o nome)**, elemento a elemento.

Como o loop for é uma ferramenta especialmente concebida para iterar através de sequências, podemos expressar a definição como: **uma sequência é um dado que pode ser analisado pelo loop**for.

Até agora encontrou uma sequência Python - a lista. A lista é um exemplo clássico de uma sequência Python, embora existam outras sequências que vale a pena mencionar, e vamos agora apresentar-lhas.

A segunda noção - a **mutabilidade** - é uma propriedade de qualquer dado Python que descreve a sua disponibilidade para ser livremente alterado durante a execução do programa. Existem dois tipos de dados Python: **mutáveis** e **imutáveis**.

**Os dados mutáveis podem ser atualizados livremente a qualquer momento** - chamamos esta operação in situ.

*In situ* é uma frase em latim que se traduz literalmente para *no local*. Por exemplo, a instrução a seguir modifica os dados in situ:

list.append(1)

**Dados imutáveis não podem ser modificados desta maneira**.

Imagine que uma lista só pode ser atribuída e lida. Não seria possível anexar-lhe um elemento, nem remover qualquer elemento da mesma. Isto significa que anexar um elemento ao fim da lista exigiria a recriação da lista a partir do zero.

Teria de construir uma lista completamente nova, constituída por todos os elementos da lista já existente, mais o novo elemento.

O tipo de dados de que lhe queremos falar agora é um **tuple**. **Um tuple é um tipo de sequência imutável**. Pode comportar-se como uma lista, mas não deve ser modificado in situ.

## O que é um tuple?

A primeira e mais clara distinção entre listas e tuples é a sintaxe utilizada para as criar - os **tuples preferem usar parêntesis curvos**, enquanto as listas gostam de ver parêntesis retos, embora também seja **possível criar um tuple a partir de um conjunto de valores separados por vírgulas**.

Veja o exemplo:

tuple\_1 = (1, 2, 4, 8)

tuple\_2 = 1., .5, .25, .125

Existem dois tuples, ambos contendo **quatro elementos**.

Vamos imprimi-los:

tuple\_1 = (1, 2, 4, 8)

tuple\_2 = 1., .5, .25, .125

print(tuple\_1)

print(tuple\_2)

Isto é o que você deve ver na consola:

(1, 2, 4, 8)

(1.0, 0.5, 0.25, 0.125)

**output**

Nota: **cada elemento tuple pode ser de um tipo diferente** (floating-point, inteiro, ou qualquer outro tipo de dados ainda não introduzidos).

## Como criar um tuple?

É possível criar um tuple vazio - são necessários então parêntesis:

empty\_tuple = ()

Se quiser criar um **tuple de um elemento**, tem de ter em consideração o facto de que, devido a razões de sintaxe (um tuple tem de ser distinguível de um valor comum e único), tem de terminar o valor com uma vírgula:

one\_element\_tuple\_1 = (1, )

one\_element\_tuple\_2 = 1.,

A remoção das vírgulas não estragará o programa em nenhum sentido sintático, mas em vez disso obterá duas variáveis únicas, não tuples.

# Como usar um tuple?

Se quiser obter os elementos de um tuple a fim de os ler, pode utilizar as mesmas convenções a que está habituado enquanto utiliza listas.

Dê uma vista de olhos ao código no editor.

my\_tuple = (1, 10, 100, 1000)

print(my\_tuple[0])

print(my\_tuple[-1])

print(my\_tuple[1:])

print(my\_tuple[:-2])

for elem in my\_tuple:

print(elem)

O programa deve produzir o seguinte output - execute-o e verifique:

1

1000

(10, 100, 1000)

(1, 10)

1

10

100

1000

**output**

As semelhanças podem ser enganadoras - **não tente modificar o conteúdo de um tuple**! Não é uma lista!

Todas estas instruções (exceto a mais acima) causarão um erro de runtime:

my\_tuple = (1, 10, 100, 1000)

my\_tuple.append(10000)

del my\_tuple[0]

my\_tuple[1] = -10

Esta é a mensagem que o Python lhe dará na janela da consola:

AttributeError: 'tuple' object has no attribute 'append'

**output**

Que mais podem os tuples fazer por si?

* a função len() aceita tuples, e devolve o número de elementos contidos no seu interior;
* o operador + pode juntar tuples (já lhe mostrámos isto)
* o operador \* pode multiplicar tuples, assim como listas;
* os operadores in e not in trabalham da mesma maneira que nas listas.

O snippet no editor apresenta-os a todos.

my\_tuple = (1, 10, 100)

t1 = my\_tuple + (1000, 10000)

t2 = my\_tuple \* 3

print(len(t2))

print(t1)

print(t2)

print(10 in my\_tuple)

print(-10 not in my\_tuple)

O output deve ter a seguinte aparência:

9

(1, 10, 100, 1000, 10000)

(1, 10, 100, 1, 10, 100, 1, 10, 100)

True

True

**output**

Uma das propriedades mais úteis da tuple é a sua capacidade de **aparecer no lado esquerdo do operador de atribuição**. Viu este fenómeno há algum tempo, quando foi necessário encontrar uma ferramenta elegante para trocar os valores de duas variáveis.

Dê uma vista de olhos no snippet abaixo:

var = 123

t1 = (1, )

t2 = (2, )

t3 = (3, var)

t1, t2, t3 = t2, t3, t1

print(t1, t2, t3)

Mostra três tuples que interagem - com efeito, os valores neles armazenados "circulam" - t1 torna-se t2, t2 torna-se t3, e t3 torna-se t1.

Nota: o exemplo apresenta mais um facto importante: os **elementos de um tuple podem ser variáveis**, e não apenas literais. Além disso, podem ser expressões se estiverem no lado direito do operador de atribuição.

# O que é um dicionário?

O **dicionário** é outra estrutura de dados Python. **Não é um tipo de sequência** (mas pode ser facilmente adaptado ao processamento de sequências) e é **mutável**.

Para explicar o que é realmente o dicionário Python, é importante compreender que se trata literalmente de um dicionário.

O dicionário Python funciona da mesma forma que **um dicionário bilingue**. Por exemplo, tem uma palavra inglesa (por exemplo, cat) e precisa do seu equivalente francês. Navega no dicionário para encontrar a palavra (pode usar diferentes técnicas para o fazer - não importa) e eventualmente obtém-na. A seguir, verifica o homólogo francês e é (muito provavelmente) a palavra "chat".



No mundo de Python, a palavra que se procura chama-se key. A palavra que obtém do dicionário chama-se um value.

Isto significa que um dicionário é um conjunto de pares de **key-values** (valores-chave). Nota:

* cada chave deve ser **única** - não é possível ter mais do que uma chave com o mesmo valor;
* uma chave pode ser **qualquer tipo de objeto imutável**: pode ser um número (inteiro ou float), ou mesmo uma string, mas não uma lista;
* um dicionário não é uma lista - uma lista contém um conjunto de valores numerados, enquanto que um **dicionário contém pares de valores**;
* a função len() funciona também para dicionários - devolve o número de elementos de key-value no dicionário;
* um dicionário é uma **ferramenta de sentido único** - se tiver um dicionário inglês-francês, pode procurar por equivalentes franceses de termos ingleses, mas não vice-versa.

Agora podemos mostrar-lhe alguns exemplos de trabalho.

## Como fazer um dicionário?

Se quiser atribuir alguns pares iniciais a um dicionário, deverá utilizar a seguinte sintaxe:

dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}

phone\_numbers = {'boss': 5551234567, 'Suzy': 22657854310}

empty\_dictionary = {}

print(dictionary)

print(phone\_numbers)

print(empty\_dictionary)

No primeiro exemplo, o dicionário usa chaves e valores que são ambas strings. No segundo, as chaves são strings, mas os valores são inteiros. A disposição inversa (teclas → números, valores → strings) também é possível, bem como a combinação número-número.

A lista de pares é **rodeada por chavetas**, enquanto os pares em si são **separados por vírgulas**, e as chaves e valores por **dois pontos**.

O primeiro dos nossos dicionários é um dicionário inglês-francês muito simples. O segundo - uma diretoria telefónica muito pequena.

Os dicionários vazios são construídos por um **par vazio de chavetas** - nada de invulgar.

O dicionário como um todo pode ser impresso com uma única invocação print() . O snippet **pode** produzir o seguinte output:

{'dog': 'chien', 'horse': 'cheval', 'cat': 'chat'}

{'Suzy': 5557654321, 'boss': 5551234567}

{}

**output**

Notou alguma coisa surpreendente? A ordem dos pares impressos é diferente do que na atribuição inicial. O que é que isso significa?

Em primeiro lugar, é uma confirmação de que **os dicionários não são listas** - não preservam a ordem dos seus dados, uma vez que a ordem não tem qualquer significado (ao contrário do que acontece nos dicionários de papel reais). A ordem em que um dicionário **armazena os seus dados está completamente fora do seu controlo**, e das suas expectativas. Isso é normal. (\*)

NOTA

(\*) Em Python 3.6x os dicionários tornaram-se coleções **ordenadas** por defeito. Os seus resultados podem variar dependendo da versão Python que estiver a utilizar.

**Como usar um dicionário?**

Se quiser obter algum dos valores, tem de entregar um key-value válido:

print(dictionary['cat'])

print(phone\_numbers['Suzy'])

A obtenção do valor de um dicionário assemelha-se a uma indexação, especialmente graças aos parêntesis retos que rodeiam o valor da chave.

Nota:

* se a chave for uma string, é necessário especificá-la como uma string;
* **as chaves são sensíveis a maiúsculas e minúsculas**: 'Suzy' é algo diferente de 'suzy'.

O snippet produz duas linhas de texto:

chat

5557654321

**output**

E agora a notícia mais importante: **não se deve usar uma chave inexistente**. Tentar algo assim:

print(phone\_numbers['president'])

causará um erro de runtime. Tente fazê-lo.

Felizmente, há uma maneira simples de evitar tal situação. O operador in , juntamente com o seu companheiro, not in, podem salvar esta situação.

O seguinte código procura com segurança algumas palavras francesas:

dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}

words = ['cat', 'lion', 'horse']

for word in words:

if word in dictionary:

print(word, "->", dictionary[word])

else:

print(word, "is not in dictionary")

O output do código é o seguinte:

cat -> chat

lion is not in dictionary

horse -> cheval

**output**

NOTA

Quando se escreve uma expressão grande ou longa, pode ser uma boa ideia mantê-la verticalmente alinhada. É assim que pode tornar o seu código mais legível e mais fácil de programar, por exemplo

# Example 1:

dictionary = {

"cat": "chat",

"dog": "chien",

"horse": "cheval"

}

# Example 2:

phone\_numbers = {'boss': 5551234567,

'Suzy': 22657854310

}

Tais formas de formatação de código são chamadas **hanging indents** (indentações penduradas).

# o keys()

Os dicionários podem ser **consultados** usando o loop for , como listas ou tuples?

Não e sim.

Não, porque um dicionário **não é um tipo de sequência** - o loop for é inútil com ele.

Sim, porque existem ferramentas simples e muito eficazes que podem **adaptar qualquer dicionário aos**for**requisitos do loop** (por outras palavras, construindo um link intermediário entre o dicionário e uma entidade de sequência temporária).

O primeiro deles é um método chamado keys(), possuído por cada dicionário. O método **devolve um objeto iterável que consiste em todas as chaves recolhidas dentro do dicionário**. Ter um grupo de chaves permite-lhe aceder a todo o dicionário de uma forma fácil e prática.

Tal como aqui:

dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}

for key in dictionary.keys():

print(key, "->", dictionary[key]

O output do código é o seguinte:

horse -> cheval

dog -> chien

cat -> chat

**output**

## O método sorted() .

Quer que seja **classificado**? Apenas enriqueça o loop for , para obter tal forma:

for key in sorted(dictionary.keys()):

A função sorted() fará o seu melhor - o output irá ficar assim:

cat -> chat

dog -> chien

horse -> cheval

**output**

# Os métodos items() e values() .

Outra forma é baseada na utilização de um método de dicionário chamado items(). O método **devolve tuples** (este é o primeiro exemplo onde os tuples são algo mais do que apenas um exemplo de si mesmos) **onde cada tuple é um par key-value**.

É assim que funciona:

dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}

for english, french in dictionary.items():

print(english, "->", french)

Note-se a forma como o tuple foi utilizado como uma for variável de loop.

O exemplo imprime:

cat -> chat

dog -> chien

horse -> cheval

**output**

Há também um método chamado values(), que funciona de forma semelhante a keys(), mas **devolve valores**.

Aqui está um exemplo simples:

dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}

for french in dictionary.values():

print(french)

Como o dicionário não consegue encontrar automaticamente uma chave para um determinado valor, o papel deste método é bastante limitado.

Este é o output esperado:

cheval

chien

chat

**output**

# modificar e adicionar valores

Atribuir um novo valor a uma chave existente é simples - visto os dicionários serem totalmente **mutáveis**, não há obstáculos para os modificar.

Vamos substituir o valor "chat" com "minou", o que não é muito preciso, mas funcionará bem com o nosso exemplo.

Veja:

dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}

dictionary['cat'] = 'minou'

print(dictionary)

O output é:

{'cat': 'minou', 'dog': 'chien', 'horse': 'cheval'}

**output**

## Adicionar uma nova chave

Adicionar um novo par key-value a um dicionário é tão simples como alterar um valor - basta atribuir um valor a uma nova **chave, previamente inexistente**.

Nota: este é um comportamento muito diferente em relação às listas, que não permitem atribuir valores a índices inexistentes.

Vamos adicionar um novo par de palavras ao dicionário - um pouco estranho, mas ainda assim válido:

dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}

dictionary['swan'] = 'cygne'

print(dictionary)

O output do exemplo:

{'cat': 'chat', 'dog': 'chien', 'horse': 'cheval', 'swan': 'cygne'}

**output**

**EXTRA**

Também pode inserir um item num dicionário utilizando o método update() , por exemplo:

dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}

dictionary.update({"duck": "canard"})

print(dictionary)

## Remover uma chave

Consegue adivinhar como se remove uma chave de um dicionário?

Nota: remover uma chave causará sempre a **remoção do valor associado**. **Valores não podem existir sem as suas chaves**.

Isto é feito com a del instrução.

Aqui está o exemplo:

dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}

del dictionary['dog']

print(dictionary)

Nota: **a remoção de uma chave não existente causa um erro**.

O output do exemplo:

{'cat': 'chat', 'horse': 'cheval'}

**output**

**EXTRA**

Para remover o último item num dicionário, pode-se utilizar o método popitem() :

dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}

dictionary.popitem()

print(dictionary) # outputs: {'cat': 'chat', 'dog': 'chien'}

Nas versões mais antigas do Python, ou seja, antes do 3.6.7, o método popitem() remove um item aleatório de um dicionário.

**Tuples e dicionários podem trabalhar em conjunto**

Preparámos um exemplo simples, mostrando como tuples e dicionários podem trabalhar em conjunto.

Vamos imaginar o seguinte problema:

* é necessário um programa para avaliar as notas médias dos alunos;
* o programa deve pedir o nome do aluno, seguido da sua pontuação única;
* Os nomes podem ser indicados em qualquer ordem;
* a introdução de um nome vazio termina a introdução dos dados (nota 1: a introdução de uma pontuação vazia levantará a exceção ValueError, mas não se preocupe com isso agora, verá como lidar com tais casos quando falarmos de exceções na segunda parte da série de cursos Python Essentials)
* uma lista de todos os nomes, juntamente com a pontuação média avaliada, deve então ser emitida.

Veja o código no editor. Esta é a forma de o fazer.

Agora, vamos analisá-lo linha a linha:

* **linha 1**: criar um dicionário vazio para os dados introduzidos; o nome do aluno é usado como chave, enquanto todas as notas associadas são armazenadas num tuple (o tuple pode ser um valor de dicionário - isso não é um problema)
* **linha 3**: entrar num loop "infinito" (não se preocupe, vai quebrar-se no momento certo)
* **linha 4**: ler o nome do aluno aqui;
* **linha 5-6**: se o nome for uma string vazia (), leave the loop;
* **linha 8**: pedir uma das pontuações do aluno (um inteiro do intervalo 0-10)
* **linha 9-10**: se a pontuação introduzida não estiver dentro do intervalo de 0 a 10, deixar o loop;
* **linha 12-13**: se o nome do aluno já estiver no dicionário, alongar o tuple associado com a nova pontuação (note o operador +=)
* **linha 14-15**: se este for um novo aluno (desconhecido do dicionário), criar uma nova entrada - o seu valor é um tuple de um elemento contendo a pontuação introduzida;
* **linha 17**: iterar através dos nomes dos alunos ordenados;
* **linha 18-19**: inicializar os dados necessários para avaliar a média (soma e contador)
* **linha 20-22**: iteramos através do tuple, tomando todas as pontuações subsequentes e atualizando a soma, juntamente com o contador;
* **linha 23**: avaliar e imprimir o nome e a pontuação média do aluno.

Este é um registo de uma conversa que tivemos com o nosso programa:

Enter the student's name: Bob

Enter the student's score (0-10): 7

Enter the student's name: Andy

Enter the student's score (0-10): 3

Enter the student's name: Bob

Enter the student's score (0-10): 2

Enter the student's name: Andy

Enter the student's score (0-10): 10

Enter the student's name: Andy

Enter the student's score (0-10): 3

Enter the student's name: Bob

Enter the student's score (0-10): 9

Enter the student's name:

Andy : 5.333333333333333

Bob : 6.0

**output**

# Key takeaways: tuples

1. Os **tuples** são coleções de dados ordenados e imutáveis. Podem ser imaginados como listas imutáveis. São escritos entre parêntesis curvos:

my\_tuple = (1, 2, True, "a string", (3, 4), [5, 6], None)

print(my\_tuple)

my\_list = [1, 2, True, "a string", (3, 4), [5, 6], None]

print(my\_list)

Cada elemento tuple pode ser de um tipo diferente (ou seja, inteiros, strings, booleanos, etc.). Além disso, os tuples podem conter outros tuples ou listas (e o inverso).

2. Pode-se criar um tuple vazio como este:

empty\_tuple = ()

print(type(empty\_tuple)) # outputs: <class 'tuple'>

3. Um tuple de um elemento pode ser criado da seguinte forma:

one\_elem\_tuple\_1 = ("one", ) # Brackets and a comma.

one\_elem\_tuple\_2 = "one", # No brackets, just a comma.

Se remover a vírgula, dirá ao Python para criar uma **variável** e não um tuple:

my\_tuple\_1 = 1,

print(type(my\_tuple\_1)) # outputs: <class 'tuple'>

my\_tuple\_2 = 1 # This is not a tuple.

print(type(my\_tuple\_2)) # outputs: <class 'int'>

4. Pode aceder aos elementos do tuple através da indexação:

my\_tuple = (1, 2.0, "string", [3, 4], (5, ), True)

print(my\_tuple[3]) # outputs: [3, 4]

5. Os tuples são **imutáveis**, o que significa que não se pode alterar os seus elementos (não se pode anexar tuples, ou modificar, ou remover elementos tuple). O snippet a seguir causará uma exceção:

my\_tuple = (1, 2.0, "string", [3, 4], (5, ), True)

my\_tuple[2] = "guitar" # The TypeError exception will be raised.

No entanto, é possível apagar um tuple como um todo:

my\_tuple = 1, 2, 3,

del my\_tuple

print(my\_tuple) # NameError: name 'my\_tuple' is not defined

6. Pode fazer um loop através dos elementos de um tuple (Exemplo 1), verificar se um elemento específico está (ou não) presente num tuple (Exemplo 2), utilizar a função len() para verificar quantos elementos existem num tuple (Exemplo 3), ou mesmo juntar/multiplicar tuples (Exemplo 4):

# Example 1

tuple\_1 = (1, 2, 3)

for elem in tuple\_1:

print(elem)

# Example 2

tuple\_2 = (1, 2, 3, 4)

print(5 in tuple\_2)

print(5 not in tuple\_2)

# Example 3

tuple\_3 = (1, 2, 3, 5)

print(len(tuple\_3))

# Example 4

tuple\_4 = tuple\_1 + tuple\_2

tuple\_5 = tuple\_3 \* 2

print(tuple\_4)

print(tuple\_5)

**EXTRA**

Também pode criar um tuple usando uma função Python incorporada chamada tuple(). Isto é particularmente útil quando se pretende converter um certo iterável (por exemplo, uma lista, range, string, etc.) num tuple:

my\_tuple = tuple((1, 2, "string"))

print(my\_tuple)

my\_list = [2, 4, 6]

print(my\_list) # outputs: [2, 4, 6]

print(type(my\_list)) # outputs: <class 'list'>

tup = tuple(my\_list)

print(tup) # outputs: (2, 4, 6)

print(type(tup)) # outputs: <class 'tuple'>

Da mesma forma, quando se pretende converter um iterável numa lista, pode-se usar uma função Python integrada chamada list():

tup = 1, 2, 3,

my\_list = list(tup)

print(type(my\_list)) # outputs: <class 'list'>

# Key takeaways: dicionários

1. Os dicionários são coleções de dados desordenados**\***, alteráveis (mutáveis), e indexados. (**\***Em Python 3.6x, os dicionários tornaram-se ordenados por defeito.

Cada dicionário é um conjunto de *chaves: pares de valores*. Pode criá-lo usando a seguinte sintaxe:

my\_dictionary = {

key1: value1,

key2: value2,

key3: value3,

}

2. Se quiser aceder a um item de dicionário, pode fazê-lo fazendo referência à sua chave dentro de um par de parêntesis retos (ex. 1) ou usando o método get() (ex. 2):

pol\_eng\_dictionary = {

"kwiat": "flower",

"woda": "water",

"gleba": "soil"

}

item\_1 = pol\_eng\_dictionary["gleba"] # ex. 1

print(item\_1) # outputs: soil

item\_2 = pol\_eng\_dictionary.get("woda")

print(item\_2) # outputs: water

3. Se quiser alterar o valor associado a uma chave específica, pode fazê-lo referindo-se ao nome da chave do item da seguinte forma:

pol\_eng\_dictionary = {

"zamek": "castle",

"woda": "water",

"gleba": "soil"

}

pol\_eng\_dictionary["zamek"] = "lock"

item = pol\_eng\_dictionary["zamek"]

print(item) # outputs: lock

4. Para adicionar ou remover uma chave (e o valor associado), utilize a seguinte sintaxe:

phonebook = {} # an empty dictionary

phonebook["Adam"] = 3456783958 # create/add a key-value pair

print(phonebook) # outputs: {'Adam': 3456783958}

del phonebook["Adam"]

print(phonebook) # outputs: {}

Também pode inserir um item num dicionário utilizando o método update() , e remover o último elemento usando o método popitem() , por exemplo:

pol\_eng\_dictionary = {"kwiat": "flower"}

pol\_eng\_dictionary.update({"gleba": "soil"})

print(pol\_eng\_dictionary) # outputs: {'kwiat': 'flower', 'gleba': 'soil'}

pol\_eng\_dictionary.popitem()

print(pol\_eng\_dictionary) # outputs: {'kwiat': 'flower'}

5. Pode utilizar o loop for para percorrer um dicionário, por exemplo:

pol\_eng\_dictionary = {

"zamek": "castle",

"woda": "water",

"gleba": "soil"

}

for item in pol\_eng\_dictionary:

print(item)

# outputs: zamek

# woda

# gleba

6. Se quiser percorrer as chaves e valores de um dicionário, pode usar o método items() , por exemplo:

pol\_eng\_dictionary = {

"zamek": "castle",

"woda": "water",

"gleba": "soil"

}

for key, value in pol\_eng\_dictionary.items():

print("Pol/Eng ->", key, ":", value)

7. Para verificar se uma determinada chave existe num dicionário, pode utilizar a in keyword:

pol\_eng\_dictionary = {

"zamek": "castle",

"woda": "water",

"gleba": "soil"

}

if "zamek" in pol\_eng\_dictionary:

print("Yes")

else:

print("No")

8. Pode utilizar a keyword del para remover um item específico, ou apagar um dicionário. Para remover todos os itens do dicionário, é necessário utilizar o clear() método:

pol\_eng\_dictionary = {

"zamek": "castle",

"woda": "water",

"gleba": "soil"

}

print(len(pol\_eng\_dictionary)) # outputs: 3

del pol\_eng\_dictionary["zamek"] # remove an item

print(len(pol\_eng\_dictionary)) # outputs: 2

pol\_eng\_dictionary.clear() # removes all the items

print(len(pol\_eng\_dictionary)) # outputs: 0

del pol\_eng\_dictionary # removes the dictionary

9. Para copiar um dicionário, use o copy() método:

pol\_eng\_dictionary = {

"zamek": "castle",

"woda": "water",

"gleba": "soil"

}

copy\_dictionary = pol\_eng\_dictionary.copy()

# Key takeaways: tuples e dicionários

**Exercício 1**

O que acontece quando se tenta executar o seguinte snippet?

my\_tup = (1, 2, 3)

print(my\_tup[2])

Verifique

**Exercício 2**

Qual é o output do seguinte snippet?

tup = 1, 2, 3

a, b, c = tup

print(a \* b \* c)

Verifique

**Exercício 3**

Complete o código para utilizar corretamente o método count() para encontrar o número de duplicados de 2 no seguinte tuple.

tup = 1, 2, 3, 2, 4, 5, 6, 2, 7, 2, 8, 9

duplicates = # Write your code here.

print(duplicates) # outputs: 4

Verifique

**Exercício 4**

Escreva um programa que irá "colar" os dois dicionários (d1 e d2) e criar um novo (d3).

d1 = {'Adam Smith': 'A', 'Judy Paxton': 'B+'}

d2 = {'Mary Louis': 'A', 'Patrick White': 'C'}

d3 = {}

for item in (d1, d2):

# Write your code here.

print(d3)

Verifique

**Exercício 5**

Escreva um programa que irá converter a lista my\_list num tuple.

my\_list = ["car", "Ford", "flower", "Tulip"]

t = # Write your code here.

print(t)

Verifique

**Exercício 6**

Escreva um programa que irá converter o tuple colors num dicionário.

colors = (("green", "#008000"), ("blue", "#0000FF"))

# Write your code here.

print(colors\_dictionary)

Verifique

**Exercício 7**

O que acontecerá quando executar o seguinte código?

my\_dictionary = {"A": 1, "B": 2}

copy\_my\_dictionary = my\_dictionary.copy()

my\_dictionary.clear()

print(copy\_my\_dictionary)

Verifique

**Exercício 8**

Qual é o output do seguinte programa?

colors = {

"white": (255, 255, 255),

"grey": (128, 128, 128),

"red": (255, 0, 0),

"green": (0, 128, 0)

}

for col, rgb in colors.items():

print(col, ":", rgb)

## Tempo estimado

30-120 minutos

## Nível de dificuldade

Médio/Difícil

## Objetivos

* aperfeiçoar as competências do aluno na utilização de Python para a resolução de problemas complexos,
* integração de técnicas de programação num programa consituído por várias partes.

## Cenário

A sua tarefa é escrever **um programa simples que finja jogar *tic-tac-toe* com o utilizador**. Para lhe facilitar as coisas, decidimos simplificar o jogo. Aqui estão os nossos pressupostos:

* o computador (ou seja, o seu programa) deve jogar o jogo utilizando 'X's;
* o utilizador (por exemplo, você) deve jogar o jogo utilizando 'O's;
* o primeiro movimento pertence ao computador - coloca sempre o seu primeiro 'X' no meio do tabuleiro;
* todos os quadrados são numerados fila a fila começando por 1 (veja a sessão de exemplo abaixo, para referência)
* o utilizador introduz a sua jogada inserindo o número do quadrado que escolhe - o número deve ser válido, ou seja, deve ser um número inteiro, deve ser maior do que 0 e menos do que 10, e não pode apontar para um campo que já esteja ocupado;
* o programa verifica se o jogo terminou - há quatro vereditos possíveis: o jogo deve continuar, ou o jogo termina com um empate, a sua vitória ou a vitória do computador;
* o computador responde com a sua jogada e a verificação é repetida;
* não implemente nenhuma forma de inteligência artificial - uma escolha de campo aleatória feita pelo computador é suficientemente boa para o jogo.

A sessão de exemplo com o programa pode ter a aparência seguinte:

+-------+-------+-------+

| | | |

| 1 | 2 | 3 |

| | | |

+-------+-------+-------+

| | | |

| 4 | X | 6 |

| | | |

+-------+-------+-------+

| | | |

| 7 | 8 | 9 |

| | | |

+-------+-------+-------+

Enter your move: 1

+-------+-------+-------+

| | | |

| O | 2 | 3 |

| | | |

+-------+-------+-------+

| | | |

| 4 | X | 6 |

| | | |

+-------+-------+-------+

| | | |

| 7 | 8 | 9 |

| | | |

+-------+-------+-------+

+-------+-------+-------+

| | | |

| O | X | 3 |

| | | |

+-------+-------+-------+

| | | |

| 4 | X | 6 |

| | | |

+-------+-------+-------+

| | | |

| 7 | 8 | 9 |

| | | |

+-------+-------+-------+

Enter your move: 8

+-------+-------+-------+

| | | |

| O | X | 3 |

| | | |

+-------+-------+-------+

| | | |

| 4 | X | 6 |

| | | |

+-------+-------+-------+

| | | |

| 7 | O | 9 |

| | | |

+-------+-------+-------+

+-------+-------+-------+

| | | |

| O | X | 3 |

| | | |

+-------+-------+-------+

| | | |

| 4 | X | X |

| | | |

+-------+-------+-------+

| | | |

| 7 | O | 9 |

| | | |

+-------+-------+-------+

Enter your move: 4

+-------+-------+-------+

| | | |

| O | X | 3 |

| | | |

+-------+-------+-------+

| | | |

| O | X | X |

| | | |

+-------+-------+-------+

| | | |

| 7 | O | 9 |

| | | |

+-------+-------+-------+

+-------+-------+-------+

| | | |

| O | X | X |

| | | |

+-------+-------+-------+

| | | |

| O | X | X |

| | | |

+-------+-------+-------+

| | | |

| 7 | O | 9 |

| | | |

+-------+-------+-------+

Enter your move: 7

+-------+-------+-------+

| | | |

| O | X | X |

| | | |

+-------+-------+-------+

| | | |

| O | X | X |

| | | |

+-------+-------+-------+

| | | |

| O | O | 9 |

| | | |

+-------+-------+-------+

You won!

## Requisitos

Implemente as seguintes características:

* o tabuleiro deve ser armazenado como uma lista de três elementos, enquanto cada elemento é outra lista de três elementos (as listas internas representam linhas) de modo a que todos os quadrados possam ser acedidos utilizando a seguinte sintaxe:  
    
  board[row][column]
* cada um dos elementos da lista interna pode conter 'O', 'X', ou um dígito que representa o número do quadrado (tal quadrado é considerado livre)
* a aparência do tabuleiro deve ser exatamente a mesma que a apresentada no exemplo.
* implemente as funções definidas para si no editor.

O desenho de um número inteiro aleatório pode ser feito utilizando uma função Python chamada randrange(). O programa de exemplo abaixo mostra como utilizá-lo (o programa imprime dez números aleatórios de 0 a 8).

Nota: a instrução from-import fornece um acesso à função randrange definida dentro de um módulo externo de Python chamado random.

from random import randrange

for i in range(10):

print(randrange(8))