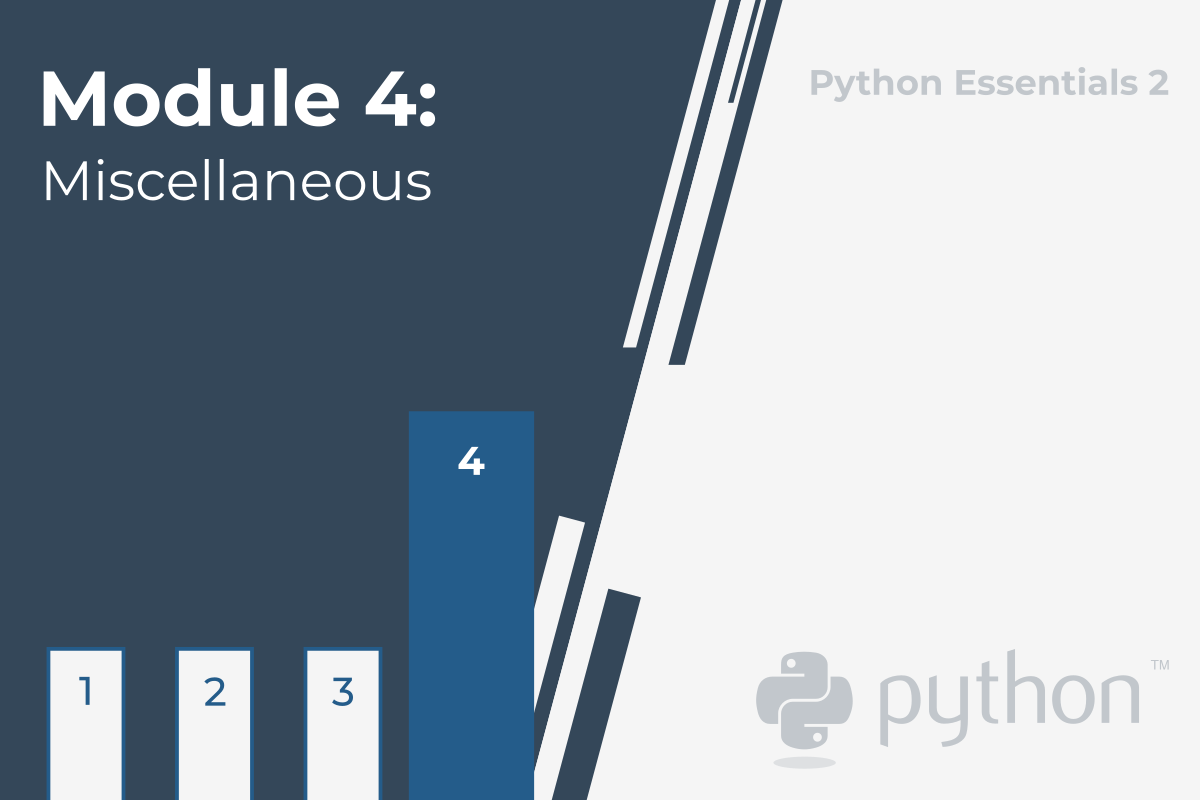
**Python Essentials 2:  
Módulo 4**

**Diversos**

Neste módulo, aprenderá sobre:

* Geradores, iteradores e closures;
* Trabalhar com sistemas-ficheiro, árvore de diretorias e ficheiros;
* Módulos selecionados da Biblioteca Padrão de Python (*os*, *datetime*, *time*, e *calendar*.)



# Geradores - onde encontrá-los

**Gerador** - a que associa esta palavra? Talvez se refira a algum dispositivo eletrónico. Ou talvez se refira a uma máquina pesada e séria concebida para produzir energia, elétrica ou outra.

Um gerador Python é **uma peça de código especializado capaz de produzir uma série de valores, e de controlar o processo de iteração**. É por isso que os geradores são muitas vezes chamados **iteradores**, e embora alguns possam encontrar uma distinção muito subtil entre estes dois, vamos tratá-los como um só.

Pode não se aperceber, mas já encontrou geradores muitas, muitas vezes antes. Veja este snippet muito simples:

for i in range(5):

print(i)

A função range() é, de facto, um gerador, que é (de facto, mais uma vez) um iterador.

Qual é a diferença?

Uma função devolve um valor bem definido - pode ser o resultado de uma avaliação mais ou menos complexa de, por exemplo, um polinómio, e é invocada uma vez - apenas uma vez.

Um gerador **devolve uma série de valores** e, em geral, é (implicitamente) invocado mais de uma vez.

No exemplo, o gerador range() é invocado seis vezes, fornecendo cinco valores subsequentes de zero a quatro, e finalmente sinalizando que a série está completa.

O processo acima é completamente transparente. Vamos lançar alguma luz sobre ele. Vamos mostrar-lhe o **protocolo iterador**.

O **protocolo iterador é uma forma de um objeto se comportar de acordo com as regras impostas pelo contexto das declarações**for**e**in. Um objeto em conformidade com o protocolo iterador é chamado um **iterador**.

Um iterador deve fornecer dois métodos:

* \_\_iter\_\_() que deve **devolver o objeto em si** e que é invocado uma vez (é necessário para que Python inicie com sucesso a iteração)
* \_\_next\_\_() que se destina a **devolver o próximo valor** (primeiro, segundo, etc.) da série desejada - será invocado pelas declarações for/in a fim de passar pela próxima iteração; se não houver mais valores a fornecer, o método deve **levantar a exceção**StopIteration.

Parece estranho? De modo algum. Veja o exemplo no editor.

* class Fib:
* def \_\_init\_\_(self, nn):
* print("\_\_init\_\_")
* self.\_\_n = nn
* self.\_\_i = 0
* self.\_\_p1 = self.\_\_p2 = 1
* def \_\_iter\_\_(self):
* print("\_\_iter\_\_")
* return self
* def \_\_next\_\_(self):
* print("\_\_next\_\_")
* self.\_\_i += 1
* if self.\_\_i > self.\_\_n:
* raise StopIteration
* if self.\_\_i in [1, 2]:
* return 1
* ret = self.\_\_p1 + self.\_\_p2
* self.\_\_p1, self.\_\_p2 = self.\_\_p2, ret
* return ret
* for i in Fib(10):
* print(i)

Construímos uma classe capaz de iterar através dos primeiros valores n (onde n é um parâmetro construtor) dos números de Fibonacci.

Recordemos - os números Fibonacci (Fibi) são definidos da seguinte forma:

Fib1 = 1  
Fib2 = 1  
Fibi = Fibi-1 + Fibi-2

Por outras palavras:

* os dois primeiros números de Fibonacci são iguais a 1;
* qualquer outro número de Fibonacci é a soma dos dois anteriores (por exemplo, Fib3 = 2, Fib4 = 3, Fib5 = 5, e assim por diante)

Vamos mergulhar no código:

* linhas 2 a 6: o construtor da classe imprime uma mensagem (vamos usá-la para rastrear o comportamento da classe), prepara algumas variáveis (\_\_n para armazenar o limite da série, \_\_i para rastrear o número atual de Fibonacci a fornecer, e \_\_p1 juntamente com \_\_p2 para guardar os dois números anteriores);

* linhas 8 a 10: o método \_\_iter\_\_ é obrigado a devolver o próprio objeto iterador; o seu objetivo pode ser um pouco ambíguo aqui, mas não há mistério; tente imaginar um objeto que não seja um iterador (por exemplo, é uma coleção de algumas entidades), mas um dos seus componentes é um iterador capaz de digitalizar a coleção; o método \_\_iter\_\_ deve **extrair o iterador e confiar-lhe a execução do protocolo de iteração**; como pode ver, o método inicia a sua ação imprimindo uma mensagem;

* linhas 12 a 21: o método \_\_next\_\_ é responsável pela criação da sequência; é um tanto ou quanto minucioso, mas isto deve torná-lo mais legível; primeiro, imprime uma mensagem, depois atualiza o número de valores desejados, e se chegar ao fim da sequência, o método quebra a iteração levantando a exceção StopIteration; o resto do código é simples, e reflete precisamente a definição que lhe mostrámos anteriormente;

* as linhas 24 e 25 fazem uso do iterador.

O código produz o seguinte output:

\_\_init\_\_

\_\_iter\_\_

\_\_next\_\_

1

\_\_next\_\_

1

\_\_next\_\_

2

\_\_next\_\_

3

\_\_next\_\_

5

\_\_next\_\_

8

\_\_next\_\_

13

\_\_next\_\_

21

\_\_next\_\_

34

\_\_next\_\_

55

\_\_next\_\_

**output**

Veja:

* o objeto iterador é instanciado primeiro;
* em seguida, o Python invoca o método \_\_iter\_\_ para obter acesso ao iterador real;
* o método \_\_next\_\_ é invocado onze vezes - as primeiras dez vezes produzem valores úteis, enquanto a décima primeira termina a iteração.

O exemplo anterior mostra uma solução em que o **objeto iterador faz parte de uma classe mais complexa**.

O código não é realmente sofisticado, mas apresenta o conceito de forma clara.

Dê uma vista de olhos ao código no editor.

class Fib:

def \_\_init\_\_(self, nn):

self.\_\_n = nn

self.\_\_i = 0

self.\_\_p1 = self.\_\_p2 = 1

def \_\_iter\_\_(self):

print("Fib iter")

return self

def \_\_next\_\_(self):

self.\_\_i += 1

if self.\_\_i > self.\_\_n:

raise StopIteration

if self.\_\_i in [1, 2]:

return 1

ret = self.\_\_p1 + self.\_\_p2

self.\_\_p1, self.\_\_p2 = self.\_\_p2, ret

return ret

class Class:

def \_\_init\_\_(self, n):

self.\_\_iter = Fib(n)

def \_\_iter\_\_(self):

print("Class iter")

return self.\_\_iter;

object = Class(8)

for i in object:

print(i)

Nós construímos o iterador Fib noutra classe (podemos dizer que o compusemos na classe Class ). É instanciado juntamente com Class do objeto.

O objeto da classe pode ser usado como um iterador quando (e apenas quando) responde positivamente à invocação \_\_iter\_\_ - esta classe pode fazê-lo, e se for invocada desta forma, fornece um objeto capaz de obedecer ao protocolo de iteração.

É por isto que o output do código é o mesma que anteriormente, embora o objeto da classe Fib não seja usado explicitamente dentro do contexto do loop for .

# A função yield .

O protocolo iterador não é particularmente difícil de compreender e utilizar, mas também é indiscutível que o **protocolo é bastante inconveniente**.

O principal desconforto que traz é **a necessidade de guardar o estado da iteração entre subsequentes invocações**\_\_iter\_\_.

Por exemplo, o iterador Fib é forçado a armazenar com precisão o local onde a última invocação foi interrompida (ou seja, o número avaliado e os valores dos dois elementos anteriores). Isso torna o código maior e menos compreensível.

É por isso que o Python oferece uma maneira muito mais eficaz, conveniente e elegante de escrever iteradores.

O conceito é fundamentalmente baseado num mecanismo muito específico e poderoso fornecido pela keyword yield .

Pode pensar na keyword yield como uma irmã mais inteligente da declaração return , com uma diferença essencial.

Dê uma vista de olhos nesta função:

def fun(n):

for i in range(n):

return i

Tem um aspecto estranho, não tem? É claro que o loop for não tem qualquer hipótese de terminar a sua primeira execução, uma vez que o return vai quebrá-lo irrevogavelmente.

Além disso, invocar a função não mudará nada - o loop for começará do zero e será quebrado imediatamente.

Podemos dizer que tal função não é capaz de guardar e restaurar o seu estado entre invocações subsequentes.

Isto também significa que uma função como esta **não pode ser usada como gerador**.

Substituímos exatamente uma palavra no código - consegue vê-la?

def fun(n):

for i in range(n):

yield i

Adicionámos yield em vez de return. Esta pequena emenda **transforma a função num gerador**, e executar a declaração yield tem alguns efeitos muito interessantes.

Antes de mais, fornece o valor da expressão especificada após a keyword yield , assim como return, mas não perde o estado da função.

Todos os valores das variáveis são congelados e aguardam a próxima invocação, quando a execução é retomada (não tirada do zero, como depois return).

Há uma limitação importante: tal **função não deve ser invocada explicitamente** porque - na realidade - já não é uma função; **é um objeto gerador**.

A invocação **devolverá o identificador do objeto**, não a série que esperamos do gerador.

Devido às mesmas razões, a função anterior (a função com a declaração return ) só pode ser invocada explicitamente, e não deve ser usada como um gerador.

# Como construir um gerador

Deixe-nos mostrar-lhe o novo gerador em ação.

É assim que podemos utilizá-lo:

def fun(n):

for i in range(n):

yield i

for v in fun(5):

print(v)

Consegue adivinhar o output?

# Como construir o seu próprio gerador

E se precisar de um **gerador para produzir as primeiras *n* potências de *2***?

Nada mais fácil. Basta olhar para o código abaixo:

def powers\_of\_2(n):

power = 1

for i in range(n):

yield power

power \*= 2

for v in powers\_of\_2(8):

print(v)

Consegue adivinhar o output? Copie o código para o editor e execute-o para verificar os seus palpites.

**Compreensões de lista**

Os geradores também podem ser utilizados no âmbito das **compreensões de lista**, tal como aqui:

def powers\_of\_2(n):

power = 1

for i in range(n):

yield power

power \*= 2

t = [x for x in powers\_of\_2(5)]

print(t)

Execute o exemplo e verifique o output.

**A list() .**

A função list() pode transformar uma série de invocações de geradores subsequentes numa **lista real**:

def powers\_of\_2(n):

power = 1

for i in range(n):

yield power

power \*= 2

t = list(powers\_of\_2(3))

print(t)

Mais uma vez, tente prever o output e execute o código para verificar as suas previsões.

**O operador in .**

Além disso, o contexto criado pelo operador in também permite a utilização de um gerador.

O exemplo mostra como o fazer:

def powers\_of\_2(n):

power = 1

for i in range(n):

yield power

power \*= 2

for i in range(20):

if i in powers\_of\_2(4):

print(i)

Qual é o output do código? Execute o programa e verifique.

**O gerador de números Fibonacci**

Agora vamos ver um **gerador de números Fibonacci**, e assegurar que tenha muito melhor aspeto do que a versão objetiva baseada na implementação do protocolo de iterador direto.

Aqui está:

def fibonacci(n):

p = pp = 1

for i in range(n):

if i in [0, 1]:

yield 1

else:

n = p + pp

pp, p = p, n

yield n

fibs = list(fibonacci(10))

print(fibs)

Adivinhe o output (uma lista) produzida pelo gerador, e execute o código para verificar se estava certo.

# Mais sobre compreensões de lista

Deverá ser capaz de se lembrar das regras que regem a criação e utilização de um fenómeno Python muito especial denominado **compreensão de listas - uma forma simples e muito impressionante de criar listas e o seu conteúdo**.

Caso precise, fornecemos um lembrete rápido no editor.

list\_1 = []

for ex in range(6):

list\_1.append(10 \*\* ex)

list\_2 = [10 \*\* ex for ex in range(6)]

print(list\_1)

print(list\_2)

Há duas partes dentro do código, ambas criando uma lista contendo algumas das primeiras potências naturais de dez.

A primeira utiliza uma forma rotineira de utilizar o loop for , enquanto a última faz uso da compreensão de lista e constrói a lista in situ, sem precisar de um loop, ou de qualquer outro código alargado.

Parece que a lista é criada dentro de si mesma - não é verdade, claro, pois o Python tem de realizar quase as mesmas operações que no primeiro snippet, mas é indiscutível que o segundo formalismo é simplesmente mais elegante, e permite ao leitor evitar quaisquer detalhes desnecessários.

O exemplo produz duas linhas idênticas contendo o seguinte texto:

[1, 10, 100, 1000, 10000, 100000]

[1, 10, 100, 1000, 10000, 100000]

**output**

Execute o código para verificar se estamos certos.

Há uma sintaxe muito interessante que queremos mostrar-lhe agora. A sua usabilidade não se limita a compreensões de lista, mas temos de admitir que as compreensões são o ambiente ideal para ela.

É uma **expressão condicional - uma forma de selecionar um de dois valores diferentes com base no resultado de uma expressão Booleana**.

Veja:

expression\_one if condition else expression\_two

Pode parecer um pouco surpreendente à primeira vista, mas é preciso ter em mente que **não se trata de uma instrução condicional**. Além disso, não é uma instrução de todo. É um operador.

O valor que ele fornece é igual a expression\_one quando a condição é True, e expression\_two caso contrário.

Um bom exemplo dir-lhe-á mais. Veja o código no editor.

the\_list = []

for x in range(10):

the\_list.append(1 if x % 2 == 0 else 0)

print(the\_list)

O código preenche uma lista com 1e 0- se o index de um determinado elemento for estranho, o elemento é definido para 0, e para 1 caso contrário.

Simples? Talvez não à primeira vista. Elegante? Indiscutivelmente.

Pode-se usar o mesmo truque dentro de uma compreensão de lista? Sim, pode-se.

Veja o exemplo no editor.

the\_list = [1 if x % 2 == 0 else 0 for x in range(10)]

print(the\_list)

Compactidão e elegância - estas duas palavras vêm à mente ao olhar para o código.

Então, o que é que têm em comum, geradores e compreensões de lista? Existe alguma conexão entre eles? Sim. Uma conexão bastante ligeira, mas inequívoca.

Apenas uma mudança pode **transformar qualquer compreensão de lista num gerador**.

**Compreensões de lista vs. geradores**

Agora olhe para o código abaixo e veja se consegue encontrar o detalhe que transforma uma compreensão de lista num gerador:

the\_list = [1 if x % 2 == 0 else 0 for x in range(10)]

the\_generator = (1 if x % 2 == 0 else 0 for x in range(10))

for v in the\_list:

print(v, end=" ")

print()

for v in the\_generator:

print(v, end=" ")

print()

São os **parêntesis**. Os parêntesis retos fazem uma compreensão, os parêntesis curvos fazem um gerador.

O código, no entanto, quando executado, produz duas linhas idênticas:

1 0 1 0 1 0 1 0 1 0

1 0 1 0 1 0 1 0 1 0

**output**

Como pode saber que a segunda tarefa cria um gerador, não uma lista?

Há algumas provas que lhe podemos mostrar. Aplicar a função len() para ambas as entidades.

len(the\_list) avaliará a 10. Claro e previsível. len(the\_generator) irá levantar uma exceção, e verá a seguinte mensagem:

TypeError: object of type 'generator' has no len()

**output**

Claro que não é necessário guardar a lista ou o gerador - pode criá-los exatamente no local onde precisa deles - tal como aqui:

for v in [1 if x % 2 == 0 else 0 for x in range(10)]:

print(v, end=" ")

print()

for v in (1 if x % 2 == 0 else 0 for x in range(10)):

print(v, end=" ")

print()

Nota: a mesma aparência do output não significa que ambos os loops funcionam da mesma forma. No primeiro loop, a lista é criada (e iterada através) como um todo - ela existe de facto quando o loop está a ser executado.

No segundo loop, não existe qualquer lista - existem apenas valores subsequentes produzidos pelo gerador, um por um.

Realize as suas próprias experiências.

**A função lambda .**

A função lambda é um conceito emprestado da matemática, mais especificamente, de uma parte chamada *cálculo Lambda*, mas estes dois fenómenos não são a mesma coisa.

Os matemáticos utilizam *o cálculo Lambda* em muitos sistemas formais ligados à lógica, recorrência, ou provabilidade do teorema. Os programadores usam a função lambda para simplificar o código, para torná-lo mais claro e mais fácil de entender.

Uma função lambda é uma função sem nome (também se pode chamar-lhe **uma função anónima**). Evidentemente, tal afirmação levanta imediatamente a questão: como utilizar qualquer coisa que não possa ser identificada?

Felizmente, não é um problema, pois pode nomear tal função se realmente precisar, mas, de facto, em muitos casos, a função lambda pode existir e trabalhar enquanto permanece totalmente incógnita.

A declaração da função lambda não se assemelha a uma declaração de função normal de forma alguma - veja por si mesmo:

lambda parameters: expression

Tal cláusula **devolve o valor da expressão quando se tem em conta o valor atual do atual argumento**lambda.

Como de costume, um exemplo será útil. O nosso exemplo usa três funções lambda , mas dá-lhes nomes. Olhe com atenção:

two = lambda: 2

sqr = lambda x: x \* x

pwr = lambda x, y: x \*\* y

for a in range(-2, 3):

print(sqr(a), end=" ")

print(pwr(a, two()))

Vamos analisá-lo:

* a primeira lambda é uma **função anónima sem parâmetros** que devolve sempre 2. Como **a atribuímos a uma variável chamada**two, podemos dizer que a função já não é anónima, e podemos usar o nome para a invocar.

* a segunda é uma **função anónima de um parâmetro** que devolve o valor do seu argumento ao quadrado. Também a nomeámos como tal.

* a terceira lambda **toma dois parâmetros** e devolve o valor do primeiro elevado à potência do segundo. O nome da variável que transporta a lambda fala por si. Não utilizamos pow para evitar confusão com a função integrada do mesmo nome e do mesmo objetivo.

O programa produz o seguinte output:

4 4

1 1

0 0

1 1

4 4

**output**

Este exemplo é suficientemente claro para mostrar como lambdasão declaradas e como se comportam, mas não diz nada sobre o porquê de serem necessárias, e para que são usadas, uma vez que todas elas podem ser substituídas por funções Python de rotina.

Onde está o benefício?

**Como usar lambdas e para quê?**

A parte mais interessante da utilização de lambdas surge quando se pode utilizá-los na sua forma pura - **como partes anónimas de código destinadas a avaliar um resultado**.

Imagine que precisamos de uma função (vamos nomeá-la print\_function) que imprime os valores de uma determinada (outra) função para um conjunto de argumentos selecionados.

Queremos que print\_function seja universal - deve aceitar um conjunto de argumentos colocados numa lista e uma função a ser avaliada, ambos como argumentos - não queremos codificar nada.

Veja o exemplo no editor. Foi assim que implementámos a ideia.

def print\_function(args, fun):

for x in args:

print('f(', x,')=', fun(x), sep='')

def poly(x):

return 2 \* x\*\*2 - 4 \* x + 2

print\_function([x for x in range(-2, 3)], poly)

Vamos analisá-la. A função print\_function() toma dois parâmetros:

* o primeiro, uma lista de argumentos para os quais queremos imprimir os resultados;
* o segundo, uma função que deve ser invocada tantas vezes quanto o número de valores que são recolhidos dentro do primeiro parâmetro.

Nota: também definimos uma função chamada poly() - esta é a função cujos valores vamos imprimir. O cálculo que a função realiza não é muito sofisticado - é o polinomial (daí o seu nome) de uma forma:

f(x) = 2x2 - 4x + 2

O nome da função é então passado para o print\_function() juntamente com um conjunto de cinco argumentos diferentes - o conjunto é construído com uma cláusula de compreensão de lista.

O código imprime as seguintes linhas:

f(-2)=18

f(-1)=8

f(0)=2

f(1)=0

f(2)=2

**output**

Podemos evitar definir a função poly() , já que não a vamos utilizar mais do que uma vez? Sim, podemos - este é o benefício que uma lambda pode trazer.

Veja o exemplo em baixo. Consegue ver a diferença?

def print\_function(args, fun):

for x in args:

print('f(', x,')=', fun(x), sep='')

print\_function([x for x in range(-2, 3)], lambda x: 2 \* x\*\*2 - 4 \* x + 2)

A função print\_function() permaneceu exatamente a mesma, mas não há nenhuma poly() função. Já não precisamos dela, pois o polinómio está agora diretamente dentro da invocação print\_function() sob a forma de um lambda definido da seguinte forma:

lambda x: 2 \* x\*\*2 - 4 \* x + 2

O código tornou-se mais curto, mais claro e mais legível.

Deixe-nos mostrar-lhe outro lugar onde os lambdas podem ser úteis. Começaremos com uma descrição de map(), uma função integrada de Python. O seu nome não é demasiado descritivo, a sua ideia é simples, e a função em si é realmente utilizável.

**Lambdas e a função map() .**

No mais simples de todos os casos possíveis, a função map() .

map(function, list)

toma dois argumentos:

* uma função;
* uma lista.

A descrição acima é extremamente simplificada, uma vez que:

* o segundo argumento map() pode ser qualquer entidade que possa ser iterada (por exemplo, um tuple, ou apenas um gerador)
* map() pode aceitar mais de dois argumentos.

A função map()**aplica a função passada pelo seu primeiro argumento a todos os elementos do seu segundo argumento, e devolve um iterador entregando todos os resultados subsequentes da função**.

Pode usar o iterador resultante num loop, ou convertê-lo numa lista usando a função list() .

Consegue ver aqui um papel para qualquer lambda?

Veja o código no editor - utilizámos dois lambdas no mesmo.

list\_1 = [x for x in range(5)]

list\_2 = list(map(lambda x: 2 \*\* x, list\_1))

print(list\_2)

for x in map(lambda x: x \* x, list\_2):

print(x, end=' ')

print()

Esta é a intriga:

* construa o list\_1 com valores de 0 até 4;
* em seguida, utilize map juntamente com o primeiro lambda para criar uma nova lista na qual todos os elementos tenham sido avaliados como 2 elevados à potência retirada do elemento correspondente de list\_1;
* list\_2 é então impresso;
* no passo seguinte, utilize a função map() novamente para fazer uso do gerador que devolve e para imprimir diretamente todos os valores que fornece; como pode ver, envolvemos o segundo lambda aqui - apenas eleva ao quadrado cada elemento de list\_2.

Tente imaginar o mesmo código sem lambdas. Seria melhor? É pouco provável.

# Lambdas e a função filter() .

Outra função Python que pode ser significativamente embelezada pela aplicação de um lambda é filter().

Espera o mesmo tipo de argumentos que map(), mas faz algo diferente - **filtra o seu segundo argumento ao mesmo tempo que é guiado por instruções que fluem da função especificada como o primeiro argumento** (a função é invocada para cada elemento da lista, tal como em map()).

Os elementos que retornam True da função **passam o filtro** - os outros são rejeitados.

O exemplo no editor mostra a função filter() em ação.

from random import seed, randint

seed()

data = [randint(-10,10) for x in range(5)]

filtered = list(filter(lambda x: x > 0 and x % 2 == 0, data))

print(data)

print(filtered)

Nota: fizemos uso do módulo random para inicializar o gerador de números aleatórios (não confundir com os geradores de que acabámos de falar) com a função seed() , e para produzir cinco valores inteiros aleatórios a partir de -10 até 10 utilizando a função randint() .

A lista é então filtrada, e apenas os números que são iguais e superiores a zero são aceites.

Claro que não é provável que receba os mesmos resultados, mas os nossos resultados são estes:

[6, 3, 3, 2, -7]

[6, 2]

**Um breve olhar sobre os closures**

Comecemos por uma definição: o **closure é uma técnica que permite o armazenamento de valores apesar do facto de o contexto em que foram criados já não existir**. Intrincado? Um pouco.

Vamos analisar um exemplo simples:

def outer(par):

loc = par

var = 1

outer(var)

print(var)

print(loc)

O exemplo é obviamente errado.

As duas últimas linhas causarão uma exceção NameError - nem par nem loc são acessíveis fora da função. Ambas as variáveis existem quando e só quando a função outer() está a ser executada.

Veja o exemplo no editor. Modificámos o código de forma significativa.

def outer(par):

loc = par

def inner():

return loc

return inner

var = 1

fun = outer(var)

print(fun())

Há um novo elemento no mesmo - uma função (denominada inner) dentro de outra função (nomeada outer).

Como é que funciona? Tal como qualquer outra função, exceto o facto de que inner() só pode ser invocada a partir do interior outer(). Podemos dizer que inner() é a ferramenta privada de outer()- nenhuma outra parte do código lhe pode aceder.

Veja com atenção:

* a função inner() devolve o valor da variável acessível dentro do seu scope, uma vez que inner() pode utilizar qualquer uma das entidades à disposição de outer()
* a função outer() devolve a função inner() em si; mais precisamente, devolve uma cópia da função inner() , a que estava congelada no momento de invocação outer(); a função congelada contém o seu ambiente completo, incluindo o estado de todas as variáveis locais, o que também significa que o valor de loc é retido com sucesso, embora outer() tenha deixado de existir há muito tempo.

Na verdade, o código é totalmente válido, e tem como output:

1

A função devolvida durante a invocação outer() é um **closure**.

**Um closure tem de ser invocado exatamente da mesma forma como foi declarado**.

No exemplo abaixo:

def outer(par):

loc = par

def inner():

return loc

return inner

var = 1

fun = outer(var)

print(fun())

a função inner() é sem parâmetros, pelo que temos de a invocar sem argumentos.

Agora veja o código no editor. É totalmente possível **declarar um closure equipado com um número arbitrário de parâmetros**, por exemplo, um, tal como a power() função.

def make\_closure(par):

loc = par

def power(p):

return p \*\* loc

return power

fsqr = make\_closure(2)

fcub = make\_closure(3)

for i in range(5):

print(i, fsqr(i), fcub(i))

Isto significa que o closure não só faz uso do ambiente congelado, como também pode **modificar o seu comportamento, utilizando valores retirados do exterior**.

Este exemplo mostra mais uma circunstância interessante - pode **criar tantos closures quantos quiser usando um e o mesmo código**. Isso é feito com uma função chamada make\_closure(). Nota:

* o primeiro closure obtido a partir de make\_closure() define uma ferramenta que enquadra o seu argumento;
* o segundo foi concebido para cubrir o argumento.

É por isso que o código produz o seguinte output:

0 0 0

1 1 1

2 4 8

3 9 27

4 16 64

Realize os seus próprios testes.

**Key takeaways**

1. Um **iterador** é um objeto de uma classe que fornece pelo menos **dois** métodos (sem contar com o construtor!):

* \_\_iter\_\_() é invocado uma vez quando o iterador é criado e devolve **o próprio** objeto do iterador;
* \_\_next\_\_() é invocado para fornecer **o valor da próxima iteração** e levanta uma exceção StopIteration quando a iteração **chega ao fim**.

2. O método yield só pode ser utilizada dentro de funções. A declaração yield suspende a execução da função e faz com que a função devolva o argumento do yield como resultado. Tal função não pode ser invocada de forma regular - a sua única finalidade é ser usada como um **gerador** (ou seja, num contexto que requer uma série de valores, como um loop for .)

3. Uma **expressão condicional** é uma expressão construída usando o operador if-else . Por exemplo:

print(True if 0 >=0 else False)

tem como output True.

4. Uma **compreensão de lista** torna-se um **gerador** quando usada dentro de **parêntesis curvos** (usada dentro de parêntesis retos, produz uma lista regular). Por exemplo:

for x in (el \* 2 for el in range(5)):

print(x)

tem como output 02468.

4. Uma **função lambda** é uma ferramenta para a criação de **funções anónimas**. Por exemplo:

def foo(x,f):

return f(x)

print(foo(9, lambda x: x \*\* 0.5))

tem como output 3.0.

5. O módulo map(fun, list) cria uma **cópia** de um argumento list , e aplica a função fun a todos os seus elementos, devolvendo um **gerador** que fornece o novo conteúdo da lista elemento por elemento. Por exemplo:

short\_list = ['mython', 'python', 'fell', 'on', 'the', 'floor']

new\_list = list(map(lambda s: s.title(), short\_list))

print(new\_list)

tem como output ['Mython', 'Python', 'Fell', 'On', 'The', 'Floor'].

6. A função filter(fun, list) cria uma **cópia** desses elementos list , que causam que a função fun devolva True. O resultado da função é um **gerador** que fornece o novo conteúdo da lista elemento por elemento. Por exemplo:

short\_list = [1, "Python", -1, "Monty"]

new\_list = list(filter(lambda s: isinstance(s, str), short\_list))

print(new\_list)

tem como output ['Python', 'Monty'].

7. O closure é uma técnica que permite o **armazenamento de valores** apesar do facto de o **contexto** em que foram criados **já não existir**. Por exemplo:

def tag(tg):

tg2 = tg

tg2 = tg[0] + '/' + tg[1:]

def inner(str):

return tg + str + tg2

return inner

b\_tag = tag('<b>')

print(b\_tag('Monty Python'))

tem como output <b>Monty Python</b>

**Exercício 1**

Qual é o output esperado do seguinte código?

class Vowels:

def \_\_init\_\_(self):

self.vow = "aeiouy " # Yes, we know that y is not always considered a vowel.

self.pos = 0

def \_\_iter\_\_(self):

return self

def \_\_next\_\_(self):

if self.pos == len(self.vow):

raise StopIteration

self.pos += 1

return self.vow[self.pos - 1]

vowels = Vowels()

for v in vowels:

print(v, end=' ')

Verifique

**Exercício 2**

Escreva uma função **lambda**, definindo a parte menos significativa do seu argumento inteiro, e aplique-a à função map() para produzir a string 1 3 3 5 na consola.

any\_list = [1, 2, 3, 4]

even\_list = # Complete the line here.

print(even\_list)

Verifique

**Exercício 3**

Qual é o output esperado do seguinte código?

def replace\_spaces(replacement='\*'):

def new\_replacement(text):

return text.replace(' ', replacement)

return new\_replacement

stars = replace\_spaces()

print(stars("And Now for Something Completely Different"))

Verifique

**Nota**

[PEP 8](https://www.python.org/dev/peps/pep-0008/#programming-recommendations), o Guia de Estilo do Código Python, recomenda que os **lambdas não sejam atribuídos a variáveis, mas sim que sejam definidos como funções**.

Isto significa que é melhor usar uma declaração def , e evitar utilizar uma declaração de atribuição que ligue uma expressão lambda a um identificador. Por exemplo:

# Recommended:

def f(x): return 3\*x

# Not recommended:

f = lambda x: 3\*x

A vinculação de lambdas a identificadores geralmente duplica a funcionalidade da declaração def . Utilizar declarações def , por outro lado, gera mais linhas de código.

É importante compreender que a realidade gosta frequentemente de desenhar os seus próprios cenários, que não seguem necessariamente as convenções ou recomendações formais. A decisão de os seguir ou não dependerá de muitas coisas: as suas preferências, outras convenções adotadas, diretrizes internas da empresa, compatibilidade com o código existente, etc. Esteja atento a isto.

# Aceder a ficheiros a partir de código Python

Um dos problemas mais comuns no trabalho do programador é **processar dados armazenados em ficheiros**, enquanto os ficheiros são normalmente armazenados fisicamente utilizando dispositivos de armazenamento - discos rígidos, ópticos, de rede ou solid-state.

É fácil imaginar um programa que ordena 20 números, e é igualmente fácil imaginar o utilizador deste programa a introduzir estes vinte números diretamente a partir do teclado.

É muito mais difícil imaginar a mesma tarefa quando há 20.000 números a serem ordenados, e não há um único utilizador que seja capaz de introduzir estes números sem cometer um erro.

É muito mais fácil imaginar que estes números são armazenados no ficheiro de disco que é lido pelo programa. O programa classifica os números e não os envia para o ecrã, mas em vez disso cria um novo ficheiro e guarda a sequência ordenada de números lá.

Se quisermos implementar um banco de dados simples, a única maneira de armazenar a informação entre execuções do programa é guardá-la num ficheiro (ou ficheiros se a sua base de dados for mais complexa).

Em princípio, qualquer problema de programação não simples depende do uso de ficheiros, quer processe imagens (armazenadas em ficheiros), multiplique matrizes (armazenadas em ficheiros) ou calcule salários e impostos (leitura de dados armazenados em ficheiros).



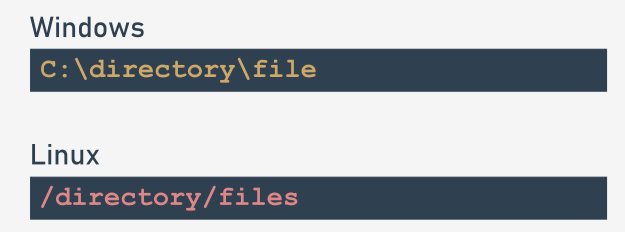
Pode perguntar-se porque esperámos até agora para lhe mostrar estas questões.

A resposta é muito simples - a forma do Python de aceder e processar ficheiros é implementada usando um conjunto consistente de objetos. Não há melhor momento para falar sobre o assunto.

# Nomes de ficheiro

Diferentes sistemas operativos podem tratar os ficheiros de diferentes formas. Por exemplo, o Windows utiliza uma convenção de nomes diferente da adotada nos sistemas Unix/Linux.

Se utilizarmos a noção de um nome de ficheiro canónico (um nome que define de forma única a localização do ficheiro independentemente do seu nível na árvore de diretoria) podemos perceber que estes nomes têm um aspeto diferente no Windows e no Unix/Linux:



Como pode ver, os sistemas derivados de Unix/Linux não utilizam a letra da unidade de disco (por exemplo, C:) e todas as diretorias crescem a partir de uma diretoria de raiz chamada /, enquanto os sistemas Windows reconhecem a diretoria de raiz como \.

Além disso, os nomes dos ficheiros do sistema Unix/Linux são sensíveis a maiúsculas e minúsculas. Os sistemas Windows armazenam a maiúscula ou minúscula das letras utilizadas no nome do ficheiro, mas não distinguem de todo se são de facto maiúsculas ou minúsculas.

Isto significa que estas duas strings: ThisIsTheNameOfTheFile e thisisthenameofthefile descrevem dois ficheiros diferentes em sistemas Unix/Linux, mas são o mesmo nome para apenas um ficheiro em sistemas Windows.

A principal e mais marcante diferença é que tem de usar **dois separadores diferentes para os nomes das diretorias**: \ no Windows, e / em Unix/Linux.

Esta diferença não é muito importante para o utilizador normal, mas é **muito importante quando se escreve programas em Python**.

Para entender o porquê, tente recordar o papel muito específico desempenhado pelo \ dentro de strings Python.

Suponha que está interessado num ficheiro específico localizado na diretoria dir, e nomeado file.

Suponha também que pretende atribuir uma string contendo o nome do ficheiro.

Em sistemas Unix/Linux, pode parecer-se com o seguinte:

name = "/dir/file"

Mas se o tentar codificar para o sistema Windows:

name = "\dir\file"

terá uma surpresa desagradável: ou o Python irá gerar um erro, ou a execução do programa irá comportar-se de forma estranha, como se o nome do ficheiro tivesse sido distorcido de alguma forma.

Na verdade, não é estranho, mas bastante óbvio e natural. O Python usa o \ como um caratere de escape (como \n).

Isto significa que os nomes dos ficheiros do Windows devem ser escritos da seguinte forma:

name = "\\dir\\file"

Felizmente, há também mais uma solução. O Python é suficientemente inteligente para ser capaz de converter barras em barras invertidas, cada vez que descobre que tal é exigido pelo sistema operativo.

Isto significa que quaisquer das seguintes tarefas:

name = "/dir/file"

name = "c:/dir/file"

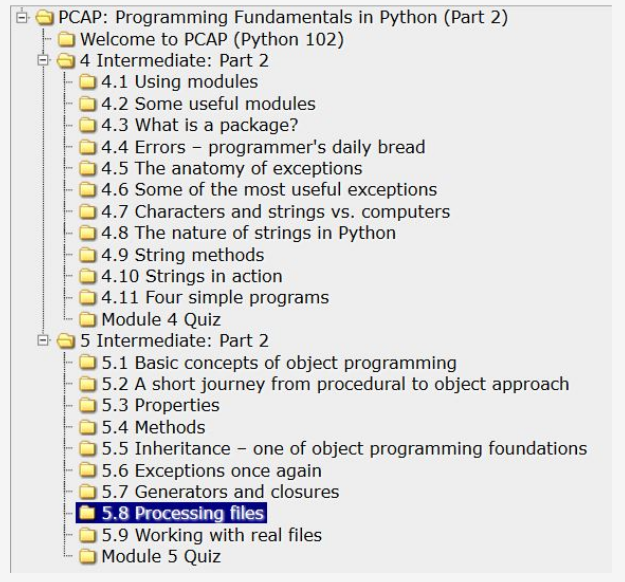
funcionará com o Windows, também.

Qualquer programa escrito em Python (e não só em Python, porque essa convenção se aplica a praticamente todas as linguagens de programação) não comunica diretamente com os ficheiros, mas através de algumas entidades abstratas que são nomeadas de forma diferente em diferentes linguagens ou ambientes - os termos mais utilizados são **handles** ou **streams** (vamos usá-los aqui como sinónimos).

O programador, tendo um conjunto de funções/métodos mais ou menos ricos, é capaz de realizar certas operações no stream, que afetam os ficheiros reais utilizando mecanismos contidos no kernel do sistema operativo.

Desta forma, é possível implementar o processo de acesso a qualquer ficheiro, mesmo quando o nome do ficheiro é desconhecido no momento da redação do programa.

As operações realizadas com o stream abstrato refletem as atividades relacionadas com o ficheiro físico.



Para conectar (vincular) o stream com o ficheiro, é necessário realizar uma operação explícita.

A operação de conectar o stream com um ficheiro chama-se **abrir o ficheiro**, enquanto que desconectar esta ligação chama-se **fechar o ficheiro**.

Assim, a conclusão é que a primeira operação realizada no stream é sempre open e a última é close. O programa, na verdade, é livre de manipular o stream entre estes dois eventos e para lidar com o ficheiro associado.

Esta liberdade é limitada, naturalmente, pelas características físicas do ficheiro e pela forma como o ficheiro foi aberto.

Digamos novamente que a abertura do stream pode falhar, e pode acontecer devido a várias razões: a mais comum é a falta de um ficheiro com um nome especificado.

Também pode acontecer que o ficheiro físico exista, mas o programa não está autorizado a abri-lo. Há também o risco de que o programa tenha aberto demasiados streams, e o sistema operacional específico pode não permitir a abertura simultânea de mais de n ficheiros (por exemplo, 200).

Um programa bem escrito deve detetar estas aberturas falhadas, e reagir em conformidade.

# Stream de ficheiros

A abertura do stream não está apenas associada ao ficheiro, mas deve também declarar a forma pela qual o stream será processado. Esta declaração é chamada **open mode** (modo aberto).

Se a abertura for bem sucedida, o **programa será autorizado a executar apenas as operações que sejam consistentes com o open mode declarado**.

Há duas operações básicas executadas no stream:

* **ler** a partir do stream: as porções dos dados são recuperadas do ficheiro e colocadas numa área da memória gerida pelo programa (por exemplo, uma variável);
* **escrever** para o stream: as porções dos dados da memória (por exemplo, uma variável) são transferidas para o ficheiro.

Existem três modos básicos usados para abrir o stream:

* **read mode** (modo de leitura): um stream aberto neste modo permite **apenas operações de leitura**; tentar escrever no stream causará uma exceção (nomeada UnsupportedOperation, que herda OSError e ValueError, e vem do io módulo);
* **write mode** (modo de escrita): um stream aberto neste modo permite **apenas operações de escrita**; tentar ler o stream causará a exceção mencionada acima;
* **update mode** (modo de atualização): um stream aberto neste modo permite **tanto escritas como leituras**.

Antes de discutirmos como manipular os streams, devemos-lhe algumas explicações. **O stream comporta-se quase como um gravador**.

Quando se lê algo de um stream, uma cabeça virtual move-se sobre o stream de acordo com o número de bytes transferidos do stream.

Quando se escreve algo no stream, a mesma cabeça move-se ao longo do stream gravando os dados a partir da memória.

Sempre que falarmos de ler e escrever para o stream, tente imaginar esta analogia. Os livros de programação referem-se a este mecanismo como a **current file position** (posição atual do ficheiro), e também utilizaremos este termo.



Agora é necessário mostrar-lhe o objeto responsável por representar streams em programas.

# Handles de ficheiro

O Python assume que **cada ficheiro está escondido atrás de um objeto de uma classe adequada**.

Claro, é difícil não perguntar como interpretar a palavra *adequada*.

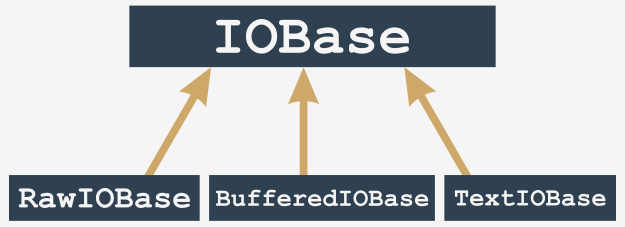
Os ficheiros podem ser processados de várias maneiras - alguns deles dependem do conteúdo do ficheiro, alguns das intenções do programador.

Em qualquer um dos casos, ficheiros diferentes podem exigir diferentes conjuntos de operações, e comportar-se de maneiras diferentes.

Um objeto de uma classe adequada é **criado quando se abre o ficheiro e aniquilado no momento do encerramento**.

Entre estes dois eventos, pode-se usar o objeto para especificar quais as operações que devem ser executadas num stream específico. As operações que tem permissão para usar são impostas pela **forma como abriu o ficheiro**.

Em geral, o objeto vem de uma das classes mostradas aqui:



Nota: nunca se usa construtores para dar vida a estes objetos. A única forma de os **obter é ao invocar a função chamada**open().

A função analisa os argumentos que forneceu e cria automaticamente o objeto necessário.

Se se quiser **livrar do objeto, invoca-se o método chamado**close().

A invocação irá cortar a ligação com o objeto e o ficheiro, e removerá o objeto.

Para os nossos propósitos, vamos preocupar-nos apenas com fluxos representados por objetos BufferIOBase e TextIOBase . Compreenderá porquê em breve.

Devido ao tipo de conteúdo do stream, **todos os streams estão divididos em streams de texto e streams binários**.

Os streams de texto são estruturados em linhas; ou seja, contêm carateres tipográficos (letras, dígitos, pontuação, etc.) dispostos em filas (linhas), como se vê a olho nu quando se olha para o conteúdo do ficheiro no editor.

Este ficheiro é escrito (ou lido) na sua maioria, caratere por caratere, ou linha por linha.

Os streams binários não contêm texto mas uma sequência de bytes de qualquer valor. Esta sequência pode ser, por exemplo, um programa executável, uma imagem, um áudio ou um videoclipe, um ficheiro de base de dados, etc.

Como estes ficheiros não contêm linhas, as leituras e escritas referem-se a porções de dados de qualquer tamanho. Assim, os dados são lidos/escritos byte a byte, ou bloco a bloco, onde o tamanho do bloco varia geralmente de um a um valor arbitrariamente escolhido.

Então aparece um problema subtil. Nos sistemas Unix/Linux, as extremidades das linhas são marcadas por um único caratere chamado LF (código ASCII 10) designado nos programas Python como \n.

Outros sistemas operativos, especialmente estes derivados do sistema pré-histórico CP/M (que também se aplica aos sistemas da família Windows) utilizam uma convenção diferente: o fim da linha é marcado por um par de carateres, CR e LF (códigos ASCII 13 e 10) que podem ser codificados como \r\n.



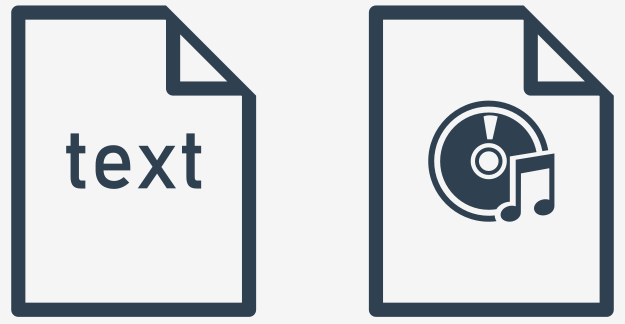
Esta ambiguidade pode causar várias consequências desagradáveis.

Se criar um programa responsável pelo processamento de um ficheiro de texto, e este for escrito para Windows, pode reconhecer as extremidades das linhas ao encontrar os carateres \r\n , mas o mesmo programa em execução num ambiente Unix/Linux será completamente inútil, e vice-versa: o programa escrito para sistemas Unix/Linux poderá ser inútil no Windows.

Estas características indesejáveis do programa, que impedem ou dificultam a utilização do programa em diferentes ambientes, são chamadas **não-portabilidade**.

Da mesma forma, a característica do programa que permite a execução em diferentes ambientes chama-se **portabilidade**. Um programa dotado de tal característica é chamado um **programa portátil**.

Uma vez que as questões de portabilidade eram (e ainda são) muito sérias, foi tomada a decisão de resolver definitivamente a questão de uma forma que não atrai a atenção do programador.



Foi feito ao nível das classes, que são responsáveis pela leitura e escrita dos carateres de e para o stream. Funciona da seguinte maneira:

* quando o stream está aberto e é avisado que os dados no ficheiro associado serão processados como texto (ou não existe tal aviso), é **mudada para o modo de texto**;

* durante a leitura/escrita de linhas de/para o ficheiro associado, nada de especial ocorre no ambiente Unix, mas quando as mesmas operações são realizadas no ambiente Windows, ocorre um processo chamado **tradução de carateres newline**: quando se lê uma linha do ficheiro, cada par de carateres \r\n é substituído por um único caratere \n , e vice-versa; durante as operações de escrita, cada caratere \n é substituído por um par de \r\n carateres;

* o mecanismo é completamente **transparente** para o programa, que pode ser escrito como se fosse destinado a processar apenas ficheiros de texto Unix/Linux; o source code executado num ambiente Windows também funcionará corretamente;

* quando o stream está aberto e é aconselhado a fazê-lo, o seu conteúdo é tomado tal como está, **sem qualquer conversão** - nenhum byte é adicionado ou omitido.

# Abertura dos streams

A **abertura do stream** é realizada por uma função que pode ser invocada da seguinte forma:

stream = open(file, mode = 'r', encoding = None)

Vamos analisá-la:

* o nome da função (open) fala por si; se a abertura for bem sucedida, a função devolve um objeto de stream; caso contrário, é levantada uma exceção (por exemplo, FileNotFoundError **se o ficheiro que vai ler não existir**);

* o primeiro parâmetro da função (file) especifica o nome do ficheiro a ser associado ao stream;

* o segundo parâmetro (mode) especifica o modo aberto utilizado para o stream; é uma string cheia de carateres, e cada um deles tem o seu significado especial (mais detalhes em breve);

* o terceiro parâmetro (encoding) especifica o tipo de codificação (por exemplo, UTF-8 quando se trabalha com ficheiros de texto)

* a abertura deve ser a primeira operação realizada no stream.

Nota: o modo e os argumentos de codificação podem ser omitidos - os seus valores por defeito são então assumidos. O modo de abertura padrão é a leitura em modo de texto, enquanto que a codificação padrão depende da plataforma utilizada.

Deixe-nos agora apresentar-lhe os modos abertos mais importantes e úteis. Preparado?

## Abrir os streams: modos

r modo aberto: read

* o stream será aberto em **read mode** (modo de leitura);
* o ficheiro associado ao stream **tem de existir** e tem de ser legível, caso contrário a função open() levanta uma exceção.

w modo aberto: write

* o stream será aberto em **write mode** (modo de escrita);
* o ficheiro associado ao stream **não precisa de existir**; se não existir, será criado; se existir, será truncado até ao comprimento de zero (apagado); se a criação não for possível (por exemplo, devido às permissões do sistema) a função open() levanta uma exceção.

a modo aberto: append

* o stream será aberto em **append mode** (modo anexo);
* o ficheiro associado ao stream **não precisa de existir**; se não existir, será criado; se existir, a cabeça de gravação virtual será colocada no fim do ficheiro (o conteúdo anterior do ficheiro permanece intocado).

r+ modo aberto: read and update

* o stream será aberto em **read and update mode** (modo de leitura e atualização);
* o ficheiro associado ao stream **tem de existir e tem de ser gravável**, caso contrário a função open() levanta uma exceção;
* tanto as operações de leitura como de escrita são permitidas para o stream.

w+ modo aberto: write and update

* o stream será aberto em **write and update mode** (modo de gravação e atualização);
* o ficheiro associado ao stream **não precisa de existir**; se não existir, será criado; o conteúdo anterior do ficheiro permanece intocado;
* tanto as operações de leitura como de escrita são permitidas para o stream.

## Seleção de texto e modos binários

Se houver uma letra b no final da string de modo significa que o stream deve ser aberto no **modo binário**.

Se a string de modo termina com uma letra t o stream é aberto no **modo de texto**.

O modo de texto é o comportamento por defeito assumido quando não é utilizado um especificador do modo binário/texto.

Finalmente, a abertura bem sucedida do ficheiro irá definir a posição atual do ficheiro (a cabeça de leitura/escrita virtual) antes do primeiro byte do ficheiro, **se o modo não for**a e após o último byte de ficheiro **se o modo estiver definido para**a.

| **Modo de texto** | **Modo binário** | **Descrição** |
| --- | --- | --- |
| rt | rb | read |
|  |  |  |
| wt | wb | write |
| at | ab | append |
| r+t | r+b | read and update |
| w+t | w+b | write and update |

EXTRA

Também se pode abrir um ficheiro para a sua criação exclusiva. Pode fazer isto usando o x modo aberto. Se o ficheiro já existir, a função open() irá levantar uma exceção.

## Abrir o stream pela primeira vez

Imagine que queremos desenvolver um programa que leia o conteúdo do ficheiro de texto nomeado: C:\Users\User\Desktop\file.txt.

Como abrir esse ficheiro para leitura? Aqui está o snippet relevante do código:

try:

stream = open("C:\Users\User\Desktop\file.txt", "rt")

# Processing goes here.

stream.close()

except Exception as exc:

print("Cannot open the file:", exc)

O que se passa aqui?

* abrimos o bloco try-except porque queremos lidar suavemente com os erros de runtime;
* utilizámos a função open() para tentar abrir o ficheiro especificado (note a forma como especificámos o nome do ficheiro)
* o modo aberto é definido como texto a ler (como **o texto é a definição padrão**, podemos saltar o t na string de modo)
* em caso de sucesso, obtemos um objeto da função open() e atribuímo-lo à variável de stream;
* Se open() falhar, tratamos da exceção imprimindo a informação de erro completa (é bom saber o que aconteceu exatamente)

## Streams pré-abertos

Dissemos anteriormente que qualquer operação de stream deve ser precedida pela open() invocação de função. Existem três exceções bem definidas à regra.

Quando o nosso programa começa, os três streams já estão abertos e não requerem quaisquer preparações extra. Além disso, o seu programa pode utilizar estes streams explicitamente se tiver o cuidado de importar o módulo sys :

import sys

porque é aí que a declaração dos três streams é colocada.

Os nomes desses streams são: sys.stdin, sys.stdout, e sys.stderr.

Vamos analisá-los:

* sys.stdin
  + stdin (de *standard input*)
  + o stream stdin é normalmente associado ao teclado, pré-aberto para leitura e considerado como a principal fonte de dados para os programas em execução;
  + a bem conhecida função input() lê dados de stdin por defeito.

* sys.stdout
  + stdout (de *standard output*)
  + o stream stdout é normalmente associado ao ecrã, pré-aberto para escrita, considerado como o alvo principal para o output de dados pelo programa em execução;
  + a bem conhecida função print() faz output dos dados para o stream stdout .

* sys.stderr
  + stderr (de *standard error output*)
  + o stream stderr é normalmente associado ao ecrã, pré-aberto para escrita, considerado como o local principal onde o programa em execução deve enviar informações sobre os erros encontrados durante o seu trabalho;
  + não apresentámos nenhum método para enviar os dados para este stream (fá-lo-emos em breve, prometemos)
  + a separação de stdout (resultados úteis produzidos pelo programa) a partir de stderr (mensagens de erro, inegavelmente úteis mas que não fornecem resultados) dá a possibilidade de redirecionar estes dois tipos de informação para os diferentes alvos. Uma discussão mais alargada sobre esta questão está para além do âmbito do nosso curso. O manual do sistema de funcionamento fornecerá mais informações sobre estas questões.

## Fechar streams

A última operação executada num stream (não inclui streams stdin, stdout, e stderr que não o exijam) deve estar **a fechar**.

Essa ação é executada por um método invocado de dentro do objeto de stream aberto: stream.close().

* o nome da função definitivamente comenta-se a si próprio: close()
* a função não espera exatamente nenhum argumento; o stream não precisa de ser aberto
* a função não devolve nada, mas levanta a exceção IOError em caso de erro;
* a maioria dos programadores acreditam que a função close() é sempre bem sucedida e, portanto, não há necessidade de verificar se cumpriu devidamente a sua tarefa.  
    
  Esta crença é apenas parcialmente justificada. Se o stream foi aberto para escrita e, em seguida, uma série de operações de escrita foram executadas, pode acontecer que os dados enviados para o stream ainda não tenham sido transferidos para o dispositivo físico (devido a um mecanismo chamado **caching** ou **buffering**).  
    
  Uma vez que o fecho do stream força os buffers a lhes fazer flush, pode ser que os flushes falhem e, portanto, a close() falha também.

Já mencionámos falhas causadas por funções que operam com streams, mas não mencionámos uma palavra sobre como podemos identificar exatamente a causa da falha.

A possibilidade de fazer um diagnóstico existe e é fornecida por um componente de exceção dos streams de que vamos falar agora.

## Diagnosticar problemas de stream

O objeto IOError está equipado com uma propriedade chamada errno (o nome vem da frase *error number*) e pode acedê-lo da seguinte forma:

try:

# Some stream operations.

except IOError as exc:

print(exc.errno)

O valor do atributo errno pode ser comparado com uma das constantes simbólicas predefinidas, definidas no módulo errno .

Vamos dar uma vista de olhos a algumas **constantes úteis para detetar erros de stream**:

* errno.EACCES → Permission denied  
    
  O erro ocorre quando tenta, por exemplo, abrir um ficheiro com o atributo *read only* para escrita.

* errno.EBADF → Bad file number  
    
  O erro ocorre quando tenta, por exemplo, operar com um stream não aberto.

* errno.EEXIST → File exists  
    
  O erro ocorre quando tenta, por exemplo, renomear um ficheiro com o seu nome anterior.

* errno.EFBIG → File too large  
    
  O erro ocorre quando tenta criar um ficheiro maior do que o máximo permitido pelo sistema operativo.

* errno.EISDIR → Is a directory  
    
  O erro ocorre quando tenta tratar um nome de diretoria como o nome de um ficheiro comum.

* errno.EMFILE → Too many open files  
    
  O erro ocorre quando tenta abrir simultaneamente mais streams do que os aceitáveis para o seu sistema operativo.

* errno.ENOENT → No such file or directory  
    
  O erro ocorre quando tenta aceder a um ficheiro/diretoria inexistente.

* errno.ENOSPC → No space left on device  
    
  O erro ocorre quando não há espaço livre na media.

A lista completa é muito mais longa (inclui também alguns códigos de erro não relacionados ao processamento de stream.)

Se você for programador muito cuidadoso, poderá sentir a necessidade de utilizar a sequência de afirmações semelhante às apresentadas no editor.

import errno

try:

s = open("c:/users/user/Desktop/file.txt", "rt")

# Actual processing goes here.

s.close()

except Exception as exc:

if exc.errno == errno.ENOENT:

print("The file doesn't exist.")

elif exc.errno == errno.EMFILE:

print("You've opened too many files.")

else:

print("The error number is:", exc.errno)

Felizmente, existe uma função que pode drasticamente **simplificar o código de tratamento de erros**.

O seu nome é strerror(), e vem do módulo os e **espera apenas um argumento - um número de erro**.

O seu papel é simples: dá-se um número de erro e obtém-se uma string descrevendo o significado do erro.

Nota: se passar um código de erro inexistente (um número que não está vinculado a nenhum erro real), a função irá levantar uma exceção ValueError .

Agora podemos simplificar o nosso código da seguinte forma:

from os import strerror

try:

s = open("c:/users/user/Desktop/file.txt", "rt")

# Actual processing goes here.

s.close()

except Exception as exc:

print("The file could not be opened:", strerror(exc.errno))

Okay. Agora é altura de lidar com ficheiros de texto e familiarizar-se com algumas técnicas básicas que pode utilizar para os processar.

**Key takeaways**

1. Um ficheiro tem de estar **aberto** antes de poder ser processado por um programa, e deve estar **fechado** quando o processamento estiver terminado.

A abertura do ficheiro associa-o ao **stream**, que é uma representação abstrata dos dados físicos armazenados na media. A forma como o stream é processado é chamada **modo aberto**. Existem **três** modos abertos:

* **modo de leitura** — apenas são permitidas operações de leitura;
* **modo de escrita** — apenas são permitidas operações de escrita;
* **modo de update** — tanto as escritas como as leituras são permitidas.

2. Dependendo do conteúdo do ficheiro físico, diferentes classes de Python podem ser utilizadas para processar ficheiros. Em geral, o BufferedIOBase é capaz de processar qualquer ficheiro, enquanto TextIOBase é uma classe especializada dedicada ao processamento de ficheiros de texto (ou seja, ficheiros contendo textos visíveis para humanos, divididos em linhas utilizando marcadores de nova linha). Assim, os streams podem ser divididos em **binários** e de **texto**.

3. A seguinte sintaxe da função open() é utilizada para abrir um ficheiro:  
  
open(file\_name, mode=open\_mode, encoding=text\_encoding)  
A invocação cria um objeto de fluxo e associa-o ao ficheiro com o nome file\_name, usando o especificado open\_mode e definindo o especificado text\_encoding, ou **levanta uma exceção no caso de um erro**.

4. Três streams **predefinidos** já estão abertos quando o programa é iniciado:

* sys.stdin — standard input;
* sys.stdout — standard output;
* sys.stderr — standard error output.

4. O objeto da exceção IOError , criado quando qualquer operação de ficheiro falha (incluindo operações abertas), contém uma propriedade chamada errno, que contém o código de conclusão da ação falhada. Use este valor para diagnosticar o problema.

**Exercício 1**

Como codificar o valor do argumento open() da função mode se for criar um novo ficheiro de texto para o preencher apenas com um artigo?

Verifique

**Exercício 2**

Qual é o significado do valor representado por errno.EACESS?

Verifique

**Exercício 3**

Qual é o output esperado do código seguinte, supondo que o ficheiro nomeado *file* não existe?

import errno

try:

stream = open("file", "rb")

print("exists")

stream.close()

except IOError as error:

if error.errno == errno.ENOENT:

print("absent")

else:

print("unknown")

# Processamento de ficheiros de texto

Nesta lição vamos preparar um ficheiro de texto simples com algum conteúdo curto e simples.

# Opening tzop.txt in read mode, returning it as a file object:

stream = open("tzop.txt", "rt", encoding = "utf-8")

print(stream.read()) # printing the content of the file

Vamos mostrar-lhe algumas técnicas básicas que pode utilizar para **ler o conteúdo do ficheiro**, a fim de o processar.

O processamento será muito simples - vai copiar o conteúdo do ficheiro para a consola, e contar todos os carateres que o programa tenha lido.

Mas lembre-se - a nossa compreensão de um ficheiro de texto é muito rigorosa. No nosso sentido, é um ficheiro de texto simples - pode conter apenas texto, sem quaisquer decorações adicionais (formatação, tipos de letra diferentes, etc.).

É por isso que deve evitar criar o ficheiro utilizando qualquer processador de texto avançado como MS Word, LibreOffice Writer, ou algo do género. Utilize os básicos que o seu sistema operativo oferece: Bloco de notas, vim, gedit, etc.

Se os seus ficheiros de texto contiverem alguns carateres nacionais não abrangidos pelo padrão de carateres ASCII, poderá necessitar de um passo adicional. A sua invocação de função open() pode exigir um argumento que denote uma codificação de texto específica.

Por exemplo, se estiver a usar um SO Unix/Linux configurado para usar o UTF-8 como uma configuração de todo o sistema, a função open() pode ter o seguinte aspeto:

stream = open('file.txt', 'rt', encoding='utf-8')

onde o argumento de codificação tem de ser definido para um valor que é uma string que representa a codificação de texto adequada (UTF-8, aqui).

Consulte a documentação do seu SO para encontrar um nome codificador adequado ao seu ambiente.

Nota

Para os fins das nossas experiências com processamento de ficheiros realizadas nesta secção, vamos utilizar um conjunto de ficheiros pré-carregados (ou seja, ficheiros tzop.txt, ou text.txt ) com os quais poderá trabalhar. Se desejar trabalhar com os seus próprios ficheiros localmente na sua máquina, encorajamo-lo vivamente a fazê-lo, e a utilizar o IDLE (ou qualquer outro IDE que possa preferir) para realizar os seus próprios testes.

A leitura do conteúdo de um ficheiro de texto pode ser feita usando vários métodos diferentes - nenhum deles é melhor ou pior do que qualquer outro. Depende de si qual deles prefere e gosta.

Alguns deles serão por vezes mais práticos, e por vezes mais problemáticos. Seja flexível. Não tenha medo de mudar as suas preferências.

O mais básico destes métodos é o que é oferecido pela função read() , que foi capaz de ver em ação na lição anterior.

Se aplicada a um ficheiro de texto, a função é capaz de:

* ler um número desejado de carateres (incluindo apenas um) do ficheiro, e devolvê-los como uma string;
* ler todo o conteúdo do ficheiro, e devolvê-lo como uma string;
* se não houver mais nada para ler (a cabeça de leitura virtual chega ao fim do ficheiro), a função devolve uma string vazia.

Começaremos com a variante mais simples e utilizaremos um ficheiro com o nome text.txt. O ficheiro tem o seguinte conteúdo:

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

**text.txt**

Vejamos agora o código no editor, e vamos analisá-lo.

from os import strerror

try:

cnt = 0

s = open('text.txt', "rt")

ch = s.read(1)

while ch != '':

print(ch, end='')

cnt += 1

ch = s.read(1)

s.close()

print("\n\nCharacters in file:", cnt)

except IOError as e:

print("I/O error occurred: ", strerror(e.errno))

A rotina é bastante simples:

* utilizar o mecanismo try-except e abrir o ficheiro do nome pré-determinado (text.txt no nosso caso)
* tentar ler o primeiro caratere do ficheiro (ch = s.read(1))
* se for bem sucedido (isto é comprovado por um resultado positivo da verificação do estado while ), fazer output do caratere (note o argumento end= - é importante! Não se quer saltar para uma nova linha depois de cada caratere!);
* atualizar o contador (cnt), também;
* tentar ler o próximo caratere, e o processo repete-se.

Se tem a certeza absoluta de que o comprimento do ficheiro é seguro e pode ler todo o ficheiro para a memória de uma só vez, pode fazê-lo - a função read() , invocada sem quaisquer argumentos ou com um argumento que avalia None, vai fazer o trabalho por si.

Lembre-se - **a leitura de um ficheiro de um terabyte usando este método pode corromper o seu sistema operativo**.

Não espere milagres - a memória do computador não é esticável.

Veja o código no editor.

from os import strerror

try:

cnt = 0

s = open('text.txt', "rt")

content = s.read()

for ch in content:

print(ch, end='')

cnt += 1

s.close()

print("\n\nCharacters in file:", cnt)

except IOError as e:

print("I/O error occurred: ", strerr(e.errno))

O que pensa disto?

Vamos analisá-lo:

* abra o ficheiro como anteriormente;
* leia o seu conteúdo através de uma invocação da função read() ;
* em seguida, processe o texto, iterando através dele com um loop for , e atualizando o valor do contador a cada turno do loop;

O resultado será exatamente o mesmo que anteriormente.

# Processamento de ficheiros de texto: readline()

Se quiser tratar o conteúdo do ficheiro **como um conjunto de linhas** e não como um monte de carateres, o método readline() irá ajudá-lo com isso.

O método tenta **ler uma linha completa de texto do ficheiro**, e devolve-a como uma string em caso de sucesso. Caso contrário, ele devolve uma string vazia.

Isto abre novas oportunidades - agora também se podem contar linhas facilmente, não apenas carateres.

Vamos fazer uso disso. Veja o código no editor.

from os import strerror

try:

ccnt = lcnt = 0

s = open('text.txt', 'rt')

line = s.readline()

while line != '':

lcnt += 1

for ch in line:

print(ch, end='')

ccnt += 1

line = s.readline()

s.close()

print("\n\nCharacters in file:", ccnt)

print("Lines in file: ", lcnt)

except IOError as e:

print("I/O error occurred:", strerror(e.errno))

Como se pode ver, a ideia geral é exatamente a mesma que nos dois exemplos anteriores.

# Processamento de ficheiros de texto: readlines()

Outro método, que trata o ficheiro de texto como um conjunto de linhas e não de carateres, é readlines().

A classe readlines() , quando invocado sem argumentos, tenta **ler todo o conteúdo do ficheiro, e devolve uma lista de strings, um elemento por linha de ficheiro**.

Se não tiver a certeza se o tamanho do ficheiro é suficientemente pequeno e não quiser testar o SO, pode convencer o método readlines() a ler não mais do que um número especificado de bytes ao mesmo tempo (o valor de retorno permanece o mesmo - é uma lista de uma string).

Sinta-se à vontade para experimentar o seguinte código de exemplo para compreender como o método readlines() funciona:

s = open("text.txt")

print(s.readlines(20))

print(s.readlines(20))

print(s.readlines(20))

print(s.readlines(20))

s.close()

**O tamanho máximo aceite do buffer de input é passado para o método como seu argumento**.

Pode esperar que readlines() pode processar o conteúdo de um ficheiro de forma mais eficaz do que readline(), uma vez que pode precisar de ser invocado menos vezes.

Nota: quando não há nada a ler do ficheiro, o método devolve uma lista vazia. Utilize-o para detetar o fim do ficheiro.

Na medida do tamanho do buffer, é de esperar que o seu aumento possa melhorar o desempenho de input, mas não há nenhuma regra de ouro para isso - tente você mesmo encontrar os valores ótimos.

from os import strerror

try:

ccnt = lcnt = 0

s = open('text.txt', 'rt')

lines = s.readlines(20)

while len(lines) != 0:

for line in lines:

lcnt += 1

for ch in line:

print(ch, end='')

ccnt += 1

lines = s.readlines(10)

s.close()

print("\n\nCharacters in file:", ccnt)

print("Lines in file: ", lcnt)

except IOError as e:

print("I/O error occurred:", strerror(e.errno))

Veja o código no editor. Modificámo-lo para lhe mostrar como usar readlines().

Decidimos usar um buffer de 15 bytes. Não pense que se trata de uma recomendação.

Utilizámos esse valor para evitar a situação em que a primeira invocação readlines() consome o ficheiro inteiro.

Queremos que o método seja forçado a trabalhar mais, e a demonstrar as suas capacidades.

Existem **dois nested loops no código**: o externo utiliza o resultado de readlines()para iterar através dele, enquanto que o interior imprime as linhas caratere a caratere.

O último exemplo que queremos apresentar mostra uma característica muito interessante do objeto devolvido pela função open() no modo de texto.

Achamos que pode surpreendê-lo - **o objeto é uma instância da classe iterável**.

Estranho? De modo algum. Utilizável? Sim, completamente.

O **protocolo de iteração definido para o objeto de ficheiro** é muito simples - o seu método \_\_next\_\_ apenas **devolve a próxima linha lida a partir do ficheiro**.

Além disso, pode esperar que o objeto invoque automaticamente close() quando qualquer uma das leituras do ficheiro chegar ao fim do ficheiro.

Olhe para o editor e veja como o código se tornou agora simples e claro.

from os import strerror

try:

ccnt = lcnt = 0

for line in open('text.txt', 'rt'):

lcnt += 1

for ch in line:

print(ch, end='')

ccnt += 1

print("\n\nCharacters in file:", ccnt)

print("Lines in file: ", lcnt)

except IOError as e:

print("I/O error occurred: ", strerror(e.errno))

# Lidar com ficheiros de texto: write()

Escrever ficheiros de texto parece ser mais simples, pois de facto existe um método que pode ser utilizado para realizar tal tarefa.

O método é chamado write() e espera apenas um argumento - uma string que será transferida para um ficheiro aberto (não esquecer - o modo aberto deve refletir a forma como os dados são transferidos - **escrever um ficheiro aberto em modo de leitura não será bem sucedido**).

Nenhum caratere newline é adicionado ao argumento write(), pelo que terá de ser você próprio a adicioná-lo se quiser que o ficheiro seja preenchido com um certo número de linhas.

O exemplo no editor mostra um código muito simples que cria um ficheiro com o nome newtext.txt (nota: o modo aberto w assegura que **o ficheiro será criado do zero**, mesmo que este exista e contenha dados) e depois coloca dez linhas no mesmo.

A string a ser gravada consiste na linha da palavra, seguida do número da linha. Decidimos escrever o conteúdo da string caratere a caratere (isto é feito pelo loop for interno) mas não é obrigado a fazê-lo desta forma.

Só queríamos mostrar-lhe que write() é capaz de operar em carateres individuais.

from os import strerror

try:

fo = open('newtext.txt', 'wt') # A new file (newtext.txt) is created.

for i in range(10):

s = "line #" + str(i+1) + "\n"

for ch in s:

fo.write(ch)

fo.close()

except IOError as e:

print("I/O error occurred: ", strerror(e.errno))

O código cria um ficheiro preenchido com o seguinte texto:

line #1

line #2

line #3

line #4

line #5

line #6

line #7

line #8

line #9

line #10

**output**

Pode imprimir o conteúdo do ficheiro para a consola?

Encorajamo-lo a testar o comportamento do método write() localmente na sua máquina.

Veja o exemplo no editor. Modificámos o código anterior para escrever linhas inteiras no ficheiro de texto.

from os import strerror

try:

fo = open('newtext.txt', 'wt')

for i in range(10):

fo.write("line #" + str(i+1) + "\n")

fo.close()

except IOError as e:

print("I/O error occurred: ", strerror(e.errno))

O conteúdo do ficheiro recentemente criado é o mesmo.

Nota: pode usar o mesmo método para escrever para o stream stderr , mas não tente abri-lo, pois está sempre aberto implicitamente.

Por exemplo, se quiser enviar uma string de mensagem para stderr para o distinguir do output normal de programa, pode ter este aspeto:

import sys

sys.stderr.write("Error message")

# O que é um bytearray?

Antes de começarmos a falar de ficheiros binários, temos de lhe falar de uma das **classes especializadas que o Python utiliza para armazenar dados amorfos**.

**Dados amorfos são dados que não têm formato ou forma específica** - são apenas uma série de bytes.

Isto não significa que estes bytes não possam ter o seu próprio significado, ou não possam representar qualquer objeto útil, por exemplo, gráficos bitmap.

O aspeto mais importante disto é que no local onde temos contacto com os dados, não somos capazes, ou simplesmente não queremos, saber nada sobre isso.

Os dados amorfos não podem ser armazenados utilizando qualquer um dos meios anteriormente apresentados - eles não são strings nem listas.

Deve haver um recipiente especial capaz de tratar de tais dados.

O Python tem mais do que um desses recipientes - um deles é um **bytearray de nome de classe especializado** - como o nome sugere, é **um array contendo bytes (amorfos)**.

Se quiser ter um tal recipiente, por exemplo, para ler uma imagem bitmap e processá-la de qualquer forma, precisa de o criar explicitamente, utilizando um dos construtores disponíveis.

Dê uma vista de olhos:

data = bytearray(10)

Tal invocação cria um objeto bytearray capaz de armazenar dez bytes.

Nota: tal construtor **preenche todo o array com zeros**.

Os bytearrays assemelham-se a listas em muitos aspetos. Por exemplo, são **mutáveis**, são um assunto da função len() , e pode aceder a qualquer um dos seus elementos utilizando a indexação convencional.

Existe uma limitação importante - **não deve definir nenhum elemento de byte array com um valor que não seja um número inteiro** (violar esta regra causará uma exceção TypeError ) e **não lhe é permitido atribuir um valor que não venha do intervalo 0 a 255 inclusive** (a menos que queira provocar uma ValueError exceção).

Pode **tratar qualquer elemento do byte array como valores inteiros** - tal como no exemplo do editor.

data = bytearray(10)

for i in range(len(data)):

data[i] = 10 - i

for b in data:

print(hex(b))

Nota: utilizámos dois métodos para iterar os byte arrays, e fizemos uso da função hex() para ver os elementos impressos como valores hexadecimais.

Agora vamos mostrar-lhe **como escrever um byte array num ficheiro binário** - binário, pois não queremos guardar a sua representação legível - queremos escrever uma cópia um-para-um do conteúdo da memória física, byte por byte.

Então, como escrevemos uma byte array para um ficheiro binário?

Veja o código no editor. Vamos analisá-lo:

from os import strerror

data = bytearray(10)

for i in range(len(data)):

data[i] = 10 + i

try:

bf = open('file.bin', 'wb')

bf.write(data)

bf.close()

except IOError as e:

print("I/O error occurred:", strerror(e.errno))

# Your code that reads bytes from the stream should go here.

* primeiro, inicializamos bytearray com valores subsequentes a partir de 10; se quiser que o conteúdo do ficheiro seja claramente legível, substitua 10 por algo como ord('a') - isto produzirá bytes contendo valores correspondentes à parte alfabética do código ASCII (não pense que fará do ficheiro um ficheiro de texto - ainda é binário, pois foi criado com uma bandeira wb );
* em seguida, criamos o ficheiro usando a função open() - a única diferença em comparação com as variantes anteriores é o modo aberto que contém a bandeira b ;
* o método write() toma o seu argumento (bytearray) e envia-o (como um todo) para o ficheiro;
* a stream é então fechada de forma rotineira.

O método write() devolve uma série de bytes escritos com sucesso.

Se os valores diferirem do comprimento dos argumentos do método, poderá anunciar alguns erros de escrita.

Neste caso, não fizemos uso do resultado - isto pode não ser apropriado em todos os casos.

Tente executar o código e analisar o conteúdo do ficheiro de output recém-criado.

Vai utilizá-lo na próxima etapa.

# Como ler bytes de uma stream

A leitura a partir de um ficheiro binário requer o uso de um nome de método especializado readinto(), uma vez que o método não cria um novo objeto de bytearray, mas preenche um previamente criado com os valores retirados do ficheiro binário.

Nota:

* o método devolve o número de bytes lidos com sucesso;
* o método tenta preencher todo o espaço disponível dentro do seu argumento; se houver mais dados no ficheiro do que espaço no argumento, a operação lida para antes do fim do ficheiro; caso contrário, o resultado do método pode indicar que a bytearray só foi preenchida de forma fragmentada (o resultado mostrá-lo-á, também, e a parte do array que não está a ser utilizada pelo conteúdo recentemente lido permanece intacta)

Veja o código completo abaixo:

from os import strerror

data = bytearray(10)

try:

bf = open('file.bin', 'rb')

bf.readinto(data)

bf.close()

for b in data:

print(hex(b), end=' ')

except IOError as e:

print("I/O error occurred:", strerror(e.errno))

Vamos analisá-lo:

* primeiro, abrimos o ficheiro (aquele que criou utilizando o código anterior) com o modo descrito como rb;
* em seguida, lemos seu conteúdo na bytearray chamada data, de tamanho dez bytes;
* finalmente, imprimimos o conteúdo da bytearray - são os mesmos que você esperava?

Execute o código e verifique se está a funcionar.

Uma forma alternativa de ler o conteúdo de um ficheiro binário é oferecida pelo método denominado read().

Invocado sem argumentos, tenta **ler todo o conteúdo do ficheiro na memória**, tornando-os parte de um objeto recentemente criado da classe bytes.

from os import strerror

data = bytearray(10)

for i in range(len(data)):

data[i] = 10 + i

try:

bf = open('file.bin', 'wb')

bf.write(data)

bf.close()

except IOError as e:

print("I/O error occurred:", strerror(e.errno))

# Your code that reads bytes from the stream should go here.

Esta classe tem algumas semelhanças com bytearray, com exceção de uma diferença significativa - é **imutável**.

Felizmente, não existem obstáculos à criação de uma byte array tirando o seu valor inicial diretamente do objeto de bytes, tal como aqui:

from os import strerror

try:

bf = open('file.bin', 'rb')

data = bytearray(bf.read())

bf.close()

for b in data:

print(hex(b), end=' ')

except IOError as e:

print("I/O error occurred:", strerror(e.errno))

Tenha cuidado - **não utilize este tipo de leitura se não tiver a certeza de que o conteúdo do ficheiro se adequa à memória disponível**.

Se o método read() é invocado com um argumento, **especifica o número máximo de bytes a serem lidos**.

O método tenta ler o número desejado de bytes do ficheiro, e o comprimento do objeto devolvido pode ser utilizado para determinar o número de bytes efetivamente lidos.

from os import strerror

data = bytearray(10)

for i in range(len(data)):

data[i] = 10 + i

try:

bf = open('file.bin', 'wb')

bf.write(data)

bf.close()

except IOError as e:

print("I/O error occurred:", strerror(e.errno))

# Your code that reads bytes from the stream should go here.

Pode usar o método tal como aqui:

try:

bf = open('file.bin', 'rb')

data = bytearray(bf.read(5))

bf.close()

for b in data:

print(hex(b), end=' ')

except IOError as e:

print("I/O error occurred:", strerror(e.errno))

Nota: os primeiros cinco bytes do ficheiro foram lidos pelo código - os cinco seguintes ainda estão à espera de ser processados.

# Cópia de ficheiros - uma ferramenta simples e funcional

Agora vai amalgamar todos estes novos conhecimentos, adicionar-lhe alguns elementos novos, e utilizá-los para escrever um código real que seja capaz de copiar efetivamente o conteúdo de um ficheiro.

Claro que o objetivo não é fazer um melhor substituto para comandos como *copy* (MS Windows) ou *cp* (Unix/Linux), mas ver uma forma possível de criar uma ferramenta de trabalho, mesmo que ninguém a queira utilizar.

Veja o código no editor.

from os import strerror

srcname = input("Enter the source file name: ")

try:

src = open(srcname, 'rb')

except IOError as e:

print("Cannot open the source file: ", strerror(e.errno))

exit(e.errno)

dstname = input("Enter the destination file name: ")

try:

dst = open(dstname, 'wb')

except Exception as e:

print("Cannot create the destination file: ", strerror(e.errno))

src.close()

exit(e.errno)

buffer = bytearray(65536)

total = 0

try:

readin = src.readinto(buffer)

while readin > 0:

written = dst.write(buffer[:readin])

total += written

readin = src.readinto(buffer)

except IOError as e:

print("Cannot create the destination file: ", strerror(e.errno))

exit(e.errno)

print(total,'byte(s) succesfully written')

src.close()

dst.close()

Vamos analisá-lo:

* linhas 3 a 8: pedir ao utilizador o nome do ficheiro a copiar, e tentar abri-lo para ler; terminar a execução do programa se a abertura falhar; nota: utilizar a função exit() para parar a execução do programa e para passar o código de conclusão para o SO; qualquer código de conclusão que não 0 diz que o programa encontrou alguns problemas; use o valor errno para especificar a natureza do problema;
* linhas 10 a 16: repetir quase a mesma ação, mas desta vez para o ficheiro de output;
* linha 18: preparar um pedaço de memória para transferir dados do source file para o ficheiro alvo; essa área de transferência é muitas vezes chamada de buffer, daí o nome da variável; o tamanho do buffer é arbitrário - neste caso, decidimos usar 64 kilobytes; tecnicamente, um buffer maior é mais rápido a copiar itens, pois um buffer maior significa menos operações I/O; na verdade, há sempre um limite, cujo cruzamento não traz mais melhorias; teste-o você mesmo se quiser.
* linha 19: contar os bytes copiados - este é o contador e o seu valor inicial;
* linha 21: tentar preencher o buffer pela primeira vez;
* linha 22: desde que se obtenha um número não nulo de bytes, repetir as mesmas ações;
* linha 23: escrever o conteúdo do buffer no ficheiro de output (nota: utilizámos uma slice para limitar o número de bytes a escrever, uma vez que write() prefere sempre escrever todo o buffer)
* linha 24: atualizar o contador;
* linha 25: ler o próximo pedaço de ficheiro;
* linhas 30 a 32: alguma limpeza final - o trabalho está feito.

# LAB Histograma de frequência de caracteres

Médio

## Objetivos

* melhorar as competências do aluno no funcionamento com ficheiros (leitura)
* utilizar coleções de dados para a contagem de dados numerosos.

## Cenário

Um ficheiro de texto contém algum texto (nada de anormal) mas precisamos de saber com que frequência (ou quão rara) cada letra aparece no texto. Tal análise pode ser útil na criptografia, por isso queremos ser capazes de o fazer em referência ao alfabeto latino.

A sua tarefa é escrever um programa que:

* pede ao utilizador o nome do ficheiro de input;
* lê o ficheiro (se possível) e conta todas as letras latinas (letras minúsculas e maiúsculas são tratadas como iguais)
* imprime um histograma simples em ordem alfabética (só devem ser apresentadas contagens não nulas)

Crie um ficheiro de teste para o código, e verifique se o seu histograma contém resultados válidos.

Assumindo que o ficheiro do teste contém apenas uma linha preenchida com:

aBc

**samplefile.txt**

o output esperado deve ter a seguinte aparência:

a -> 1

b -> 1

c -> 1

**output**

**Dica**: Achamos que um dicionário é um meio perfeito de recolha de dados para armazenar as contagens. As letras podem ser chaves enquanto os contadores podem ser valores.

## RESPOSTA

from os import strerror

# Initialize 26 counters for each Latin letter.

# Note: we've used comprehension to do that.

counters = {chr(ch): 0 for ch in range(ord('a'), ord('z') + 1)}

file\_name = input("Enter the name of the file to analyze: ")

try:

f = open(file\_name, "rt")

for line in f:

for char in line:

# If it is a letter...

if char.isalpha():

# ... we'll treat it as lower-case and update the appropriate counter.

counters[char.lower()] += 1

f.close()

# Let's output the counters.

for char in counters.keys():

cnt = counters[char]

if cnt > 0:

print(char, '->',cnt)

except IOError as e:

print("I/O error occurred: ", strerror(e.errno))

# LAB Histograma de frequência de caracteres ordenados

Médio

## Pré-requisitos

4.3.1.15

## Objetivos

* melhorar as capacidades do aluno em operar com ficheiros (leitura/escrita)
* utilizar lambdas para alterar a ordem de classificação.

## Cenário

O código anterior precisa de ser melhorado. Está tudo bem, mas tem de ser melhor.

A sua tarefa é fazer algumas alterações, que geram os seguintes resultados:

* o histograma de output será ordenado com base na frequência dos carateres (o contador maior deve ser apresentado primeiro)
* o histograma deve ser enviado para um ficheiro com o mesmo nome que o de input, mas com o sufixo '.hist' (deve ser concatenado com o nome original)

Assumindo que o ficheiro de input contém apenas uma linha preenchida:

cBabAa

**samplefile.txt**

o output esperado deve ter a seguinte aparência:

a -> 3

b -> 2

c -> 1

**output**

**Dica**: Use um lambda para alterar a ordem de classificação.

## RESPOSTA

from os import strerror

counters = {chr(ch): 0 for ch in range(ord('a'), ord('z') + 1)}

file\_name = input("Enter the name of the file to analyze: ")

try:

f = open(file\_name, "rt")

for line in f:

for char in line:

if char.isalpha():

counters[char.lower()] += 1

f.close()

f = open(file\_name + '.hist', 'wt')

# Note: we've used lambda to access the directory's elements and set reverse to get a valid order.

for char in sorted(counters.keys(), key=lambda x: counters[x], reverse=True):

cnt = counters[char]

if cnt > 0:

f.write(char + ' -> ' + str(cnt) + '\n')

f.close()

except IOError as e:

print("I/O error occurred: ", strerror(e.errno))

# LAB Avaliar os resultados dos estudantes

Médio

## Objetivos

* melhorar as competências do aluno em operar com ficheiros (leitura)
* aperfeiçoar as capacidades do aluno na definição e utilização de exceções e dicionários auto-definidos.

## Cenário

O Prof. Jekyll realiza aulas com os estudantes e faz regularmente anotações num ficheiro de texto. Cada linha do ficheiro contém três elementos: o primeiro nome do estudante, o último nome do estudante, e o número de pontos que o estudante recebeu durante certas aulas.

Os elementos são separados por espaços em branco. Cada estudante pode aparecer mais de uma vez no ficheiro do Prof. Jekyll.

O ficheiro pode ter a seguinte aparência:

John Smith 5

Anna Boleyn 4.5

John Smith 2

Anna Boleyn 11

Andrew Cox 1.5

**samplefile.txt**

A sua tarefa é escrever um programa que:

* pergunta ao utilizador o nome do ficheiro do Prof. Jekyll;
* lê o conteúdo do ficheiro e conta a soma dos pontos recebidos por cada estudante;
* imprime um relatório simples (mas ordenado), tal como este:

Andrew Cox 1.5

Anna Boleyn 15.5

John Smith 7.0

**output**

Nota:

* o seu programa deve ser totalmente protegido contra todas as falhas possíveis: a inexistência do ficheiro, o vazio do ficheiro, ou qualquer falha nos dados introduzidos; o encontro de qualquer erro de dados deve causar a terminação imediata do programa, e o erro deve ser apresentado ao utilizador;
* implemente e utilize a sua própria hierarquia de exceções - apresentámo-la no editor; a segunda exceção deve ser levantada quando uma linha má é detetada, e a terceira quando o source file existe mas está vazio.

**Dica**: Utilize um dicionário para armazenar os dados dos estudantes.

class StudentsDataException(Exception):

pass

class BadLine(StudentsDataException):

# Write your code here.

class FileEmpty(StudentsDataException):

# Write your code here.

## RESPOSTA

# A base exception class for our code:

class StudentsDataException(Exception):

pass

# An exception for erroneous lines:

class BadLine(StudentsDataException):

def \_\_init\_\_(self, line\_number, line\_string):

super().\_\_init\_\_(self)

self.line\_number = line\_number

self.line\_string = line\_string

# An exception for an empty file.

class FileEmpty(StudentsDataException):

def \_\_init\_\_(self):

super().\_\_init\_\_(self)

from os import strerror

# A dictionary for students' data:

data = { }

file\_name = input("Enter student's data filename: ")

line\_number = 1

try:

f = open(file\_name, "rt")

# Read the whole file into list.

lines = f.readlines()

f.close()

# Is the file empty?

if len(lines) == 0:

raise FileEmpty()

# Scan the file line by line.

for i in range(len(lines)):

# Get the i'th line.

line = lines[i]

# Divide it into columns.

columns = line.split()

# There shoule be 3 columns - are they there?

if len(columns) != 3:

raise BadLine(i + 1, line)

# Build a key from student's given name and surname.

student = columns[0] + ' ' + columns[1]

# Get points.

try:

points = float(columns[2])

except ValueError:

raise BadLine(i + 1, line)

# Update dictionary.

try:

data[student] += points

except KeyError:

data[student] = points

# Print results.

for student in sorted(data.keys()):

print(student,'\t', data[student])

except IOError as e:

print("I/O error occurred: ", strerror(e.errno))

except BadLine as e:

print("Bad line #" + str(e.line\_number) + " in source file:" + e.line\_string)

except FileEmpty:

print("Source file empty")

# Key takeaways

1. Para ler o conteúdo de um ficheiro, podem ser utilizados os seguintes métodos de stream:

* read(number) — lê os carateres/bytes number do ficheiro e devolve-os como uma string; é capaz de ler o ficheiro inteiro de uma só vez;
* readline() — lê uma única linha do ficheiro de texto;
* readlines(number) — lê as linhas number do ficheiro de texto; é capaz de ler todas as linhas ao mesmo tempo;
* readinto(bytearray) — lê os bytes do ficheiro e preenche o bytearray com eles;

2. Para escrever novos conteúdos num ficheiro, podem ser utilizados os seguintes métodos de stream:

* write(string) — escreve um string para um ficheiro de texto;
* write(bytearray) — escreve todos os bytes de bytearray para um ficheiro;

3. Os loops open() devolve um objeto iterável que pode ser utilizado para iterar através de todas as linhas do ficheiro dentro de um loop for . Por exemplo:

for line in open("file", "rt"):

print(line, end='')

O código copia o conteúdo do ficheiro para a consola, linha a linha. **Nota**: o stream fecha-se **automaticamente** quando chega ao fim do ficheiro.

**Exercício 1**

O que esperamos do método readlines() quando o stream está associado a um ficheiro vazio?

Verifique

**Exercício 2**

O que se pretende fazer com o seguinte código?

for line in open("file", "rt"):

for char in line:

if char.lower() not in "aeiouy ":

print(char, end='')

**Exercício 3**

Vai processar um bitmap armazenado num ficheiro com o nome image.png, e deseja ler o seu conteúdo como um todo numa variável de *bytearray* chamada image. Adicione uma linha ao seguinte código para atingir este objetivo.

try:

stream = open("image.png", "rb")

# Insert a line here.

stream.close()

except IOError:

print("failed")

else:

print("success")

**Introdução ao módulo os .**

Nesta secção, aprenderá sobre um módulo chamado *os*, que lhe permite **interagir com o sistema operativo utilizando o Python**.

Este fornece funções que estão disponíveis em sistemas Unix e/ou Windows. Se estiver familiarizado com a linha de comandos, verá que algumas funções dão os mesmos resultados que os comandos disponíveis nos sistemas operativos.

Um bom exemplo disto é a função mkdir , que permite criar uma diretoria tal como o comando *mkdir* em Unix e Windows. Se não conhecer este comando, não se preocupe.

Em breve terá a oportunidade de aprender as funções do módulo *os*, para executar operações em ficheiros e diretorias juntamente com os comandos correspondentes.

Além das operações de ficheiro e diretoria, o módulo *os* permite-lhe:

* obter informações sobre o sistema operativo;
* gerir processos;
* operar em streams I/O utilizando file descriptors (descritores de ficheiro).



Dentro de momentos, verá como obter informações básicas sobre o seu sistema operativo, embora a gestão de processos e o trabalho com os file descriptors não sejam discutidos aqui, visto serem tópicos mais avançados que requerem conhecimento dos mecanismos do sistema operativo.

Preparado?

**Obter informações sobre o sistema operativo**

Antes de criar a sua primeira estrutura de diretoria, verá como pode obter informações sobre o sistema operativo atual. Isto é realmente fácil porque o módulo *os* fornece uma função chamada *uname*, que devolve um objeto contendo os seguintes atributos:

* **systemname** — armazena o nome do sistema operativo;
* **nodename** — armazena o nome da máquina na network;
* **release** — armazena o lançamento do sistema operativo;
* **version** — armazena a versão do sistema operativo;
* **machine** — armazena o identificador de hardware, por exemplo, x86\_64.

Vejamos como é na prática:

import os

print(os.uname())

Resultado:

posix.uname\_result(sysname='Linux', nodename='192d19f04766', release='4.4.0-164-generic', version='#192-Ubuntu SMP Fri Sep 13 12:02:50 UTC 2019', machine='x86\_64')

**output**

Como pode ver, a função *uname* devolve um objeto contendo informações sobre o sistema operativo. O código acima foi lançado em Ubuntu 16.04.6 LTS, por isso não se surpreenda se obtiver um resultado diferente, porque depende do seu sistema operativo.

Infelizmente, a função *uname* só funciona em alguns sistemas Unix. Se utilizar o Windows, pode usar a função *uname* no módulo *platform*, que devolve um resultado semelhante.

O módulo *os* permite-lhe distinguir rapidamente o sistema operativo utilizando o atributo *name*, que suporta um dos seguintes nomes:

* **posix** — obterá este nome se utilizar Unix;
* **nt** — obterá este nome se utilizar o Windows;
* **java** — obterá este nome se o seu código estiver escrito em Jython.

Para o Ubuntu 16.04.6 LTS, o atributo *name* devolve o nome *posix*:

import os

print(os.name)

Resultado:

posix

**output**

**NOTA:** Em sistemas Unix, existe um comando chamado *uname* que devolve as mesmas informações (se o executar com a opção -a) que a função *uname*.

**Criar diretorias em Python**

O módulo *os* fornece uma função chamada *mkdir*, que, tal como o comando *mkdir* em Unix e Windows, permite criar uma diretoria. A função *mkdir* requer um caminho, que pode ser relativo ou absoluto. Recordemos como são ambos os caminhos na prática:

* **my\_first\_directory** — este é um caminho relativo que irá criar a diretoria *my\_first\_directory* na diretoria de trabalho atual;
* **./my\_first\_directory** — este é um caminho relativo que aponta explicitamente para a diretoria de trabalho atual. Tem o mesmo efeito que o caminho acima;
* **../my\_first\_directory** — este é um caminho relativo que criará a diretoria *my\_first\_directory* na diretoria pai da diretoria de trabalho atual;
* **/python/my\_first\_directory** — este é o caminho absoluto que irá criar a diretoria *my\_first\_directory*, que por sua vez está na diretoria *Python* na diretoria de raiz.

Veja o código no editor. Este mostra um exemplo de como criar a diretoria *my\_first\_directory* usando um caminho relativo. Esta é a variante mais simples do caminho relativo, que consiste em passar apenas o nome da diretoria.

Se testar o seu código aqui, ele produzirá a diretoria recém-criada ['my\_first\_directory'] (e todo o conteúdo do catálogo de trabalho atual).

A função *mkdir* cria uma diretoria no caminho especificado. Note que a execução do programa duas vezes levantará um *FileExistsError*.

Isto significa que não podemos criar uma diretoria se ela já existir. Para além do argumento path, a função *mkdir* pode opcionalmente tomar o argumento *mode*, que especifica as permissões de diretoria. No entanto, em alguns sistemas, o argumento *mode* é ignorado.

Para alterar as permissões de diretoria, recomendamos a função *chmod*, que funciona de forma semelhante ao comando *chmod* em sistemas Unix. Pode encontrar mais informações sobre o assunto na documentação.

No exemplo acima, outra função fornecida pelo módulo *os*, chamada *listdir*, é utilizada. A função *listdir* devolve uma lista contendo os nomes dos ficheiros e diretorias que se encontram no caminho passado como um argumento.

Se nenhum argumento lhe for passado, será utilizada a diretoria de trabalho atual (como no exemplo acima). É importante que o resultado da função *listdir* omita as entradas '.' e '..', que são exibidas, por exemplo, ao usar o comando *ls -a* em sistemas Unix.

**NOTA:** No Windows e no Unix, há um comando chamado *mkdir*, que requer uma diretoria path. O equivalente ao código acima que cria a diretoria *my\_first\_directory* é o comando *mkdir my\_first\_directory*.

**Criação de diretoria recursiva**

A função mkdir é muito útil, mas e se precisar de criar outra diretoria na diretoria que acabou de criar? Obviamente, pode ir à diretoria criada e criar outra diretoria dentro dela, mas felizmente o módulo *os* fornece uma função chamada makedirs, o que torna esta tarefa mais fácil.

A função *makedirs* permite a criação de diretoria recursiva, o que significa que todas as diretorias no caminho serão criadas. Vamos olhar para o código no editor e ver como ele é na prática.

import os

os.makedirs("my\_first\_directory/my\_second\_directory")

os.chdir("my\_first\_directory")

print(os.listdir())

O código deve produzir o seguinte resultado:

['my\_second\_directory']

**output**

O código cria duas diretorias. A primeira delas é criada na diretoria de trabalho atual, enquanto a segunda é criada na diretoria *my\_first\_directory*.

Não tem de ir à diretoria *my\_first\_directory* para criar a diretoria *my\_second\_directory*, porque a função *makedirs* faz isto por si. No exemplo acima, vamos à diretoria *my\_first\_directory* para mostrar que o comando *makedirs* cria a subdiretoria *my\_second\_directory*.

Para se mover entre diretorias, pode usar uma função chamada *chdir*, que altera a diretoria de trabalho atual para o caminho especificado. Como argumento, toma qualquer caminho relativo ou absoluto. No nosso exemplo, passámos-lhe o nome da primeira diretoria.

**NOTA:** O equivalente à função *makedirs* nos sistemas Unix é o comando *mkdir* com o sinalizador *-p*, enquanto no Windows é simplesmente o comando *mkdir* com o caminho:

* Sistemas semelhantes a Unix:  
    
  mkdir -p my\_first\_directory/my\_second\_directory

* Windows:  
    
  mkdir my\_first\_directory/my\_second\_directory

# Onde estou agora?

Já sabe como criar diretorias e como se mover entre elas. Pos vezes, quando se tem uma estrutura de diretorias muito grande por onde navegar, pode não se saber em que diretoria se está atualmente a trabalhar.



Como provavelmente adivinhou, o*módulo os fornece uma função que devolve informação sobre a diretoria de trabalho atual. É chamada getcwd. Olhe para o código no editor para ver como a utilizar na prática.*

*import os*

*os.makedirs("my\_first\_directory/my\_second\_directory")*

*os.chdir("my\_first\_directory")*

*print(os.getcwd())*

*os.chdir("my\_second\_directory")*

*print(os.getcwd())*

*Resultado:*

*.../my\_first\_directory*

*.../my\_first\_directory/my\_second\_directory*

***output***

*No exemplo, criamos a diretoria my\_first\_directory e a diretoria my\_second\_directory dentro desta. Na passo seguinte, alteramos a diretoria de trabalho atual para a diretoria my\_first\_directory, e depois exibimos a diretoria de trabalho atual (primeira linha do resultado).*

*Em seguida, vamos para a diretoria my\_second\_directory e exibimos novamente a diretoria de trabalho atual (segunda linha do resultado). Como pode ver, a função getcwd devolve o caminho absoluto para as diretorias.*

***NOTA:****Em sistemas semelhantes a Unix, o equivalente à função getcwd é o comando pwd, que imprime o nome da diretoria de trabalho atual.*

# Eliminar diretorias em Python

O módulo *os* também permite eliminar diretorias. Dá-lhe a opção de eliminar uma única diretoria ou uma diretoria com as suas subdiretorias. Para eliminar uma única diretoria, pode-se utilizar uma função chamada rmdir, que toma o caminho como seu argumento. Veja o código no editor.

import os

os.mkdir("my\_first\_directory")

print(os.listdir())

os.rmdir("my\_first\_directory")

print(os.listdir())

O exemplo acima é realmente simples. Primeiro, a diretoria *my\_first\_directory* é criada e, em seguida, é removida utilizando-se a função *rmdir*. A função *listdir* é usada como prova de que a diretoria foi removida com sucesso. Neste caso, devolve uma lista vazia. Ao eliminar uma diretoria, certifique-se de que esta existe e está vazio, caso contrário será levantada uma exceção.

Para remover uma diretoria e as suas subdiretorias, pode utilizar a função removedirs , o que requer que se especifique um caminho contendo todas as diretorias que devem ser removidos:

import os

os.makedirs("my\_first\_directory/my\_second\_directory")

os.removedirs("my\_first\_directory/my\_second\_directory")

print(os.listdir())

Tal como com a função *rmdir*, se uma das diretorias não existir ou não estiver vazia, será levantada uma exceção.

**NOTA:** No Windows e no Unix existe um comando chamado *rmdir*, que, tal como a função *rmdir*, remove diretorias. Além disso, ambos os sistemas têm comandos para eliminar uma diretoria e o seu conteúdo. No Unix, este é o comando *rm* com o sinalizador *-r*.

# A função system()

Todas as funções apresentadas nesta parte do curso podem ser substituídas por uma função chamada *system*, que executa um comando que lhe é passado como uma string.

A função system está disponível tanto no Windows como no Unix. Dependendo do sistema, ela devolve um resultado diferente.

No Windows, devolve o valor retornado pela shell após executar o comando dado, enquanto que no Unix devolve o estado de exit do processo.

Vamos olhar para o código no editor e ver como ele é na prática.

import os

returned\_value = os.system("mkdir my\_first\_directory")

print(returned\_value)

Resultado:

0

**output**

O exemplo acima funcionará tanto no Windows como no Unix. No nosso caso, recebemos o exit status 0, o que indica sucesso em sistemas Unix.

Isto significa que a diretoria *my\_first\_directory* foi criada. Como parte do exercício, tente listar o conteúdo da diretoria onde criou a diretoria *my\_first\_directory*.

# LAB O módulo OS

Fácil

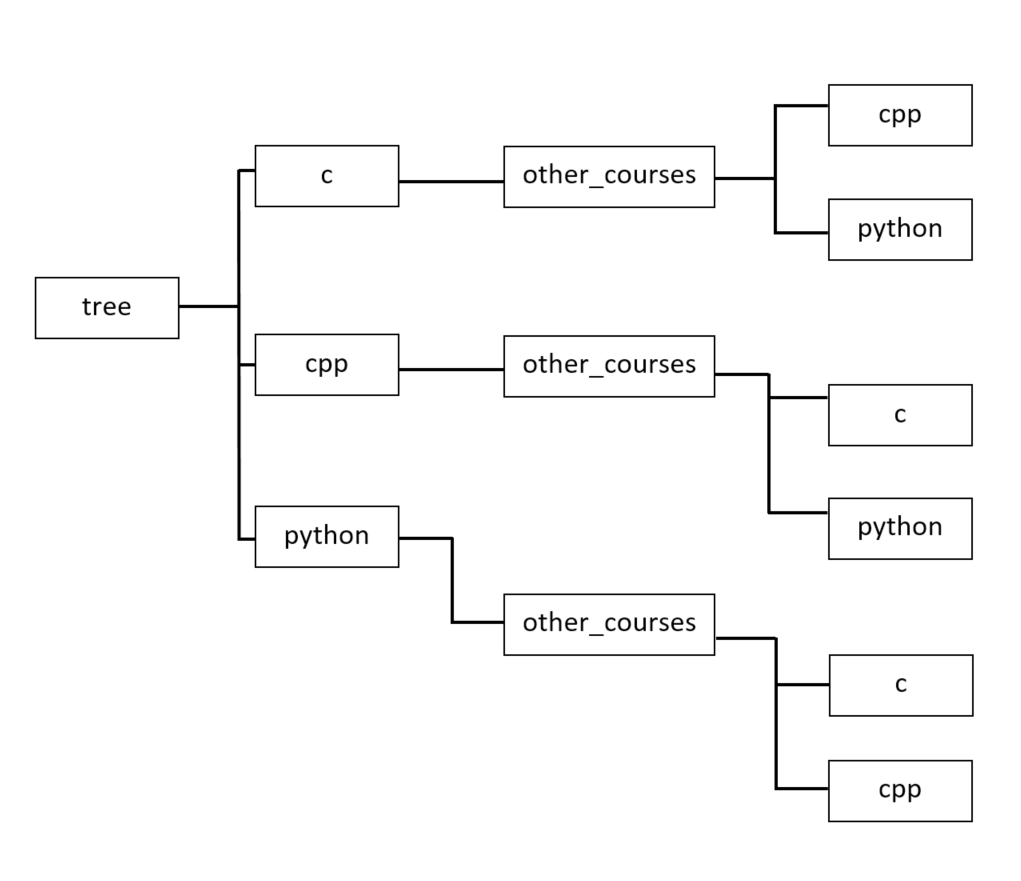
## Objetivos

* Melhorar as competências do aluno em interagir com o sistema operativo;
* uso prático de funções conhecidas fornecidas pelo módulo *os*.

## Cenário

Escusado será dizer que os sistemas operativos permitem-lhe procurar ficheiros e diretorias. Ao estudar esta parte do curso aprendeu sobre as funções do módulo *os*, que têm tudo o que precisa para escrever um programa que procurará diretorias num determinado local.

Para facilitar a sua tarefa, preparámos-lhe uma estrutura de diretoria de teste:



O seu programa deve cumprir os seguintes requisitos:

1. Escreve uma função ou método chamado *find* que toma dois argumentos chamados *path* e *dir*. O argumento *path* deve aceitar um caminho relativo ou absoluto para uma diretoria onde a pesquisa deve começar, enquanto o argumento *dir* deve ser o nome de uma diretoria que deseje encontrar no caminho dado. O seu programa deve exibir os caminhos absolutos se encontrar uma diretoria com o nome fornecido.
2. A pesquisa de diretoria deve ser feita recursivamente. Isto significa que a pesquisa também deve incluir todas as subdiretorias no caminho dado.

**Exemplo de input:**

path="./tree", dir="python"

**Exemplo de output:**

.../tree/python

.../tree/cpp/other\_courses/python

.../tree/c/other\_courses/python

## RESPOSTA

import os

class DirectorySearcher:

def find(self, path, dir):

try:

os.chdir(path)

except OSError:

# Doesn't process a file that isn't a directory.

return

current\_dir = os.getcwd()

for entry in os.listdir("."):

if entry == dir:

print(os.getcwd() + "/" + dir)

self.find(current\_dir + "/" + entry, dir)

directory\_searcher = DirectorySearcher()

directory\_searcher.find("./tree", "python")

**Key takeaways**

1. A função uname devolve um objeto que contém informação sobre o sistema operativo atual. O objeto tem os seguintes atributos:

* *systemname* (armazena o nome do sistema operativo)
* *nodename* (armazena o nome da máquina na rede)
* *release* (armazena a versão do sistema operativo)
* *version* (armazena a versão do sistema operativo)
* *machine* (armazena o identificador de hardware, por exemplo x86\_64.)

2. O atributo *name* disponível no módulo os permite distinguir o sistema operativo. Devolve um dos três valores seguintes:

* *posix* (receberá este nome se usar Unix)
* *nt* (receberá este nome se usar o Windows)
* *java* (receberá este nome se o seu código estiver escrito em algo como Jython)

3. Os loops mkdir cria uma diretoria no caminho percorrido como seu argumento. O caminho pode ser relativo ou absoluto, por exemplo

import os

os.mkdir("hello") # the relative path

os.mkdir("/home/python/hello") # the absolute path

**Nota**: Se a diretoria existir, uma exceção FileExistsError será lançada. Além da função mkdir , o módulo os fornece a função makedirs , que permite criar recursivamente todas as diretorias num caminho.

4. O resultado da função listdir() é uma lista contendo os nomes dos ficheiros e diretorias que se encontram no caminho percorrido como seu argumento.

É importante lembrar que a função listdir omite as entradas '.' e '..', que são exibidas, por exemplo, quando se utiliza o comando ls -a em sistemas Unix. Se o caminho não for passado, o resultado será devolvido para a diretoria de trabalho atual.

5. Para se deslocar entre diretorias, pode usar uma função chamada chdir(), que altera a diretoria de trabalho atual para o caminho especificado. Como seu argumento, toma qualquer caminho relativo ou absoluto.

Se quiser descobrir qual é a diretoria de trabalho atual, pode utilizar a função getcwd() , que lhe devolve o caminho.

6. Para remover uma diretoria, pode utilizar a função rmdir() , mas, para remover uma diretoria e as suas subdiretorias, utilize a função removedirs() .

7. Tanto no Unix como no Windows, pode usar a função do sistema, que executa um comando passado a ele como uma string, por exemplo:

import os

returned\_value = os.system("mkdir hello")

A função do sistema no Windows devolve o valor devolvido pela shell após executar o comando dado, enquanto que no Unix devolve o estado de saída do processo.

**Exercício 1**

Qual é o output do seguinte snippet se o executar no Unix?

import os

print(os.name)

**Exercício 2**

Qual é o output do seguinte snippet?

import os

os.mkdir("hello")

print(os.listdir())

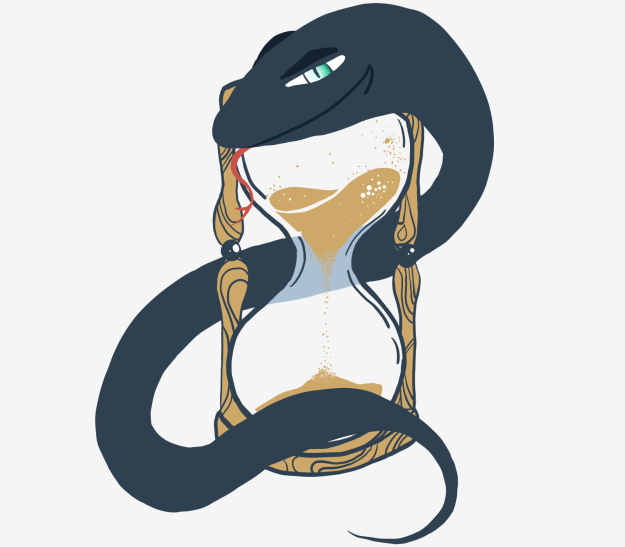
**Introdução ao módulo datetime .**

Nesta secção, aprenderá sobre um módulo Python chamado *datetime*.

Como pode adivinhar, proporciona **classes para trabalhar com data e hora**. Se pensa que não precisa de aprofundar este tópico, falemos de exemplos de utilização de data e hora na programação.

Data e hora têm inúmeras utilizações e é provavelmente difícil encontrar uma aplicação de produção que não as utilize. Aqui estão alguns exemplos:

* **registo de eventos** - graças ao conhecimento da data e hora, somos capazes de determinar quando ocorre exatamente um erro crítico na nossa aplicação. Ao criar registos, pode especificar o formato de data e hora;
* **reastreamento de alterações na base de dados** - por vezes é necessário armazenar informação sobre quando um registo foi criado ou modificado. O módulo *datetime* será perfeito para este caso;
* **validação de dados** - em breve aprenderá como ler a data e hora atuais em Python. Sabendo a data e hora atuais, podemos validar vários tipos de dados, por exemplo, se um cupão de desconto introduzido por um utilizador na nossa aplicação ainda é válido;
* **armazenar informação importante** - consegue imaginar transferências bancárias sem o armazenamento da informação de quando foram feitas? A data e a hora de certas ações devem ser preservadas, e nós temos de lidar com isso.



A data e hora são utilizadas em quase todas as áreas das nossas vidas, por isso é importante familiarizar-se com o módulo Python *datetime*. Está pronto para uma nova dose de conhecimento?

# Obter a data local atual e criar objetos de data

Uma das classes fornecidas pelo módulo datetime é uma classe chamada date. Os objetos desta classe representam uma data que consiste no ano, mês e dia. Veja o código no editor para ver como é na prática e obter a data local atual usando o today método.

Execute o código para ver o que acontece.

from datetime import date

today = date.today()

print("Today:", today)

print("Year:", today.year)

print("Month:", today.month)

print("Day:", today.day)

O método today devolve um objeto date que representa a data local atual. Note-se que o objeto date tem três atributos: *ano*, *mês* e *dia*.

Tenha cuidado, pois estes atributos são apenas de leitura. Para criar um objeto date , deve passar os parâmetros de *ano*, *mês* e *dia* como se segue:

from datetime import date

my\_date = date(2019, 11, 4)

print(my\_date)

Execute o exemplo para ver o que acontece.

Ao criar um objeto de *data*, tenha em mente as seguintes restrições:

|  |  |
| --- | --- |
| **Parâmetro** | **Restrições** |
| year | O parâmetro *ano* deve ser maior ou igual a 1 (constante MINYEAR) e menor ou igual a 9999 (constante MAXYEAR). |
| month | O parâmetro *mês* deve ser maior ou igual a 1 e menor ou igual a 12. |
| day | O parâmetro *dia* deve ser maior ou igual a 1 e menor ou igual ao último dia do mês e ano em questão. |

**Nota**: Mais tarde neste curso aprenderá a alterar o formato padrão da data.

# Criar um objeto de data a partir de um timestamp

A classe date dá-nos a capacidade de criar um objeto de *data* a partir de um *timestamp*.

Em Unix, o timestamp expressa o número de segundos desde 1 de janeiro de 1970, 00:00:00 (UTC). Esta data é chamada a **época Unix**, porque foi quando a contagem do tempo começou nos sistemas Unix.

O timestamp é na realidade a diferença entre uma determinada data (incluindo a hora) e 1 de janeiro de 1970, 00:00:00 (UTC), expressa em segundos.

Para criar um objeto de data a partir de um timestamp, temos de passar um timestamp Unix para o método fromtimestamp .

Para este fim, podemos usar o módulo time , que fornece funções relacionadas ao tempo. Uma delas é uma função chamada time() que devolve o número de segundos desde 1 de janeiro de 1970 até ao momento atual sob a forma de um número float. Dê uma vista de olhos no exemplo no editor.

Execute o código para ver o output.

from datetime import date

import time

timestamp = time.time()

print("Timestamp:", timestamp)

d = date.fromtimestamp(timestamp)

print("Date:", d)

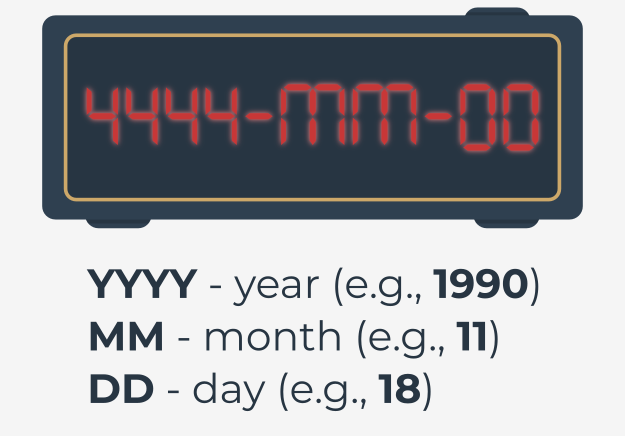
Se executar o código de amostra várias vezes, poderá ver como o timestamp se incrementa a si próprio. Vale a pena acrescentar que o resultado da função time depende da plataforma, porque **nos sistemas Unix e Windows, os segundos bissextos não são contados**.

**Nota**: Nesta parte do curso falaremos também sobre o módulo *time*.

# Criação de um objecto de data utilizando o formato ISO

O módulo datetime fornece vários métodos para criar um objeto date . Um deles é o método fromisoformat , que tem uma data no formato **YYYY-MM-DD** em conformidade com a norma ISO 8601.

A norma ISO 8601 define a forma como a data e a hora são representadas. É muitas vezes utilizada, pelo que vale a pena tomar um momento para se familiarizar com ela. Veja a imagem descrevendo os valores exigidos pelo formato:



Agora olhe para o código no editor e execute-o.

from datetime import date

d = date.fromisoformat('2019-11-04')

print(d)

No nosso exemplo, YYYY é 2019, MM é 11 (novembro) e DD é 04 (quarto dia de novembro).

Ao substituir a data, certifique-se de acrescentar 0 antes de um mês ou de um dia que seja expresso por um número inferior a 10.

**Note:** O método fromisoformat está disponível em Python desde a versão 3.7.

# Os loops replace() .

Por vezes pode ser necessário substituir o ano, mês, ou dia por um valor diferente. Não se pode fazer isto com os atributos de ano, mês e dia porque são apenas de leitura. Neste caso, pode utilizar o método denominado replace.

Execute o código no editor.

from datetime import date

d = date(1991, 2, 5)

print(d)

d = d.replace(year=1992, month=1, day=16)

print(d)

Resultado:

1991-02-05

1992-01-16

**output**

Os parâmetros *ano*, *mês* e *dia* são opcionais. Pode passar apenas um parâmetro para o método replace , por exemplo, *ano*, ou todos os três como no exemplo.

O método replace devolve um objeto com *data* alterada, pelo que se deve lembrar de o atribuir a alguma variável.

# Que dia da semana é hoje?

Um dos métodos mais úteis que facilita o trabalho com datas é o método chamado weekday. Ele devolve o dia da semana como um número inteiro, onde 0 é segunda-feira e 6 é domingo. Execute o código no editor.

from datetime import date

d = date(2019, 11, 4)

print(d.weekday())

Resultado:

0

**output**



A classe date tem um método semelhante chamado isoweekday, que também devolve o dia da semana como um inteiro, mas 1 é segunda-feira, e 7 é domingo:

from datetime import date

d = date(2019, 11, 4)

print(d.isoweekday())

Resultado:

1

**output**

Como pode ver, para a mesma data recebemos um número inteiro diferente, mas expressando o mesmo dia da semana. O número inteiro devolvido pelo método isodayweek segue a especificação ISO 85601.

# Criar objetos time .

Já sabe como apresentar uma data utilizando o objeto date . O módulo datetime também tem uma classe que lhe permite apresentar o tempo. Consegue adivinhar o seu nome? Sim, é chamada time:

time(hour, minute, second, microsecond, *tzinfo, fold*)

A função time aceita os seguintes parâmetros opcionais:

|  |  |
| --- | --- |
| **Parâmetro** | **Restrições** |
| hour | O parâmetro *hora* deve ser maior ou igual a 0 e inferior a 23. |
| minute | O parâmetro *minuto* deve ser maior ou igual a 0 e inferior a 59. |
| second | O parâmetro *segundo* deve ser maior ou igual a 0 e inferior a 59. |
| microsecond | O parâmetro *microssegundo* deve ser maior ou igual a 0 e inferior a 1000000. |
| tzinfo | O parâmetro *tzinfo* deve ser um objeto de subclasse tzinfo ou None (default). |
| fold | O parâmetro *fold* deve ser 0 ou 1 (padrão 0). |

O parâmetro *tzinfo* está associado a fusos horários, enquanto o *fold* está associado a tempo de relógio. Não os utilizaremos durante este curso, mas encorajamo-lo a familiarizar-se com eles.

Vejamos como criar um objeto time na prática. Execute o código no editor.

from datetime import time

t = time(14, 53, 20, 1)

print("Time:", t)

print("Hour:", t.hour)

print("Minute:", t.minute)

print("Second:", t.second)

print("Microsecond:", t.microsecond)

Resultado:

Time: 14:53:20.000001

Hour: 14

Minute: 53

Second: 20

Microsecond: 1

**output**

No exemplo, passámos quatro parâmetros para o construtor de classe: *hora*, *minuto*, *segundo* e *microssegundo*. Cada um deles pode ser acedido utilizando os atributos de classe.

**Nota**: Em breve lhe diremos como pode alterar a formatação temporal padrão.

# Os loops time .

Para além dos módulos time , a biblioteca padrão de Python oferece um módulo chamado time, que fornece uma função relacionada com o tempo. Já teve a oportunidade de aprender a função chamada time ao discutir a classe date . Agora vamos olhar para outra função útil disponível neste módulo.

Deve passar muitas horas em frente de um computador enquanto faz este curso. Por vezes pode sentir a necessidade de fazer uma sesta. Porque não? Vamos escrever um programa que simula a curta sesta de um estudante. Dê uma vista de olhos ao código no editor.

import time

class Student:

def take\_nap(self, seconds):

print("I'm very tired. I have to take a nap. See you later.")

time.sleep(seconds)

print("I slept well! I feel great!")

student = Student()

student.take\_nap(5)

Resultado:

I'm very tired. I have to take a nap. See you later.

I slept well! I feel great!

**output**

A parte mais importante do código de exemplo é o uso da função sleep (sim, pode recordá-la de um dos labs anteriores do curso), o que suspende a execução do programa durante um determinado número de segundos.

No nosso exemplo são 5 segundos. Tem razão, é uma sesta muito curta.

Prolongue o sesta do aluno alterando o número de segundos. Note que a função sleep aceita apenas um número inteiro ou um número floating-point.

# Os loops ctime() .

A função time fornece uma função chamada ctime, que **converte o tempo em segundos desde 1 de janeiro de 1970 (época Unix) para uma string**.

Lembra-se do resultado da função time ? É isso que precisa de passar para ctime. Dê uma vista de olhos no exemplo no editor.

import time

timestamp = 1572879180

print(time.ctime(timestamp))

Resultado:

Mon Nov 4 14:53:00 2019

**output**

A função ctime devolve uma string para o timestamp passado. No nosso exemplo, o timestamp expressa 4 de novembro de 2019 às 14:53:00.

Também é possível chamar a função ctime sem especificar o tempo em segundos. Neste caso, a hora atual será devolvida:

import time

print(time.ctime())

# Os loops gmtime() e localtime() .

Algumas das funções disponíveis no módulo time requerem conhecimento da classe *struct\_time*, mas antes de os conhecermos, vamos ver como é a classe:

time.struct\_time:

tm\_year # specifies the year

tm\_mon # specifies the month (value from 1 to 12)

tm\_mday # specifies the day of the month (value from 1 to 31)

tm\_hour # specifies the hour (value from 0 to 23)

tm\_min # specifies the minute (value from 0 to 59)

tm\_sec # specifies the second (value from 0 to 61 )

tm\_wday # specifies the weekday (value from 0 to 6)

tm\_yday # specifies the year day (value from 1 to 366)

tm\_isdst # specifies whether daylight saving time applies (1 – yes, 0 – no, -1 – it isn't known)

tm\_zone # specifies the timezone name (value in an abbreviated form)

tm\_gmtoff # specifies the offset east of UTC (value in seconds)

A classe *struct\_time* também permite o acesso a valores utilizando indexes. Index 0 devolve o valor em *tm\_year*, enquanto 8 devolve o valor em *tm\_isdst*.

As exceções são *tm\_zone* e *tm\_gmoff*, que não podem ser acedidas utilizando indexes. Vejamos como utilizar na prática a classe *struct\_time*. Execute o código no editor.

import time

timestamp = 1572879180

print(time.gmtime(timestamp))

print(time.localtime(timestamp))

Resultado:

time.struct\_time(tm\_year=2019, tm\_mon=11, tm\_mday=4, tm\_hour=14, tm\_min=53, tm\_sec=0, tm\_wday=0, tm\_yday=308, tm\_isdst=0)

time.struct\_time(tm\_year=2019, tm\_mon=11, tm\_mday=4, tm\_hour=14, tm\_min=53, tm\_sec=0, tm\_wday=0, tm\_yday=308, tm\_isdst=0)

**output**

O exemplo mostra duas funções que convertem o tempo decorrido da época Unix para o objeto *struct\_time*. A diferença entre eles é que a função gmtime devolve o objeto *struct\_time* em UTC, enquanto a função localtime devolve a hora local. Para a função gmtime , o atributo *tm\_isdst* é sempre 0.

# Os loops asctime() e mktime() .

O módulo time tem funções que esperam um objeto *struct\_time* ou um tuple que armazena valores de acordo com os indexes apresentados quando se discute a classe *struct\_time*. Execute o exemplo no editor.

import time

timestamp = 1572879180

st = time.gmtime(timestamp)

print(time.asctime(st))

print(time.mktime((2019, 11, 4, 14, 53, 0, 0, 308, 0)))

Resultado:

Mon Nov 4 14:53:00 2019

1572879180.0

**output**

A primeira das funções, chamada asctime, converte um objeto *struct\_time* ou um tuple numa string. Note-se que a função familiar gmtime é usada para obter o objeto *struct\_time*. Se não fornecer um argumento à função asctime , o tempo devolvido pela função localtime será usado.

A segunda função chamada mktime converte um objeto *struct\_time* ou um tuple que expressa a hora local para o número de segundos desde a época Unix. No nosso exemplo, passamos-lhe um tuple, que consiste nos seguintes valores:

2019 => tm\_year  
11 => tm\_mon  
4 => tm\_mday  
14 => tm\_hour  
53 => tm\_min  
0 => tm\_sec  
0 => tm\_wday  
308 => tm\_yday  
0 => tm\_isdst

# Criar objetos datetime .

No módulo datetime , data e hora podem ser representados como objetos separados ou como um só. A classe que combina data e hora é chamada datetime.

datetime(year, month, day, hour, minute, second, microsecond, *tzinfo, fold*)

O seu construtor aceita os seguintes parâmetros:

|  |  |
| --- | --- |
| **Parâmetro** | **Restrições** |
| year | O parâmetro *ano* deve ser maior ou igual a 1 (constante MINYEAR) e menor ou igual a 9999 (constante MAXYEAR). |
| month | O parâmetro *mês* deve ser maior ou igual a 1 e menor ou igual a 12. |
| day | O parâmetro *dia* deve ser maior ou igual a 1 e menor ou igual ao último dia do mês e ano em questão. |
| hour | O parâmetro *hora* deve ser maior ou igual a 0 e inferior a 23. |
| minute | O parâmetro *minuto* deve ser maior ou igual a 0 e inferior a 59. |
| second | O parâmetro *segundo* deve ser maior ou igual a 0 e inferior a 59. |
| microsecond | O parâmetro *microssegundo* deve ser maior ou igual a 0 e inferior a 1000000. |
| tzinfo | O parâmetro *tzinfo* deve ser um objeto de subclasse tzinfo ou None (default). |
| fold | O parâmetro *fold* deve ser 0 ou 1 (padrão 0). |

Agora vejamos o código no editor para ver como criamos um objeto de datetime.

from datetime import datetime

dt = datetime(2019, 11, 4, 14, 53)

print("Datetime:", dt)

print("Date:", dt.date())

print("Time:", dt.time())

Resultado:

Datetime: 2019-11-04 14:53:00

Date: 2019-11-04

Time: 14:53:00

**output**

O exemplo cria um objeto datetime representando 4 de novembro de 2019 às 14:53:00. Todos os parâmetros passados para o construtor vão para atributos de classe só de leitura. São *ano*, *mês*, *dia*, *hora*, *minuto*, *segundo*, *microssegundo*, *tzinfo*e *fold*.

O exemplo mostra dois métodos que devolvem dois objetos diferentes. O método chamado date devolve o objeto *date* com o ano, mês e dia dados, enquanto o método chamado time devolve o objeto *time* com a hora e o minuto dados.

**Métodos que devolvem a data e hora atuais**

A classe datetime tem vários métodos que devolvem a data e hora atuais. Estes métodos são:

* today() — devolve a data e hora locais atuais com o atributo *tzinfo* definido para *None*;
* now() — devolve a data e hora local atuais, tal como o método *today*, a menos que lhe passemos o argumento opcional *tz*. O argumento deste método deve ser um objeto da subclasse *tzinfo*;
* utcnow() — devolve a data e hora UTC atuais com o atributo *tzinfo* definido para *None*.

Execute o código no editor para os ver a todos na prática.

from datetime import datetime

print("today:", datetime.today())

print("now:", datetime.now())

print("utcnow:", datetime.utcnow())

O que pode dizer sobre o output?

Como pode ver, o resultado de todos os três métodos é o mesmo. As pequenas diferenças são causadas pelo tempo decorrido entre chamadas subsequentes.

**Nota:** Pode ler mais sobre os objetos *tzinfo* na documentação.

# Obter um timestamp

Existem muitos conversores disponíveis na Internet que podem calcular um timestamp com base numa determinada data e hora, mas como o podemos fazer no módulo datetime ?

Isto é possível graças ao método timestamp fornecido pela classe datetime . Veja o código no editor.

from datetime import datetime

dt = datetime(2020, 10, 4, 14, 55)

print("Timestamp:", dt.timestamp())

Resultado:

Timestamp: 1601823300.0

A função timestamp devolve um valor float expressando o número de segundos decorridos entre a data e a hora indicada pelo objeto *datetime* e 1 de janeiro de 1970, 00:00:00 (UTC).

# Formatação de data e hora (parte 1)

Todas as classes de módulo datetime apresentadas até agora têm um método chamado strftime. Este é um método muito importante, porque nos permite devolver a data e a hora no formato que especificamos.

O método strftime só aceita um argumento sob a forma de uma string especificando o formato que pode consistir em diretivas.

Uma diretiva é uma string que consiste no caratere % (percentagem) e uma letra minúscula ou maiúscula, por exemplo, a diretiva %Y significa o ano com o século como um número decimal. Vejamo-lo num exemplo. Execute o código no editor.

from datetime import date

d = date(2020, 1, 4)

print(d.strftime('%Y/%m/%d'))

Resultado:

2020/01/04

**output**

No exemplo, passámos um formato constituído por três diretivas separadas por / (barra) para o método strftime . É claro que o caratere separador pode ser substituído por outro caratere, ou mesmo por uma string.

Pode colocar quaisquer carateres no formato, mas apenas as diretivas reconhecíveis serão substituídas pelos valores apropriados. No nosso formato, utilizámos as seguintes diretivas:

* %Y — devolve o ano com o século como um número decimal. No nosso exemplo, é 2020.
* %m — devolve o mês como um número decimal de zero. No nosso exemplo, é 01.
* %d — devolve o dia como um número decimal de zero. No nosso exemplo, é 04.

**Nota:** Pode encontrar todas as diretivas disponíveis [aqui](https://docs.python.org/3/library/datetime.html#strftime-and-strptime-format-codes).

# Formatação de data e hora (parte 2)

A formatação do tempo funciona da mesma forma que a formatação da data, mas requer a utilização de diretivas apropriadas. Vamos dar uma vista de olhos mais atenta a algumas delas no editor.

from datetime import time

from datetime import datetime

t = time(14, 53)

print(t.strftime("%H:%M:%S"))

dt = datetime(2020, 11, 4, 14, 53)

print(dt.strftime("%y/%B/%d %H:%M:%S"))

Resultado:

14:53:00

20/November/04 14:53:00

**output**

O primeiro dos formatos usados diz respeito apenas ao tempo. Como pode adivinhar, %H devolve a hora como um número decimal de zero, %M devolve o minuto como um número decimal de zero, enquanto %S devolve o segundo como um número decimal de zero. No nosso exemplo, %H é substituído por 14, %M por 53, e %S por 00.

O segundo formato usado combina diretivas de data e hora. Existem duas novas directivas, %Y e %B. A diretiva %Y devolve o ano sem um século como um número decimal de zero (no nosso exemplo, é 20). A diretiva %B devolve o mês como o nome completo do local (no nosso exemplo, é novembro).

Em geral, tem muita liberdade na criação de formatos, mas deve lembrar-se de utilizar corretamente as diretivas. Como um exercício, pode verificar o que acontece se, por exemplo, tentar usar a diretiva %Y no formato passado para o método *strftime* do objeto de tempo. Tente descobrir você mesmo porque obteve este resultado. Boa sorte!

# Os loops strftime() no módulo time .

Provavelmente não ficará surpreendido por saber que a função strftime está disponível no módulo time . Difere ligeiramente dos métodos strftime nas classes fornecidas pelo módulo datetime porque, para além do argumento format, também pode levar (opcionalmente) um tuple ou um objeto struct\_time.

Se não passar um tuple ou um objeto *struct\_time*, a formatação será feita utilizando a hora local atual. Dê uma vista de olhos no exemplo no editor.

import time

timestamp = 1572879180

st = time.gmtime(timestamp)

print(time.strftime("%Y/%m/%d %H:%M:%S", st))

print(time.strftime("%Y/%m/%d %H:%M:%S"))

O nosso resultado é o seguinte:

2019/11/04 14:53:00

2020/10/12 12:19:40

**output de amostra**

A criação de um format tem o mesmo aspeto que para os métodos strftime no módulo datetime . No nosso exemplo, usamos as diretivas %Y, %m, %d, %H, %M, e %S que já conhece.

Na primeira chamada de função, formatamos o objeto *struct\_time*, enquanto na segunda chamada (sem o argumento opcional), formatamos a hora local. Pode encontrar todas as diretivas disponíveis no módulo time [aqui](https://docs.python.org/3/library/time.html#time.strftime).

# Os loops strptime() método

Saber como criar um formato pode ser útil ao usar um método chamado strptime na datetime classe. Ao contrário do método strftime , este cria um objeto datetime a partir de uma string representando uma data e hora.

O método strptime requer que se especifique o formato em que se guardou a data e hora. Vejamo-lo num exemplo. Dê uma vista de olhos ao código no editor.

from datetime import datetime

print(datetime.strptime("2019/11/04 14:53:00", "%Y/%m/%d %H:%M:%S"))

Resultado:

2019-11-04 14:53:00

**output**

No exemplo, especificámos dois argumentos necessários. O primeiro é uma data e hora como uma string: "2019/11/04 14:53:00", enquanto o segundo é um formato que facilita a análise para um datetime objeto. Tenha cuidado, porque se o formato especificado não corresponder à data e hora na string, levantará um *ValueError*.

**Nota:** No módulo time , pode encontrar uma função chamada strptime, que analisa uma string representando um tempo para um objeto *struct\_time*. A sua utilização é análoga à do módulo strptime na classe datetime :

import time

print(time.strptime("2019/11/04 14:53:00", "%Y/%m/%d %H:%M:%S"))

O seu resultado será o seguinte:

time.struct\_time(tm\_year=2019, tm\_mon=11, tm\_mday=4, tm\_hour=14, tm\_min=53, tm\_sec=0, tm\_wday=0, tm\_yday=308, tm\_isdst=-1)

# Operações de data e hora

Mais cedo ou mais tarde, terá de efetuar alguns cálculos sobre a data e hora. Felizmente, há uma classe chamada timedelta no módulo datetime que foi criado precisamente para esse fim.

Para criar um objeto timedelta , basta fazer a subtração nos objetos date ou datetime , assim como fizemos no exemplo no editor. Execute-o.

from datetime import date

from datetime import datetime

d1 = date(2020, 11, 4)

d2 = date(2019, 11, 4)

print(d1 - d2)

dt1 = datetime(2020, 11, 4, 0, 0, 0)

dt2 = datetime(2019, 11, 4, 14, 53, 0)

print(dt1 - dt2)

Resultado:

366 days, 0:00:00

365 days, 9:07:00

**output**

O exemplo mostra a subtração para ambos os objetos date e datetime . No primeiro caso, recebemos a diferença em dias, que é de 366 dias. Observe que a diferença em horas, minutos e segundos também é exibida. No segundo caso, recebemos um resultado diferente, porque especificámos o tempo que foi incluído nos cálculos. Como resultado, recebemos 365 dias, 9 horas e 7 minutos.

Daqui a pouco irá aprender mais sobre a criação de objetos timedelta e sobre as operações que pode fazer com eles.

# Criar objetos timedelta .

Já aprendeu que um objeto timedelta pode ser devolvido como o resultado de subtrair dois objetos date ou datetime .

É claro que também pode criar um objeto por si próprio. Para este efeito, vamos conhecer os argumentos aceites pelo construtor da classe, que são: days, seconds, microseconds, milliseconds, minutes, hours, e weeks. Cada um deles é opcional e o padrão é 0.

Os argumentos devem ser números inteiros ou floating point, e podem ser positivos ou negativos. Vejamos um exemplo simples no editor.

from datetime import timedelta

delta = timedelta(weeks=2, days=2, hours=3)

print(delta)

Resultado:

16 days, 3:00:00

**output**

O resultado de 16 dias é obtido através da conversão do argumento weeks para dias (2 semanas = 14 dias) e adicionando o argumento days (2 dias). Este é um comportamento normal, porque o objeto timedelta armazena internamente apenas dias, segundos e microssegundos. Da mesma forma, o argumento hour é convertido em minutos. Veja o exemplo abaixo:

from datetime import timedelta

delta = timedelta(weeks=2, days=2, hours=3)

print("Days:", delta.days)

print("Seconds:", delta.seconds)

print("Microseconds:", delta.microseconds)

Resultado:

Days: 16

Seconds: 10800

Microseconds: 0

**output**

O resultado de 10800 é obtido convertendo 3 horas em segundos. Desta forma, o objeto timedelta armazena os argumentos passados durante a sua criação. As semanas são convertidas em dias, horas e minutos em segundos, e milissegundos em microssegundos.

Já sabe como o objeto timedelta armazena os argumentos passados internamente. É tempo de ver como pode ser utilizado na prática.

Vejamos algumas operações apoiadas pelas classes do módulo datetime . Execute o código que fornecemos no editor.

from datetime import timedelta

from datetime import date

from datetime import datetime

delta = timedelta(weeks=2, days=2, hours=2)

print(delta)

delta2 = delta \* 2

print(delta2)

d = date(2019, 10, 4) + delta2

print(d)

dt = datetime(2019, 10, 4, 14, 53) + delta2

print(dt)

Resultado:

16 days, 2:00:00

32 days, 4:00:00

2019-11-05

2019-11-05 18:53:00

**output**

A função timedelta pode ser multiplicado por um inteiro. No nosso exemplo, multiplicamos o objeto que representa 16 dias e 2 horas por 2. Como resultado, recebemos um objeto timedelta representando 32 dias e 4 horas.

Note que tanto os dias como as horas foram multiplicados por 2. Outra operação interessante usando o objeto timedelta é a adição. No exemplo, adicionamos o objeto timedelta à *data* e datetime objetos.

Como resultado destas operações, recebemos *data* e objetos datetime aumentados em dias e horas armazenados no timedelta objeto.

A operação de multiplicação apresentada permite-lhe aumentar rapidamente o valor do objeto timedelta , enquanto a multiplicação pode também ajudá-lo a obter uma data no futuro.

Claro, as classes timedelta, date, e datetime suportam muitas mais operações. Encorajamo-lo a familiarizar-se com elas na documentação.

# LAB Os módulos datatime e time

Fácil

## Objetivos

* melhorar as competências do aluno na formatação de data e hora;
* melhorar as competências do aluno na utilização do método strftime .

## Cenário

Durante este curso, aprendeu sobre o método strftime , que requer o conhecimento de diretivas para criar um formato. Chegou o momento de pôr em prática as diretivas conhecidas.

A propósito, terá a oportunidade de trabalhar com documentação, porque terá de encontrar diretivas que ainda não conhece.

**Aqui está a sua tarefa:**

Escreva um programa que crie um objeto datetime para 4 de novembro de 2020, 14:53:00. O objeto criado deve chamar o método strftime com o formato apropriado para exibir o seguinte resultado:

2020/11/04 14:53:00

20/November/04 14:53:00 PM

Wed, 2020 Nov 04

Wednesday, 2020 November 04

Weekday: 3

Day of the year: 309

Week number of the year: 44

**output esperado**

**Nota:** Cada linha de resultado deve ser criada chamando o método *strftime* com pelo menos uma diretiva no argumento do formato.

## RESPOSTA

from datetime import datetime

my\_date = datetime(2020, 11, 4, 14, 53)

print(my\_date.strftime("%Y/%m/%d %H:%M:%S"))

print(my\_date.strftime("%y/%B/%d %H:%M:%S %p"))

print(my\_date.strftime("%a, %Y %b %d"))

print(my\_date.strftime("%A, %Y %B %d"))

print(my\_date.strftime("Weekday: %w"))

print(my\_date.strftime("Day of the year: %j"))

print(my\_date.strftime("Week number of the year: %W"))

**Key takeaways**

1. Para criar um objeto date , deve passar os argumentos do ano, mês e dia como se segue:

from datetime import date

my\_date = date(2020, 9, 29)

print("Year:", my\_date.year) # Year: 2020

print("Month:", my\_date.month) # Month: 9

print("Day:", my\_date.day) # Day: 29

O objeto date tem três atributos (somente de leitura): ano, mês e dia.

2. O método today devolve um objeto de data que representa a data local atual:

from datetime import date

print("Today:", date.today()) # Displays: Today: 2020-09-29

3. Em Unix, o timestamp expressa o número de segundos desde 1 de janeiro de 1970, 00:00:00 (UTC). Esta data é chamada "época Unix", porque começou a contagem do tempo nos sistemas Unix. O timestamp é na realidade a diferença entre uma determinada data (incluindo a hora) e 1 de janeiro de 1970, 00:00:00 (UTC), expressa em segundos. Para criar um objeto de data a partir de um timestamp, temos de passar um timestamp Unix para o método fromtimestamp :

from datetime import date

import time

timestamp = time.time()

d = date.fromtimestamp(timestamp)

Note: A função time devolve o número de segundos desde 1 de janeiro de 1970 até ao momento atual sob a forma de um número float.

4. O construtor da classe time aceita seis argumentos (*hora*, *minuto*, *segundo*, *microssegundo*, *tzinfo*e *fold*). Cada um destes argumentos é opcional.

from datetime import time

t = time(13, 22, 20)

print("Hour:", t.hour) # Hour: 13

print("Minute:", t.minute) # Minute: 22

print("Second:", t.second) # Second: 20

5. O módulo time contém a função sleep , que suspende a execução do programa por um determinado número de segundos, por exemplo:

import time

time.sleep(10)

print("Hello world!") # This text will be displayed after 10 seconds.

6. no módulo datetime , data e hora podem ser representados como objetos separados ou como um único objeto. A classe que combina data e hora é chamada *datetime*. Todos os argumentos passados ao construtor vão para atributos de classe só de leitura. São *ano*, *mês*, *dia*, *hora*, *minuto*, *segundo*, *microssegundo*, *tzinfo*e *fold*:

from datetime import datetime

dt = datetime(2020, 9, 29, 13, 51)

print("Datetime:", dt) # Displays: Datetime: 2020-09-29 13:51:00

7. O método strftime só aceita um argumento sob a forma de uma string especificando um formato que pode consistir em diretivas. Uma diretiva é uma string que consiste no caratere % (percentagem) e uma letra minúscula ou maiúscula. Abaixo estão algumas diretivas úteis:

* %Y – devolve o ano com o século como um número decimal;
* %m — devolve o mês como um número decimal de zero;
* %d — devolve o dia como um número decimal de zero;
* %H — devolve a hora como um número decimal de zero;
* %M — devolve o minuto como um número decimal de zero;
* %S — devolve o segundo como um número decimal de zero.

Exemplo:

from datetime import date

d = date(2020, 9, 29)

print(d.strftime('%Y/%m/%d')) # Displays: 2020/09/29

8. É possível realizar cálculos em objetos date e datetime , por exemplo:

from datetime import date

d1 = date(2020, 11, 4)

d2 = date(2019, 11, 4)

d = d1 - d2

print(d) # Displays: 366 days, 0:00:00.

print(d \* 2) # Displays: 732 days, 0:00:00.

O resultado da subtração é devolvido como um objeto timedelta que expressa a diferença em dias entre as duas datas no exemplo acima.

Observe que a diferença em horas, minutos e segundos também é exibida. O objeto timedelta pode ser usado para cálculos adicionais (por exemplo, pode multiplicá-lo por 2).

**Exercício 1**

Qual é o output do seguinte snippet?

from datetime import time

t = time(14, 39)

print(t.strftime("%H:%M:%S"))

**Exercício 2**

Qual é o output do seguinte snippet?

from datetime import datetime

dt1 = datetime(2020, 9, 29, 14, 41, 0)

dt2 = datetime(2020, 9, 28, 14, 41, 0)

print(dt1 - dt2)

# Introdução ao módulo calendar .

Para além dos módulos datetime e time , a biblioteca padrão Python fornece um módulo chamado calendar que, como o nome sugere, oferece **funções relacionadas com o calendário**.

Uma delas é, evidentemente, a exibição do calendário. É importante que os dias da semana sejam exibidos de segunda a domingo, e cada dia da semana tenha a sua representação sob a forma de um número inteiro:

|  |  |  |
| --- | --- | --- |
| **Dia da semana** | **Valor inteiro** | **Constante** |
| Segunda-feira | 0 | calendar.MONDAY |
| Terça-feira | 1 | calendar.TUESDAY |
| Quarta-feira | 2 | calendar.WEDNESDAY |
| Quinta-feira | 3 | calendar.THURSDAY |
| Sexta-feira | 4 | calendar.FRIDAY |
| Sábado | 5 | calendar.SATURDAY |
| Domingo | 6 | calendar.SUNDAY |

A tabela acima mostra a representação dos dias da semana no módulo calendar . O primeiro dia da semana (segunda-feira) é representado pelo valor *0* e pela constante *calendar.MONDAY*, enquanto que o último dia da semana (domingo) é representado pelo valor *6* e pela constante *calendar.SUNDAY*.



Para os meses, os valores inteiros são indexados a partir de 1, ou seja, janeiro é representado por 1, e dezembro por 12. Infelizmente, não há constantes que expressem os meses.

As informações acima referidas ser-lhe-ão úteis quando trabalhar com o módulo calendar nesta parte do curso, mas primeiro vamos começar com alguns exemplos simples de calendário.

**O seu primeiro calendário**

Começará a sua aventura com o módulo calendar com uma função simples chamada calendar, que permite **exibir o calendário para todo o ano**. Vejamos como usá-lo para exibir o calendário de 2020. Execute o código no editor e veja o que acontece.

import calendar

print(calendar.calendar(2020))

O resultado exibido é semelhante ao resultado do comando *cal* disponível em Unix. Se quiser alterar a formatação padrão do calendário, pode utilizar os seguintes parâmetros:

* w – largura da coluna de data (default 2)
* l – número de linhas por semana (default 1)
* c — número de espaços entre as colunas do mês (default 6)
* m — número de colunas (default 3)

A função de calendário requer que se especifique o ano, enquanto que os outros parâmetros responsáveis pela formatação são opcionais. Encorajamo-lo a experimentar por si próprio estes parâmetros.

Uma boa alternativa à função acima é a função chamada *prcal*, que também toma os mesmos parâmetros que a função calendar , mas não requer o uso da função print para exibir o calendário. A sua utilização assemelha-se a esta:

import calendar

calendar.prcal(2020)

**Calendário para um mês específico**

O módulo calendar tem uma função chamada month, que permite exibir um calendário para um mês específico. A sua utilização é muito simples, apenas precisa de especificar o ano e o mês - verifique o código no editor.

import calendar

print(calendar.month(2020, 11))

O exemplo exibe o calendário para novembro de 2020. Como na função calendar , pode alterar a formatação padrão usando os seguintes parâmetros:

* w – largura da coluna de data (default 2)
* l – número de linhas por semana (default 1)

**Nota:** Também pode usar a função prmonth , que tem os mesmos parâmetros que a função month , mas não exige que use a função print para exibir o calendário.

# Os loops setfirstweekday() .

Como já sabe, por defeito no módulo calendar , o primeiro dia da semana é segunda-feira. No entanto, pode alterar este comportamento usando uma função chamada setfirstweekday.

Lembra-se da tabela que mostra os dias da semana e a sua representação sob a forma de valores inteiros? É altura de a utilizar, porque o módulo setfirstweekday requer um parâmetro expressando o dia da semana sob a forma de um valor inteiro. Dê uma vista de olhos no exemplo no editor.

import calendar

calendar.setfirstweekday(calendar.SUNDAY)

calendar.prmonth(2020, 12)

O exemplo utiliza a constante calendar.SUNDAY , que contém um valor de *6*. Claro que pode passar esse valor diretamente para a função setfirstweekday , mas a versão com uma constante é mais elegante.

Como resultado, obtemos um calendário mostrando o mês de dezembro de 2020, no qual o primeiro dia de todas as semanas é domingo.

# Os loops weekday() .

Outra função útil fornecida pelo módulo calendar é a função chamada weekday, que devolve o dia da semana como um valor inteiro para o ano, mês e dia em questão. Vamos vê-la na prática.

Execute o código no editor para verificar o dia da semana que cai a 24 de dezembro de 2020.

import calendar

print(calendar.weekday(2020, 12, 24))

Resultado:

3

A função weekday devolve 3, o que significa que 24 de dezembro de 2020 é uma quinta-feira.

# Os loops weekheader() .

Já deve ter reparado que o calendário contém cabeçalhos semanais de forma abreviada. Se necessário, pode obter nomes curtos de dias de semana usando o método weekheader .

O método weekheader exige que especifique a largura em carateres para um dia da semana. Se a largura que fornecer for superior a 3, ainda receberá os nomes abreviados dos dias da semana compostos por três carateres.

Vejamos então como obter um cabeçalho mais pequeno. Execute o código no editor.

import calendar

print(calendar.weekheader(2))

Resultado:

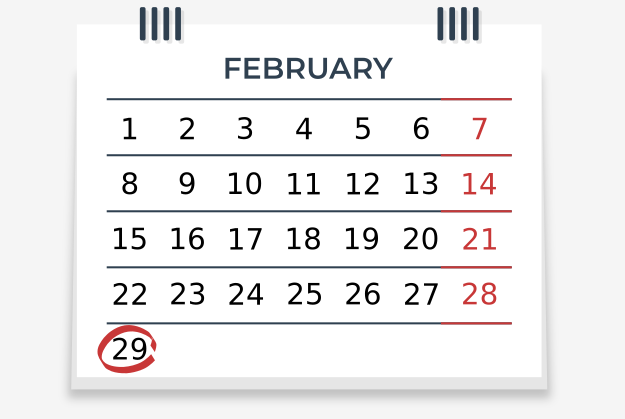
Mo Tu We Th Fr Sa Su

**output**

**Nota**: Se mudar o primeiro dia da semana, por exemplo, utilizando a função setfirstweekday , afetará o resultado da função weekheader .

# Como verificamos se um ano é um ano bissexto?

O módulo calendar fornece duas funções úteis para verificar se os anos são anos bissextos.



O primeiro, chamado isleap, devolve *True* se o ano passado for bissexto, ou *False* caso contrário. O segundo, chamado leapdays, devolve o número de anos bissextos num determinado intervalo de anos.

Execute o código no editor.

import calendar

print(calendar.isleap(2020))

print(calendar.leapdays(2010, 2021)) # Up to but not including 2021.

Resultado:

True

3

**output**

No exemplo, obtemos o resultado 3, porque no período de 2010 a 2020 há apenas três anos bissextos (nota: 2021 não está incluído). São os anos de 2012, 2016 e 2020.

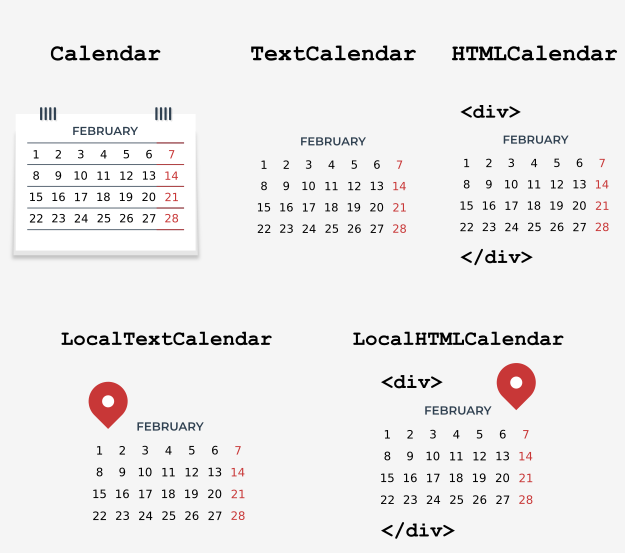
**Classes para criação de calendários**

As funções apresentadas não são tudo o que o módulo calendar oferece. Para além delas, podemos utilizar as seguintes classes:

* calendar.Calendar - fornece métodos para preparar dados de calendário para formatação;
* calendar.TextCalendar - é utilizada para criar calendários de texto regulares;
* calendar.HTMLCalendar - é utilizada para criar calendários HTML;
* calendar.LocalTextCalendar — é uma subclasse da classe calendar.TextCalendar . O construtor desta classe toma o parâmetro *locale*, que é utilizado para devolver os nomes apropriados dos meses e dias da semana.
* calendar.LocalHTMLCalendar — é uma subclasse da classe calendar.HTMLCalendar . O construtor desta classe toma o parâmetro locale, que é utilizado para devolver os nomes apropriados dos meses e dias da semana.

Durante este curso, já teve a oportunidade de criar calendários de texto ao discutir as funções do módulo calendar .

Está na hora de experimentar algo novo. Vamos analisar mais de perto os métodos da classe calendar .



# Criar um objeto Calendar .

O construtor de classes Calendar toma um parâmetro opcional chamado firstweekday, por defeito igual a 0 (segunda-feira).

O parâmetro firstweekday deve ser um inteiro entre 0-6. Para este efeito, podemos utilizar as constantes já conhecidas - ver o código no editor.

import calendar

c = calendar.Calendar(calendar.SUNDAY)

for weekday in c.iterweekdays():

print(weekday, end=" ")

O programa fará output do seguinte resultado:

6 0 1 2 3 4 5

**output**

O exemplo de código usa o método de classe Calendar chamado iterweekdays, que devolve um iterador para os números dos dias da semana.

O primeiro valor devolvido é sempre igual ao valor da propriedade firstweekday . Visto no nosso exemplo o primeiro valor devolvido ser 6, isso significa que a semana começa num domingo.

# Os loops itermonthdates() .

A classe Calendar tem vários métodos que devolvem um iterador. Um deles é o método itermonthdates , que requer a especificação do ano e mês.

Como resultado, todos os dias no mês e ano especificados são devolvidos, tal como todos os dias antes do início do mês ou do final do mês que são necessários para obter uma semana completa.

Cada dia é representado por um objeto datetime.date . Dê uma vista de olhos no exemplo no editor.

import calendar

c = calendar.Calendar()

for date in c.itermonthdates(2019, 11):

print(date, end=" ")

O código exibe todos os dias de novembro de 2019. Como o primeiro dia de novembro de 2019 foi uma sexta-feira, os dias seguintes também são devolvidos para obter a semana completa: 10/28/2019 (segunda-feira) 10/29/2019 (terça-feira) 10/30/2019 (quarta-feira) 10/31/2019 (quinta-feira).

O último dia de novembro de 2019 foi um sábado, portanto, para manter a semana completa, mais um dia é devolvido 12/01/2019 (sexta-feira).

**Outros métodos que devolvem iteradores**

Outro método útil na classe Calendar é o método chamado itermonthdates, que toma ano e mês como parâmetros e, em seguida, devolve o iterador para os dias da semana representados por números.

Dê uma vista de olhos no exemplo no editor.

import calendar

c = calendar.Calendar()

for iter in c.itermonthdays(2019, 11):

print(iter, end=" ")

Certamente notou no grande número de 0s devolvidos como resultado do código de exemplo. Estes são dias fora do intervalo de meses especificado, que são adicionados para manter a semana completa.

Os quatro primeiros zeros representam 28/10/2019 (segunda-feira) 29/10/2019 (terça-feira) 30/10/2019 (quarta-feira) 31/10/2019 (quinta-feira). Os restantes números são dias no mês, exceto o último valor de 0, que substitui a data 01/12/2019 (domingo).

Existem quatro outros métodos semelhantes na classe Calendar que diferem nos dados devolvidos:

* itermonthdates2 — devolve dias sob a forma de tuples constituídas por um número de dia do mês e um número de dia da semana;
* itermonthdates3 — devolve dias sob a forma de tuples constituídas por um número do ano, um do mês e um do dia do mês. Este método está disponível a partir da versão 3.7 do Python.
* itermonthdates4 — devolve dias sob a forma de tuples constituídas por um número do ano, um do mês, um do dia do mês e um do dia da semana. Este método está disponível a partir da versão 3.7 do Python.

Para fins de teste, utilize o exemplo acima e veja como se apresentam na prática os valores de retorno dos métodos descritos.

# Os loops monthdays2calendar() .

A classe Calendar tem vários outros métodos úteis, sobre os quais pode aprender mais na documentação (<https://docs.python.org/3/library/calendar.html>).

Um deles é o método monthdays2calendar , que toma o ano e o mês, e, em seguida, devolve uma lista de semanas num mês específico. Cada semana é um tuple composto por números do dia e números do dia da semana. Veja o código no editor.

import calendar

c = calendar.Calendar()

for data in c.monthdays2calendar(2020, 12):

print(data)

Note-se que os números dos dias fora do mês são representados por 0, enquanto os números do dia da semana são um número de 0-6, onde 0 é segunda-feira e 6 é domingo.

Dentro de momentos, este método pode ser-lhe útil para completar uma tarefa de laboratório. Está preparado?

# LAB O módulo calendar

Fácil

## Objetivos

* Melhorar as competências do aluno na utilização da classe *Calendar*.

## Cenário

Durante este curso, olhámos um pouco para a classe Calendar . A sua tarefa é alargar a sua funcionalidade com um novo método chamado count\_weekday\_in\_year, que toma como parâmetros um ano e um dia da semana e, em seguida, devolve o número de ocorrências de um dia da semana específico no ano.

Use as seguintes dicas:

* Crie uma classe chamada MyCalendar que estende a classe Calendar ;
* crie o método count\_weekday\_in\_year com os parâmetros do ano e do dia da semana. O parâmetro do dia da semana deve ser um valor entre 0-6, onde 0 é segunda-feira e 6 é domingo. O método deve devolver o número de dias como um inteiro;
* na sua implementação, use o método monthdays2calendar da classe Calendar .

Os resultados esperados são os seguintes:

**Argumentos de amostra**

year=2019, weekday=0

**Output esperado**

52

**Argumentos de amostra**

year=2000, weekday=6

**Output esperado**

53

## RESPOSTA

import calendar

class MyCalendar(calendar.Calendar):

def count\_weekday\_in\_year(self, year, weekday):

current\_month = 1

number\_of\_days = 0

while (current\_month <= 12):

for data in self.monthdays2calendar(year, current\_month):

if data[weekday][0] != 0:

number\_of\_days = number\_of\_days + 1

current\_month = current\_month + 1

return number\_of\_days

my\_calendar = MyCalendar()

number\_of\_days = my\_calendar.count\_weekday\_in\_year(2019, calendar.MONDAY)

print(number\_of\_days)

# Key takeaways

1. no módulo calendar , os dias da semana são exibidos de segunda-feira a domingo. Cada dia da semana tem a sua representação sob a forma de um número inteiro, onde o primeiro dia da semana (segunda-feira) é representado pelo valor 0, enquanto o último dia da semana (domingo) é representado pelo valor 6.

2. Para exibir um calendário para qualquer ano, chame a função calendar com o ano passado como seu argumento, por exemplo:

import calendar

print(calendar.calendar(2020))

Nota: Uma boa alternativa para a função acima é a função chamada prcal, que também toma os mesmos parâmetros que a função calendar , mas não requer o uso da função print para exibir o calendário.

3. Para exibir um calendário para qualquer mês do ano, chame a função month , passando ano e mês para ela. Por exemplo:

import calendar

print(calendar.month(2020, 9))

Nota: Também pode utilizar a função prmonth , que tem os mesmos parâmetros que a função month , mas não requer o uso da função print para exibir o calendário.

4. O objeto da exceção setfirstweekday permite-lhe alterar o primeiro dia da semana. Toma um valor de 0 a 6, onde 0 é domingo e 6 é sábado.

5. O resultado da função weekday é um dia da semana como um valor inteiro para um determinado ano, mês e dia:

import calendar

print(calendar.weekday(2020, 9, 29)) # This displays 1, which means Tuesday.

6. A função weekheader devolve os nomes dos dias da semana numa forma abreviada. O método weekheader exige que especifique a largura em carateres para um dia da semana. Se a largura que fornecer for superior a 3, ainda receberá os nomes abreviados dos dias da semana compostos por apenas três carateres. Por exemplo:

import calendar

print(calendar.weekheader(2)) # This display: Mo Tu We Th Fr Sa Su

7. Uma função muito útil disponível no módulo calendar é a função chamada isleap, que, como o nome sugere, permite verificar se o ano é um ano bissexto ou não:

import calendar

print(calendar.isleap(2020)) # This displays: True

8. Pode criar um objeto calendar você mesmo usando a classe Calendar , que, ao criar o seu objeto, lhe permite alterar o primeiro dia da semana com o parâmetro opcional firstweekday , por exemplo:

import calendar

c = calendar.Calendar(2)

for weekday in c.iterweekdays():

print(weekday, end=" ")

# Result: 2 3 4 5 6 0 1

O iterweekdays devolve um iterador para números de dias úteis. O primeiro valor devolvido é sempre igual ao valor da propriedade firstweekday .

**Exercício 1**

Qual é o output do seguinte snippet?

import calendar

print(calendar.weekheader(1))

Verifique

**Exercício 2**

Qual é o output do seguinte snippet?

import calendar

c = calendar.Calendar()

for weekday in c.iterweekdays():

print(weekday, end=" ")

**Parabéns! Completou o *PE2: Módulo 4*.**

Muito bem! Chegou ao fim do Módulo 4 e completou um marco importante na sua educação em programação Python. Aqui está um breve resumo dos objetivos que abordou e com os quais se familiarizou no Módulo 4:

* geradores e iteradores;
* compreensões de lista;
* as funções *lambda*, *map* e *filter*;
* closures;
* trabalhar com ficheiros (streams de ficheiros, processamento de ficheiros, diagnóstico de problemas de stream)
* processamento de texto e ficheiros binários;
* módulos STL Python selecionados: *os*, *datetime*, *time* e *calendar*.

Está agora pronto para fazer o quiz do módulo e tentar o desafio final: Teste do Módulo 4, que o ajudará a avaliar o que aprendeu até agora.

