

Table of Contents

Introdução	1.1
Sobre o título	1.1.1
Porquê um livro sobre Python?	1.1.2
Porquê mais um livro de Python?	1.1.3
Sobre o nome e o criador da linguagem Python	1.1.4
Free Software e Free Documentation	1.1.5
Instalação	1.1.6
Editor	1.1.7
Sobre a versão Python	1.1.8
Hello World	1.2
Hello print and strings	1.2.1
Sobre funções	1.2.1.1
Sobre valores string	1.2.1.2
Sobre programas	1.2.1.3
Hello numbers, * and -	1.2.2
Sobre valores float	1.2.2.1
Hello round	1.2.3
Hello variáveis	1.2.4
Sobre variáveis	1.2.4.1
Hello comentários	1.2.5
Sobre comentários	1.2.5.1
Hello funções	1.2.6
Sobre a definição de funções	1.2.6.1
Hello if else	1.2.7
Sobre if else	1.2.7.1
Sobre valores int	1.2.7.2
Hello == e !=	1.2.8
Hello list e for	1.2.9
Sobre listas	1.2.9.1
Sobre o ciclo for	1.2.9.2
Hello range	1.2.10
Hello len	1.2.11
Hello return e append	1.2.12
Sobre return	1.2.12.1
Sobre append	1.2.12.2
Hello <, >=, and	1.2.13
Sobre valores bool	1.2.13.1

Sobre funções

Sobre operadores de comparação	1.2.13.2
Sobre operadores lógicos	1.2.13.3
Hello input	1.2.14
Hello while	1.2.15
Sobre o ciclo while	1.2.15.1
Hello ficheiros	1.2.16
Sobre ficheiros	1.2.16.1
Hello tuple	1.2.17
Sobre tuplos	1.2.17.1
Hello set	1.2.18
Sobre valores set	1.2.18.1
Hello dicionários	1.2.19
Sobre valores dict	1.2.19.1
[Hello from import]	1.2.20

Hello Python World

Introdução

Este livro é uma introdução à linguagem de programação Python, para quem não tem qualquer experiência de programação.

Quem já tem alguma experiência de programação, poderá ter algum interesse nos capítulos "Mais exemplos sobre..." e "Detalhes sobre...". Ou até, se estiver a aprender Python, nos restantes capítulos.

Licenças

Este livro é de utilização livre. É distribuído sob a licença [CC BY-SA 4.0](#). Sucintamente, o leitor tem a liberdade de:

- Partilhar — copiar e redistribuir o material em qualquer suporte ou formato;
- Adaptar — remisturar, transformar, e criar a partir do material para qualquer fim, mesmo que comercial.

Os programas neste livro são distribuídos sob a licença [GNU GPL3](#). São *free software*. Os utilizadores têm a liberdade de:

- Usar os programas para qualquer fim;
- Alterar os programas para os adaptar às suas necessidades;
- Partilhar os programas;
- Partilhar as alterações.

Estas liberdades são concedidas sob a condição de ser atribuído o devido crédito aos autores, e sob a condição da redistribuição ser efetuada segundo a mesma licença.

Sobre o título

Quando os programadores usam numa nova linguagem de programação, costumam começar por fazer um programa que quando é executado mostra a frase "Hello World". Funciona como o primeiro teste no novo mundo. O nome deste livro é inspirado neste hábito.

Porquê um título em inglês num livro em português? Porque a língua *standard* usada em tecnologia é o inglês. Escrevemos um livro em português porque somos portugueses e comunicamos e pensamos em português. Mas usamos o inglês para produzir tecnologia. Se os programas que fazemos podem ser úteis a outras pessoas, noutras partes do mundo, devemos escrevê-los em inglês. Neste contexto não faria nenhum sentido usar como título "Olá Mundo Python". Ninguém, mesmo em Portugal, saberia a que é que este título se referiria, porque ninguém usa a tradução "Olá Mundo" quando faz o "Hello World". Toda a gente, em Portugal e por esse mundo fora, usa "Hello World".

Porquê um livro sobre Python?

Temos a sorte de poder dizer que a linguagem Python é a linguagem de programação mais usada a nível global. Basea-mo-nos no índice [TIOBBE](#) conhecido como o ranking de referência das linguagens de programação. Segundo este índice, à data de setembro de 2022, a linguagem Python é a mais usada em todo o globo. E com uma tendência clara de subida! Na nossa opinião isto deve-se ao facto da linguagem Python ser fácil de aprender, ser muito poderosa e também muito produtiva. Mas seja qual for a justificação os números estão à vista de todos...

Dizemos que temos a sorte em poder dizer que a linguagem Python é a linguagem de programação mais usada a nível global, porque somos professores de informática e ensinamos Python. Poderíamos ter que dizer aos nossos alunos que ensinamos uma linguagem de uso exclusivamente académico, mas com todos os princípios das linguagens mais usadas. E que portanto valeria a pena aprender essa linguagem totalmente desconhecida. Mas felizmente temos a sorte de poder dizer aos nossos alunos que ensinamos a linguagem mais usada no planeta, o que quer dizer que aprender Python é sem dúvida alguma uma enorme mais valia a nível profissional.

Porquê mais um livro de Python?

Esta é uma boa pergunta. :-) (fazemos os *smiles* assim, sem imagens, porque somos programadores e os programas de computador são textos.) O que isto quer dizer é que na verdade não temos uma boa resposta para ela. :-) Mas a resposta até é simples. Este livro foi escrito para ensinar Python aos nossos alunos. Ensinar é uma atividade criativa e bastante pessoal. E nós sentimos a necessidade de escrever este livro para dar suporte à nossa prática pessoal de ensino.

Aquilo que poderemos destacar relativamente a este livro, e que poderá eventualmente ser diferenciador, é:

- Os temas são apresentados partindo de exemplos simples. Por isso adotámos o mote *Hello World*, e estendemo-lo aos vários temas. No capítulo *Hello World*, temos secções como *Hello print and strings* ou *Hello numbers and operators*, entre outras.
- Logo no capítulo *Hello World* atravessamos todos os temas abordados no livro. Isto só é possível porque optámos por nos limitarmos a um único exemplo para cada tema.
- Deixamos a exploração de mais exemplos sobre cada tema para o capítulo "Mais exemplo sobre...".
- Deixamos a apresentação dos detalhes associados a cada tema para o capítulo "Detalhes...".

O objetivo principal desta nossa abordagem é facilitar uma entrada mais rápida no mundo Python. Garantindo ao mesmo tempo que se obtêm as bases que permitirão programar em Python de forma autónoma.

Sobre o nome e o criador da linguagem Python

A linguagem de programação Python foi criada pelo programador holandês [Guido van Rossum](#) em 1989, quando estava à procura de um *hobby* que o ocupasse durante a semana do Natal, pois o seu escritório estava fechado.

Escolheu o nome Python porque era grande fã dos [Monty Python's Flying Circus](#) e estava com um humor irreverente. :-)

Mais tarde candidatou-se a financiamento para o desenvolvimento da linguagem Python, dizendo que se tratava de uma "linguagem de programação para toda a gente", com os seguintes objetivos:

- ser uma linguagem fácil e intuitiva tão poderosa como as suas concorrentes;
- de "código aberto" para que toda a gente pudesse contribuir para o seu desenvolvimento;
- tão compreensível como o Inglês;
- adequada às tarefas de todos os dias, permitindo tempos de desenvolvimento curtos.

Passados todos estes anos verifica-se por um lado que a linguagem Python tem de facto estas características. Por outro lado, dado o sucesso da linguagem, constata-se que estas características foram uma excelente escolha.

Guido van Rossum é conhecido na comunidade Python como o [benevolent dictator for life \(BDFL\)](#) mas já não exerce o seu papel de ditador benevolente desde 2018.

Free Software e Free Content

No mundo digital tudo se representa com *bits*. Um *bit* ou toma o valor zero ou o valor um. É um valor binário. Um conteúdo digital é uma sequência de bits. Por exemplo "010010". Uma cópia deste conteúdo digital é a sequência de bits "010010". A cópia é rigorosamente igual ao conteúdo original. De tal forma que até se poderá pôr em causa se os conceitos de original e cópia serão aplicáveis. As duas versões são indistinguíveis. Qual é a original? São iguais. São as duas. Esta é uma particularidade do mundo digital. É diferente do mundo físico em que em rigor não há dois objetos físicos exatamente iguais. Por força de serem constituídos por materiais, a cópia de um objecto físico nunca é rigorosamente igual ao original. A diferença pode ser microscópica, mas existe sempre.

Os programas de computador, a que se chama *software*, são conteúdos digitais. O facto das cópias dos programas de computador serem rigorosamente iguais aos originais sugere que deverão existir regras especiais para proteção de cópias de *software*. Isso é verdade e há de facto muitas regras especiais para proteção de *software* em particular e de conteúdos digitais em geral, como textos, imagens, músicas e vídeos, entre outros.

Mas há um movimento que viu nesta particularidade uma oportunidade para fazer as coisas de maneira diferente. É o movimento [Free Software](#). A ideia principal é a de que se as cópias são exatamente iguais aos originais, então que toda a gente seja livre de copiar, executar, modificar, contribuir e partilhar o *software*. Em vez de fechar o *software* sob proteções especiais, abrir o *software* para utilização generalizada. De forma a beneficiar, por um lado, quem precisa do *software* e em geral toda a sociedade, e por outro lado, beneficiar o próprio *software*, que por esta via vai sendo aperfeiçoado constantemente. Não deixa de ser curioso que uma ideia socialmente tão importante tenha sido concretizada no mundo das máquinas. Saliente-se que *free* em *Free Software* que dizer livre. E não grátis. *Free Software* diz respeito à liberdade de utilização. Não ao preço. Toda a gente é livre de vender *Free Software* pelo preço que quiser. Ou melhor pelo preço que conseguir. Isto porque não será fácil vender uma coisa que toda a gente pode obter sem pagar. O modelo de negócio associado ao *Free Software* está mais relacionada com a venda da manutenção. Não propriamente do *software*. Por exemplo, a empresa RedHat usa este modelo. Em 2018 a RedHat foi comprada pela IBM por [34 biliões de dólares](#). Existem muitas outras empresas que usam modelos de negócio baseados em *Free Software*.

A linguagem Python é *Free Software*. Mais precisamente o programa que executa a linguagem Python, é *Free Software*. Por isso dissemos acima que o linguagem Python é de "código aberto". Outros exemplos de programas *Free Software* muitíssimo utilizados são o sistema operativo GNU/Linux e a suite LibreOffice, entre outros. Mas o número programas *Free Software* é gigantesco e abranje praticamente todas as áreas da produção de *software*. Para além da liberdade, esta possibilidade é também muitíssimo importante do ponto de vista profissional em termos de custos de *software*. Tanto num computador para uso pessoal (uma *workstation*) como num servidor é possível usar-se exclusivamente programas *Free Software*. O que quer dizer que não existem custos de aquisição de *software*.

Este movimento chegou entretanto a outras áreas da produção digital para além do *software*, como os textos, as imagens e os próprios livros, entre outros. Os textos e a grande maioria das imagens da Wikipedia são de [utilização livre](#). Este livro é de utilização livre. É distribuído segundo a licença [CC BY-SA 4.0](#).

As liberdades definidas por esta licença são concedidas sob a condição de ser atribuído o devido crédito aos autores, e sob a condição da redistribuição ser efetuada sob a mesma licença. É uma forma de perpetuar a liberdade de utilização preservando os autores e os seus direitos.

Existem muitos outros livros livres. Inclusive sobre Python. Por exemplo, [A Byte of Python](#) e [Think Python 2e](#).

Instalação

Um programa de computador é uma sequência de instruções. Os computadores têm memória para armazenar programas e dados. E têm um processador, o CPU (*central processing unit*), para executar as instruções dos programas.

Mas os programas de computador são, na maior parte dos casos, escritos em linguagens como Python que se destinam a serem lidas por seres humanos, os programadores. Não são escritos com instruções dos processadores. Faz-se assim porque é muito mais produtivo escrever os programas em linguagens mais próximas das línguas como o inglês. Aliás é esta a razão da existência das linguagens de programação.

Por isso para se poder executar um programa escrito em Python é necessário ter um interpretador Python. O interpretador, que é também ele um programa, lê o programa Python, converte as instruções Python para o formato do processador existente no computador, e executa essas instruções.

Linux e MacOS

A instalação do interpretador Python em Linux e em MacOS é fácil. Já vem instalado no computador. :-)

Windows

Para instalar o interpretador Python em Windows [siga este vídeo](#).

Editor

Para se escrever um programa em Python, assim como noutras linguagens, pode usar-se um qualquer editor de texto, como por exemplo o *Word*, ou até mesmo o "Bloco de Notas". Embora isto seja possível, não é frequente usar-se um destes editores para escrever programas de computador. Isto porque há funcionalidades específicas da escrita de programas que não existem nestes editores. Por exemplo, a execução do próprio programa que está a ser escrito, é uma funcionalidade bastante útil e muito frequente em editores de linguagens de programação.

Mas há muitas mais funcionalidades específicas do desenvolvimento de programas que deram origem ao aparecimento dos chamados SDKs, *software development kits*. Existem imensos SDKs que suportam Python.

Para escrever e executar os programas deste livro não é necessário um editor muito sofisticado. Por isso, para evitar o tempo de aprendizagem de um editor sofisticado ou de um SDK, sugerimos a utilização do IDLE. O IDLE é um editor que vem incorporado com o próprio interpretador de Python. Por isso, quem já tem o interpretador de Python instalado no seu computador, já tem o IDLE.

O essencial do IDLE que permite escrever e executar os programas deste livro resume-se aos seguintes quatro comandos:

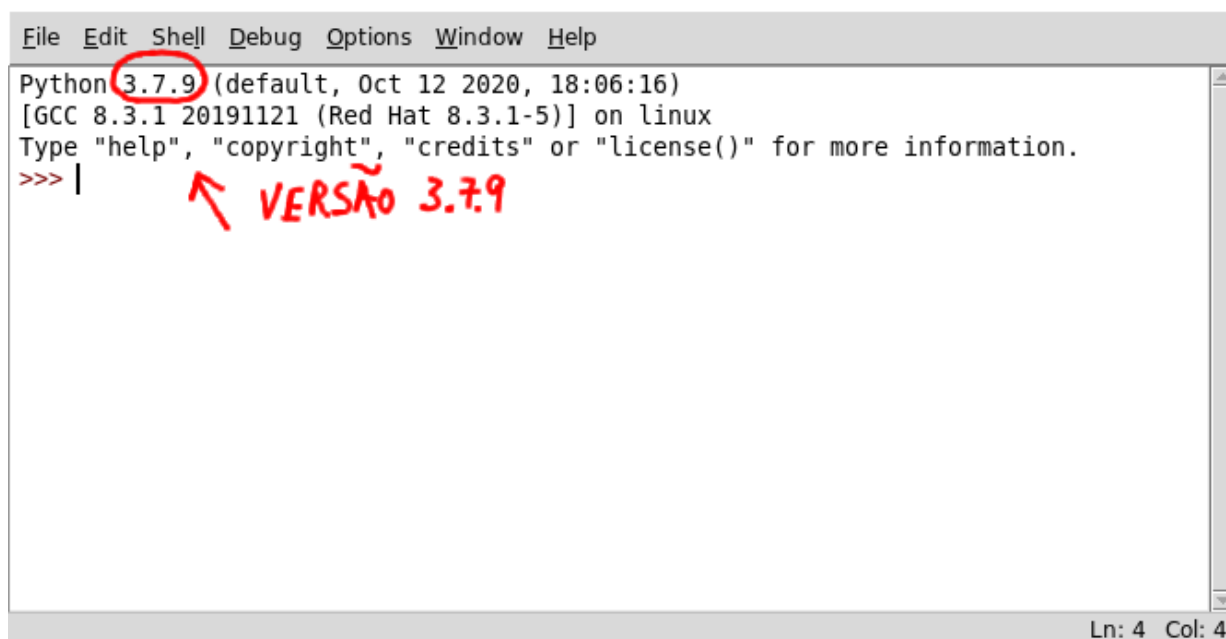
- `Crtl n` - Criar um novo programa ("n" de *new*);
- `Crtl s` - Gravar um programa ("s" de *save*);
- `F5` - Executar um programa;
- `Crtl o` - Abrir um programa já existente ("o" de *open*).

Naturalmente que incentivamos a exploração de outros comandos que facilitam o desenvolvimento dos programas.

Sobre a versão Python

As linguagens de programação vão evoluindo ao longo do tempo porque todos os dias lhes são acrescentadas novas funcionalidades. Na maior parte dos casos as novas versões são compatíveis com as anteriores. Isto é muito importante porque torna possível usar as novas versões, que têm mais funcionalidades, e são mais poderosas, com os antigos programas. Não se tem que alterar os programas já existentes para usar as novas versões. Mas no caso da evolução da linguagem Python da versão 2 para a versão 3 não foi possível assegurar essa compatibilidade. Isso veio obrigar a ter que se especificar a versão Python usada. No caso deste livro usamos a versão mais recente, que é a versão 3. Por isso quando nos referimos à linguagem Python esta-mo-nos a referir à sua versão 3.

Para verificar a versão Python que está a usar, abra no IDLE. A versão Python é indicada como aqui:



The image shows a screenshot of the Python IDLE shell window. The menu bar at the top includes 'File', 'Edit', 'Shell', 'Debug', 'Options', 'Window', and 'Help'. The main text area displays the following information: 'Python 3.7.9 (default, Oct 12 2020, 18:06:16)', '[GCC 8.3.1 20191121 (Red Hat 8.3.1-5)] on linux', and 'Type "help", "copyright", "credits" or "license()" for more information.' Below this, the prompt '>>>' is followed by a vertical bar '|'. A red circle is drawn around the version number '3.7.9'. A red arrow points from the handwritten text 'VERSÃO 3.7.9' to the circled version number. The status bar at the bottom right indicates 'Ln: 4 Col: 4'.

```
File Edit Shell Debug Options Window Help
Python 3.7.9 (default, Oct 12 2020, 18:06:16)
[GCC 8.3.1 20191121 (Red Hat 8.3.1-5)] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>> |
```

VERSÃO 3.7.9

Ln: 4 Col: 4

Tipicamente a versão Python tem três números separados por pontos. Aquilo que distingue a versão 2 da 3 é o primeiro número. Para executar os programas deste livro, pode usar-se qualquer uma das versões 3.

Hello World

O objetivo deste capítulo é ilustrar os vários temas abordados neste livro, usando exemplos. O contexto geral que escolhemos para fazer esta ilustração é o dinheiro. Escolhemos este contexto porque, por um lado, é bastante rico. Presta-se à construção de exemplos que usam todos os aspetos Python que vamos abordar. Por outro lado, por ser universal. O dinheiro interessa a toda a gente! :-)

Hello `print` and strings

Nesta secção vamos criar o primeiro programa Python. O famoso *Hello World*.

Objetivo Criar um programa que faça *output* da mensagem "Hello World".

Solução No IDLE, faça `ctrl N`. Escreva o programa que se segue. Faça `ctrl S` e escreva `hello.py`, para gravar o programa no ficheiro `hello.py`.

Programa

```
print('Hello World')
```

Faça `F5` para executar o programa. Deverá ter obtido na consola Python o *output* que se segue. Parabéns!!! :-)
Desenvolveu e executou o seu primeiro programa Python. É um primeiro passo no mundo Python. Já está a caminhar no mundo Python! :-)

Output

```
Hello World
```

Como funciona

Neste programa, `print` é uma **função** e `'Hello World'` é um **valor**.

Quando o programa é executado pelo interpretador Python, o valor `'Hello World'` é criado, e passa a existir na memória do computador. De uma forma muito superficial, a função `print` copia este valor para a memória de vídeo do computador. Como o monitor do computador está ligado à memória vídeo, o valor aparece no monitor do computador.

As funções e os valores são dois dos pilares mais importantes da linguagem Python. Assim como de muitas outras linguagens de programação.

Sobre funções

Há partes de programas que são usadas muitas vezes. Por exemplo a cópia de valores para a memória de vídeo do computador, tem que ser efetuada sempre que se quer mostrar mensagens aos utilizadores (fazer *output* de informação). E tipicamente isso é feito muitíssimas vezes.

Uma função é uma parte de um programa, reutilizável. A ideia é a seguinte. Se se quer usar uma parte de um programa muitas vezes, então define-se uma função com essa parte do programa. Sempre que se quer usar essa parte do programa chama-se a função. Em vez de se escrever essa parte muitas vezes. Isto permite poupar muito tempo de desenvolvimento.

Foi isto que foi feito pelos criadores da linguagem Python, relativamente ao *output* de informação. O *output* de informação é usado tantas vezes que foi criada uma função para fazer isso. É a função `print`. Sempre que um programador quer fazer *output*, não precisa de escrever no programa as instruções para copiarem valores para a memória de vídeo do computador. Basta chamar a função `print`, fornecendo os valores que pretender. A função `print` faz isso.

Sobre valores *string*

'Hello World' é um valor. Em Python, os valores são sempre de algum tipo. O tipo do valor 'Hello World' é *string*. Mais precisamente o nome do tipo é `str`, a abreviação de *string*. A palavra inglesa *string* quer dizer corda, ou fio. A ideia é que uma `string` é uma cadeia de caracteres, ou uma sequência de caracteres. Como a mensagem 'Hello World'.

Naturalmente que existem em Python outros tipos de valores, como por exemplo números, que veremos em seguida.

Sobre programas

Um programa Python é que sequência de linhas. As linhas têm instruções Python. O interpretador Python executa as linhas por ordem:

- Primeiro a primeira linha, depois a segunda, depois a terceira, e assim sucessivamente;
- Quando uma linha não tem nenhuma instrução, passa à execução da seguinte;
- Quando chega ao fim das linhas, termina a execução. O programa termina.

Exemplo - no ficheiro `sequencia_linhas.py`

Nota: Já sabe quais são os comandos do IDLE para escrever e executar este programa. São os mesmos que os do exemplo `Hello World`. A única diferença é que desta vez o nome do ficheiro é `sequencia_linhas.py`, em vez de `hello.py`.

```
print('Hello World')  
  
print('Olá Mundo')  
  
print('Hola Mundo')  
  
print('Bonjour le monde')  
  
print('Hallo Welt')  
  
print('नमस्ते दुनिया')  
  
print('你好世界')
```

Output

```
Hello World  
Olá Mundo  
Hola Mundo  
Bonjour le monde  
Hallo Welt  
नमस्ते दुनिया  
你好世界
```


Hello numbers, * and -

Um chocolate custa 1.23 . Está com desconto de 13% .

Objetivo

Escrever e executar um programa, que mostre o valor a pagar pelo chocolate e o valor do desconto. Ambos com duas casas decimais (parte decimal em centavos).

Nota: o separador decimal em Portugal é a vírgula. Mas em informática é usual usar-se o ponto, porque no formato de números decimais, mais usado em computadores, o separador decimal é o inglês, o ponto.

Solução

Escreva, usando o IDLE, o programa que se segue, no ficheiro `desconto.py` . Já sabe quais são os comandos do IDLE. São os mesmos que os dos exemplos anteriores.

Programa - no ficheiro `desconto.py`

```
print(1.23 * (1.0 - 0.13))
print(1.23 * 0.13)
```

Output

```
1.0701
0.159900000000000001
```

Como funciona

Neste programa, 1.23 , 1.0 e 0.13 são **números decimais**, o * e o - são **operadores**. Quando se usam operadores matemáticos com parentesis, os parentesis têm o mesmo significado que em matemática, isto é, conferem precedência.

Em Python (e na grande maioria das linguagens de programação) o asterisco * é o operador de multiplicação, e o sinal menos - o operador de subtração. Naturalmente que existem outros operadores. Por exemplo, o sinal mais + é o operador de soma, e a barra / é o operador de divisão.

O interpretador Python executa em primeiro lugar a primeira linha do programa. Encontra a chamada à função `print` . Mas como dentro dos parentesis mais exteriores está uma expressão matemática ele avalia essa expressão matemática para obter o valor que irá fornecer à função `print` . Ao avaliar a expressão matemática começa por executar `(1.0 - 0.13)` porque os parentesis conferem precedência a esta operação. Em seguida multiplica 1.23 pelo valor obtido, obtendo então o valor que fornece à função `print` . A função `print` faz então *output* do valor 1.0701 . O preço a pagar pelo chocolate.

Em seguida executa a segunda linha do programa. Multiplica 1.23 por 0.13 para obter o valor a fornecer à função `print` . A função `print` faz então *output* do valor 0.159900000000000001 . O valor do desconto obtido.

Observação

- Porque é que o valor 1.0701 tem 4 casas decimais, e o valor 0.159900000000000001 tem 17?

Para responder a esta pergunta, ainda que de uma forma superficial, é preciso notar que o resultado das operações matemáticas pode ser uma dízima infinita, isto é, pode ter infinitas casas decimais. Por exemplo, um a dividir por três é '0.3333333...' . Tem infinitos 3 depois do separador decimal, o ponto. Em matemática usa-se a notação `0.(3)` para simbolizar que se tratam de infinitos 3 . Mas a memória do processador (CPU) que faz as operações matemáticas é necessariamente finita (nenhum computador tem memória infinita. Ainda! :-)). Por isso

Sobre funções

os números são truncados, o que gera um erro de representação que se propaga pelos cálculos. É esse erro que faz com que o valor `0.15990000000000001` tenha 17 casas decimais. Na verdade este erro é muito pequeno, é `0.00000000000000001`, e na maioria dos casos os erros desta ordem de grandeza podem ser desprezados.

Sobre valores `float`

Em Python o tipo dos números decimais é `float`. São as primeiras letras de *floating point*, o que quer dizer vírgula flutuante, ou vírgula deslizante. *Floating point* é o formato de representação de números decimais mais usado pelos processadores dos computadores. Recebeu este nome porque neste formato o separador decimal não está fixo. Desliza consoante o número em causa. É uma forma de otimizar a memória disponível nos processadores para representar números decimais. Permite representar números maiores e mais pequenos do que se o separador decimal estivesse fixo. A ideia é que quando os números são muito grandes as casas decimais não são muito relevantes, e por isso pode-se representar os números usando menos memória para as casas decimais e mais memória para a parte inteira. Se por outro lado os números forem muito pequenos então a parte inteira não é relevante, é zero, e por isso pode-se representar os números usando mais memória para as casas decimais e menos memória para a parte inteira.

Exemplos - no ficheiro `exemplos_float.py`

```
print(1.13 * 123456789.0)
print(1.13 / 123456789.0)
```

Output

```
139506171.57
9.1530000832923e-09
```

Observação

O número `1.13 * 123456789.0` é muito grande, e por isso é representado apenas com duas casas decimais. Já o número `1.13 / 123456789.0` é muito pequeno e por isso é representado com 13 casas decimais.

Tal como já referimos, este formato de vírgula flutuante é muito usado nos processadores dos computadores porque permite aumentar a gama de números decimais que é possível representar com uma quantidade fixa de memória.

Hello round

O programa `desconto.py` não está ainda acabado. O seu *output* é este:

```
1.0701
0.159900000000000001
```

E estes valores não têm duas casas decimais. Como é pretendido.

Problema

Como se obtém estes valores, apenas com duas casas decimais?

Solução

Usando uma outra função, a função `round`. A palavra inglesa *round* quer dizer arredondar. A função `round` recebe dois valores. O primeiro é o número a arredondar. O segundo é o número de casas decimais que se pretende obter. Os dois valores são separados por uma vírgula.

Programa - Segunda versão - no ficheiro `desconto_v2.py`

```
print(round(1.23 * (1.0 - 0.13), 2))
print(round(1.23 * 0.13, 2))
```

Output

```
1.07
0.16
```

Obeservação

No caso deste programa não estamos a fornecer à função `print` um valor. Estamos a fornecer a chamada à função `round`. E isto funciona? Funciona sim! Porque o interpretador Python executa o que está dentro dos parentesis que definem o valor que se fornece à função `print`, os parentesis mais exteriores. Dessa execução obtém um valor. E fornece esse valor à função `print`.

Trata-se de uma possibilidade que proporciona uma grande versatilidade porque permite que em vez de um valor se possa usar uma função que, por sua vez, retorna um valor. E este mecanismo pode ser usado em qualquer parte dos programas.

Hello variáveis

Suponhamos que o preço do chocolate aumentou. Passou a custar `1.33`. E que o desconto diminuiu. Passou a ser `9%`.

Problema

Como obter os novos valores do chocolate e do desconto.

Solução

Copiar o programa no ficheiro `desconto_v2.py` para o ficheiro `desconto_v3.py`, alterar o preço do chocolate e o desconto, e executar.

Programa - no ficheiro `desconto_v3.py`

```
print(round(1.33 * (1.0 - 0.09), 2))
print(round(1.33 * 0.09, 2))
```

Output

```
1.21
0.12
```

Inaceitável! :-)

Porque é que um programador considera esta solução inaceitável?

Porque o preço do chocolate é só um. Então não se devia ter que alterar o programa em dois sítios. Da mesma forma o desconto é só um. E tem que se alterar o programa em mais dois sítios. Isto não faz sentido nenhum! Está errado. Não pode ser assim.

Acrescentámos um *smile* ao título desta secção, porque estamos a exagerar. Neste caso em particular, para alterar cada um dos valores em causa basta alterar o programa em apenas dois sítios (quatro sítios no total). Apesar de não ser desejável, não é assim tão grave. Mas se um valor estivesse espalhado pelo programa dez vezes, ou cem vezes, aí seria totalmente inaceitável. Não queremos, como programadores, ter que alterar o mesmo valor em cem sítios diferentes. Sabemos que nos vamos enganar nalgum sítio e o programa vai ficar errado. Não pode ser!

Solução

Usar variáveis. Uma variável tem um nome e é inicializada com um valor, usando o sinal de igual `=`, que é o operador de atribuição. Sempre que se pretende usar esse valor no programa, pode usar-se o nome da variável.

Programa - no ficheiro `desconto_v4.py`

```
preco    = 1.33
desconto = 9.0

print(round(preco * (1.0 - desconto/100), 2))
print(round(preco * desconto/100, 2))
```

Output

```
1.21
0.12
```

Observações

- Os nomes das variáveis devem ser sugestivos. Por exemplo, ter usado os nomes `x` e `y` para os valores do preço do chocolate e do desconto, seria totalmente desadequado. Porque tornaria o programa difícil de ler e de perceber.
- Escrevemos o `c` em `preco`, sem cedilha. Na verdade poderíamos ter usado o `ç` com cedilha. O interpretador Python suporta os caracteres portugueses ecentuados. Mas por convenção, os nomes das variáveis devem ser escritos usando apenas caracteres ASCII. ASCII é um *standard* de codificação de letras que foi criado para permitir a troca de informação entre diferentes sistemas informáticos. Como foi criada nos EUA, nos primórdios da informática, altura em que a memória dos computadores era muito limitada, não abrange os caracteres portugueses acentuados. Apesar de ser um *standard* antigo continua a ser, nos dias de hoje, aquele que fornece as maiores garantias de interoperabilidade. Por isso foi convencionado pela comunidade Python que os nomes das variáveis devem conter apenas caracteres ASCII. É por esta razão que escrevemos os nomes das variáveis sem caracteres acentuados. Na verdade o que deveríamos fazer era escrever os nomes das variáveis em inglês, `price` e `discount`. Só não o fazemos neste contexto, de uma introdução Python, porque somos portugueses. Em todos os outros contextos escrevemos tudo em inglês.

Sobre variáveis

Nomes das variáveis

Os nomes das variáveis não podem ser quaisquer. Obdecem a algumas regras, nomeadamente:

- Só podem ter letras, algarismos e o sublinhado `_`, o *underscore*, em inglês. Não podem, por exemplo, ter espaços. Nem outros símbolos, como os dos operadores matemáticos, entre outros;
- Não podem começar por um algarismo. Apenas por uma letra ou por um *underscore*;
- Os nomes em Python, em particular os nomes das variáveis, são *case sensitive*, o que quer dizer que as letras minúsculas são consideradas diferentes das maiúsculas correspondentes. Por exemplo, os nomes `desconto` e `Desconto`, são considerados nomes de variáveis diferentes.

Utilização de variáveis

Uma variável só pode ser utilizada depois de ter sido inicializada. A inicialização atribui um valor à variável. Essa atribuição é efetuada com o operador de atribuição, `=`, o sinal de igual.

Do lado esquerdo do sinal de igual fica o nome da variável. Do lado direito do sinal de igual fica um valor, ou código Python cuja execução resulte num valor.

Quando uma variável é utilizada, o interpretador Python substitui o nome da variável pelo seu valor.

Exemplo - no ficheiro `exemplo_variaveis.py`

```
preco_1    = 5.2
preco_2    = 7.3
preco_total = preco_1 + preco_2
print(preco_total)
```

Output

```
12.5
```

Observações

- No caso da inicialização das variáveis `preco_1` e `preco_2`, do lado direito do operador de atribuição, o sinal de igual, estão valores. Neste caso o interpretador Python inicializa estas variáveis com os valores respetivos.
- No caso da variável `preco_total`, do lado direito do sinal de igual não está um valor. Está `preco_1 + preco_2`. Neste caso, o interpretador Python executa o que está do lado direito do sinal de igual. Começa por substituir os nomes das variáveis `preco_1` e `preco_2` pelos seus valores. Depois, efetua a soma desses valores, obtendo o valor dessa soma. Finalmente, inicializa a variável `preco_total` com o valor obtido.
- Por convenção os nomes de variáveis que sejam constituídos por várias palavras devem ser escritos com as palavras em minúsculas, separadas por *underscores*. Por isso escrevemos assim, `preco_total`, o nome da variável que armazena o preço total.

Hello comentários

No programa `desconto_v4.py`, as variáveis `preco` e `desconto` armazenam ambos números decimais. Mas esses números têm significados diferentes. O valor de `preco` é um valor absoluto, enquanto que o valor de `desconto` é em percentagem.

Programa - no ficheiro `desconto_v4.py`

```
preco    = 1.33
desconto = 9.0

print(round(preco * (1.0 - desconto/100), 2))
print(round(preco * desconto/100, 2))
```

Problema

A diferença entre o significado destes valores não é totalmente óbvia. O desconto poderia ser também um valor absoluto, por exmplo, de `0.2`. Isto obriga a termos que verificar como é que o `desconto` está a ser usado para conseguirmos perceber o significado do `desconto`.

Haverá alguma forma de escrever no próprio programa que o `desconto` é em percentagem?

Tentativa

Tentemos escrever no programa que o `desconto` é em percentagem.

Programa - no ficheiro `desconto_v5.py`

```
preco    = 1.33
desconto = 9.0 este desconto é em percentagem

print(round(preco * (1.0 - desconto/100), 2))
print(round(preco * desconto/100, 2))
```

Qual será o resultado da execução deste programa?

Output

```
File "desconto_v5.py", line 2
    desconto = 9.0 este desconto é em percentagem
                ^
SyntaxError: invalid syntax
```

O resultado da execução deste programa é um erro. Porquê? Porque `este desconto é em percentagem` não é código Python válido. O interpretador Python não sabe o que executar quando tenta interpretar esta frase. E por isso dá um erro.

Solução

Esta necessidade, de documentar os programas, é tão frequente e tão importante, que existe uma forma de incluir texto livre nos programas, sem que esse texto livre seja interpretado. De forma a que esse texto livre seja ignorado pelo interpretador.

Usa-se o carater `#`. Tudo o que está depois desse carater, até ao fim da linha, é ignorado.

Programa - no ficheiro `desconto_v6.py`

Sobre funções

```
preco    = 1.33
desconto = 9.0 # este desconto é em percentagem. isto é um comentário

print(round(preco * (1.0 - desconto/100), 2))
print(round(preco * desconto/100, 2))
```

Output

```
1.21
0.12
```

Sobre comentários

Devem usar-se comentários para documentar os programas o mais possível. Facilitando a leitura, o entendimento e, quando necessário, a alteração dos programas. É habitual incluir-se, por exemplo, no início do programa, uma descrição do próprio programa. No caso deste programa poderíamos, por exemplo, acrescentar:

Programa - no ficheiro `desconto_v7.py`

```
# Este progrma usa o preço de um produto e um desconto em
# percentagem, para calcular e mostrar o valor a pagar pelo produto, e
# o valor do desconto.

preco    = 1.33
desconto = 9.0 # este desconto é em percentagem.

print(round(preco * (1.0 - desconto/100), 2))
print(round(preco * desconto/100, 2))
```

Hello funções

Para além do chocolate queremos comprar também uma garrafa de água. O preço da água é 0.78 e está com desconto de 5.5% .

Objetivo

Pretende-se escrever um programa que mostre o valor a pagar pelo chocolate, o valor do desconto do chocolate, o valor a pagar pela água e o valor do desconto da água.

Solução

Copy/paste. Copiar as linhas do programa que mostram os valores relativos ao chocolate, alterar os valores para os valores relativos à água, e executar.

Programa - no ficheiro `desconto_v8.py`

```
# chocolate
preco    = 1.33
desconto = 9.0 # este desconto é em percentagem.

print(round(preco * (1.0 - desconto/100), 2))
print(round(preco * desconto/100, 2))

# água
preco    = 0.78
desconto = 5.5 # este desconto é em percentagem.

print(round(preco * (1.0 - desconto/100), 2))
print(round(preco * desconto/100, 2))
```

Output

```
1.21
0.12
0.74
0.04
```

Observação

As variáveis `preco` e `desconto` foram reinicializadas com os valores correspondentes à água. Isto pode fazer-se? Sim. As variáveis podem ser reinicializadas com novos valores. Isto permite a reutilização de nomes. Depois de uma variável ser reinicializada, será substituída pelo seu novo valor.

Inaceitável! :-)

Põem-se novamente a pergunta: porque é que um programador considera esta solução inaceitável?

Porque a parte mais importante deste programa, que é a parte que faz os cálculos, está repetida. Suponhamos que decidíamos fazer os cálculos de maneira diferente. Por exemplo, calculando primeiro o valor dos descontos, e depois, subtraindo-os aos preços. Se quisermos fazer isto temos que alterar a parte mais importante do programa em dois sítios. E se tivéssemos dez ou cem produtos? Teríamos dez ou cem cópias desta parte do programa? E teríamos que alterar o programa em dez ou cem sítios. É, de facto, inaceitável! Não faz sentido!

Acrescentámos novamente ao título desta secção um *smile* porque continuamos a exagerar um pouco. Não é assim muito grave alterar ao programa em quatro sítios. Mas não é nada desejável.

Solução

Definir uma função que faça os cálculos e mostre os valores que pretendemos. E chamar essa função nos dois casos. No caso do chocolate e no caso da água.

Mas isso é possível? Já usámos funções. A função `print`, por exemplo. Mas podemos definir funções? Para fazerem aquilo que quisermos. Sim, isso mesmo! Podemos definir funções que façam aquilo que quisermos.

Programa - no ficheiro `desconto_v9.py`

```
def mostra_valores(preco, desconto):
    # o argumento desconto é em percentagem

    # resolvemos alterar a forma como calculamos os valores
    valor_desconto = preco * desconto/100
    valor_preco    = preco - valor_desconto
    print(round(valor_preco, 2))
    print(round(valor_desconto, 2))

# chocolate
preco_chocolate    = 1.33
desconto_chocolate = 9.0 # este desconto é em percentagem.
mostra_valores(preco_chocolate, desconto_chocolate)

# água
preco_agua         = 0.78
desconto_agua      = 5.5 # este desconto é em percentagem.
mostra_valores(preco_agua, desconto_agua)
```

Output

```
1.21
0.12
0.74
0.04
```

Observação

Uma função é uma parte de um programa. Reutilizável. Mas ao mesmo tempo está num programa. Como é que se distingue a parte do programa que constitui a função, do restante programa? A parte do programa que constitui a função está indentada com 4 espaços. Portanto, no caso deste programa, a linha `print(round(valor_desconto, 2))` faz parte da função. Mas a linha `preco_chocolate = 1.33` já não faz parte da função.

Como funciona

Alterámos os nomes das variáveis que armazenam os preços e os descontos, para podermos detalhar o mecanismo de chamada às funções de uma forma mais clara.

Quando o interpretador Python executa `mostra_valores(preco_chocolate, desconto_chocolate)`, substitui os nomes das variáveis pelos seus valores. Apesar de nós não o vermos, quando o programa é executado, a chamada à função é efetuada assim: `mostra_valores(1.33, 9.0)`. Antes de executar o código da função, o código que está indentado, o interpretador Python cria as variáveis `preco` e `desconto`, respetivamente, com os valores `1.33` e `9.0`. A partir daí executa o código da função e usa o mecanismo de substituição de variáveis por valores, sempre que encontra uma variável. Terminada a execução do código da função, retoma a execução do programa que chamou a função, na linha imediatamente a seguir à da chamada à função. Portanto, depois de executada a chamada `mostra_valores(preco_chocolate, desconto_chocolate)`, executa a linha `preco_agua = 0.78`. E depois as restantes linhas do programa.

Sobre a definição de funções

Estrutura

A estrutura da definição de funções é:

```
def nome_da_funcao(argumento_1, argumento_2): # cabeçalho da função

    instrucao_1      # isto é
    instrucao_2      # o corpo
    # outras instruções # da
    instrucao_n      # função
```

A definição de uma função tem duas partes: o cabeçalho da função, e o corpo da função.

Cabeçalho

- O cabeçalho começa com a palavra `def`. São as primeiras três letras da palavra inglesa *define*, que quer dizer "definir".
- Depois tem um espaço.
- Depois tem o nome da função. Os nomes das funções obedecem às mesmas regras dos nomes das variáveis. Na verdade, tanto os nomes das variáveis como os nomes das funções, são identificadores Python. E as regras que mencionámos para os nomes das variáveis aplicam-se aos identificadores Python, em geral.
- Depois tem uma sequência de nomes de variáveis, separadas por vírgulas, entre parentesis curvos. Esta sequência é a sequência de argumentos da função. Define os nomes pelos quais são conhecidos, no corpo da função, os valores fornecidos à função. O número de argumentos pode ser qualquer um. Mesmo que uma função não tenha argumentos tem que ser definida com os parentesis curvos. Neste caso, o interior dos parentesis fica vazio.
- Finalmente, tem o caracter dois pontos, `:`. Estes dois pontos são obrigatórios.

Exemplos

```
# cabeçalho de função com dois argumentos
def mostra_valores(preco, desconto):

# cabeçalho de função sem argumentos
def hello():
```

Corpo

- O corpo da função tem que estar indentado com 4 espaços, relativamente ao cabeçalho da função.
- O código do corpo da função pode usar as variáveis definidas na sequência de argumentos da função.

Exemplos

```
# corpo da função mostra_valores
    valor_desconto = preco * desconto/100
    valor_preco    = preco - valor_desconto
    print(round(valor_preco, 2))
    print(round(valor_desconto, 2))

# corpo da função hello
    print('Hello World')
```

Chamada a funções

- Quando se pretende executar uma função, diz-se que se chama a função.
- A chamada a uma função faz-se escrevendo o nome da função, e colocando entre parentesis curvos os valores a fornecer à função, separados por vírgulas. O número de valores fornecidos tem que ser igual ao número de argumentos com que a função foi definida.
- Uma função só pode ser chamada depois de ter sido definida. Se for chamada sem ter sido definida, o interpretador Python dá erro. Quando executa o programa e encontra a chamada a uma função que não conhece. Não sabe que código executar e por isso dá erro.

Exemplos

```
mostra_valores(0.78, 5.5)

hello()
```

Exemplos completos - no ficheiro `exemplos_funcoes.py`

```
# definição de uma função com dois argumentos
def mostra_valores(preco, desconto):

    valor_desconto = preco * desconto/100
    valor_preco    = preco - valor_desconto
    print(round(valor_preco, 2))
    print(round(valor_desconto, 2))

# definição de uma função sem argumentos
def hello():

    print('Hello World')

# chamada a uma função sem argumentos
hello()

# chamada a uma função com dois argumentos
mostra_valores(0.78, 5.5)
```

Output

```
Hello World
0.74
0.04
```

Funções *Built-In*

- Na verdade já por várias vezes chamámos funções. Por exemplo com um argumento: `print('Hello World')` . E com dois argumentos: `round(1.23 * 0.13, 2)` . Estas funções são tão úteis, que vêm incorporadas no próprio interpretador Python. Já estão definidas. Chamam-se funções *built-in*. Estão prontas a usar. Por isso é que apesar de não as termos definido nos nossos programas, usámo-las sem que isso tivesse dado erro.

Hello `if` `else`

A promoção dos chocolates mudou. O preço é agora `1.27` . Comprando um chocolate o desconto é de `7%` . Mas comprando dois o desconto é de `11%` .

Objetivo

Pretende-se escrever um programa que mostre o valor a pagar por um chocolate, caso se compre só um, ou caso se comprem dois.

Solução

Usar a instrução de execução condicional `if` . A palavra inglesa *if* quer dizer "se". Um `if` permite executar um dado bloco de código, **se** uma determinada condição for verdadeira. Opcionalmente um `if` pode ter um `else` . *Else* em português quer dizer "senão". A parte opcional `else` permite executar outro bloco de código caso a condição seja falsa. A ideia é: se uma condição é verdadeira executa-se um bloco de código, senão executa-se outro.

Programa - no ficheiro `desconto_quantidade.py`

```
def mostra_valor(preco, desconto):
    # o argumento desconto é em percentagem

    # resolvemos alterar a forma como calculamos os valores
    valor_desconto = preco * desconto/100
    valor_preco = preco - valor_desconto
    print(round(valor_preco, 2))

preco_chocolate = 1.27
desconto_1 = 7.0 # desconto em percentagem para quantidade = 1
desconto_2 = 11.0 # desconto em percentagem para quantidade = 2

quantidade = 1

if quantidade == 1:
    desconto = desconto_1
    mensagem = 'na compra de 1 chocolate o valor a pagar é:'
else:
    desconto = desconto_2
    mensagem = 'na compra de 2 chocolates o valor a pagar por cada um é:'

print(mensagem)
mostra_valor(preco_chocolate, desconto)
```

Execução com `quantidade = 1` .

Output

```
na compra de 1 chocolate o valor a pagar é:
1.18
```

Execução com `quantidade = 2` .

Output

```
na compra de 2 chocolates o valor a pagar por cada um é:
1.13
```

Observações

- O que é o sinal `==`, na condição `quantidade == 1`? É operador de igualdade. A condição `quantidade == 1` é verdadeira se o valor da variável `quantidade` for igual a `1`. É falsa se o valor da variável `quantidade` for igual a `2`. Intuitivamente o sinal que aqui estaríamos à espera de encontrar seria o sinal de igual, `=`. Mas o sinal de igual já está ocupado. É o operador de atribuição. Serve para atribuir valores a variáveis. Por isso o criador da linguagem Python decidiu usar como alternativa o sinal `==`.
- Os números `1` e `2`, atribuídos à variável `quantidade`, não têm parte decimal, porque são números inteiros. Não se vendem chocolates aos bocados. Não se pode comprar `1.2` ou `1.3` chocolates. Não faz sentido usar um número decimal para a quantidade de chocolates.

Como funciona

Quando a variável `quantidade` é inicializada com o valor `1`, a condição `quantidade == 1` é verdadeira, e por isso é executado o bloco `if`. Portanto a variável `desconto` é inicializada com o valor do argumento `desconto_1`. E a variável `mensagem` é inicializada com a mensagem de compra de um chocolate.

Quando a variável `quantidade` é inicializada com o valor `2`, a condição `quantidade == 1` é falsa, e por isso é executado o bloco `else`. Portanto a variável `desconto` é inicializada com o valor do argumento `desconto_2`. E a variável `mensagem` é inicializada com a mensagem de compra de dois chocolates.

Sobre `if` `else`

Estrutura

A estrutura da instrução `if` `else` é:

```
if condicao:
    instrucao_1      #
    instrucao_2      # bloco 1
    outras_instrucoes #
    instrucao_m      #
else:
    instrucao_m_mais_1 #
    instrucao_m_mais_2 # bloco 2
    outras_instrucoes #
    instrucao_n      #
```

- A condição termina com dois pontos, `:`. Estes dois pontos são obrigatórios.
- Os blocos de código `bloco 1` e `bloco 2` distinguem-se do restante programa porque estão indentados com 4 espaços relativamente às palavras `if` e `else`, respetivamente.
- Os dois pontos, `:`, depois da palavra `else`, são obrigatórios.
- A condição é avaliada. Se for verdadeira, é executado o bloco de código `bloco 1`. Se for falsa, é executado o bloco de código `bloco 2`.
- O `else`, e o respetivo bloco de código, `bloco 2`, são opcionais. A condição `if` poderia, por exemplo, ser implementada assim:

```
# assume-se que a quantidade é 1
desconto = desconto_1
mensagem = 'na compra de 1 chocolate o valor a pagar é:'

if quantidade == 2: # se a quantidade for 2
    # altera-se o desconto e a mensagem
    desconto = desconto_2
    mensagem = 'na compra de 2 chocolates o valor a pagar por cada um é:'
```

Sobre valores `int`

Em Python, os números inteiros são do tipo `int`, que é a abreviatura de *integer*. A palavra inglesa *integer* quer dizer "inteiro". Ou, neste contexto, "número inteiro".

Em Python distingue-se os números decimais dos números inteiros. Quando se escreve um número decimal tem que se incluir o separador decimal, o ponto. Quando se escreve um número inteiro não se pode escrever o separador decimal.

Exemplos - no ficheiro `exemplos_numeros.py`

```
preco_1 = 1.2      # isto é um número decimal
preco_2 = -1.0     # isto é um número decimal negativo
quantidade_1 = 10  # isto é um número inteiro
quantidade_2 = -20 # isto é um número inteiro negativo
print(preco_1)
print(preco_2)
print(quantidade_1)
print(quantidade_2)
```

Output

```
1.2
-1.0
10
-20
```

A distinção entre números decimais e números inteiros existe, essencialmente por duas razões.

- Os cálculos numéricos são efetuados pelo processador do computador. Os algoritmos de cálculo com números inteiros, são substancialmente diferentes dos algoritmos de cálculo com números decimais. Relembremos, por exemplo, a divisão inteira e a divisão decimal. O algoritmo que aprendemos na escola para fazer a divisão inteira era substancialmente diferente do da divisão decimal. Por isso há partes dos processadores dedicadas aos cálculos com números decimais e outras partes dedicadas aos cálculos com números inteiros. Têm características diferentes, por exemplo, em termos de velocidade de cálculo. E por isso é importante que se faça esta distinção.
- Por outro lado, há contextos onde os números decimais não fazem sentido. Por exemplo, a lotação de um supermercado é um número inteiro. São, por exemplo `10` ou `20` pessoas. Não faz sentido serem `10.5` pessoas. Não faz sentido.

Hello == e !=

Operador de igualdade ==

O sinal `==` é o operador de comparação de igualdade. É usado para definir a condição de igualdade entre dois valores. Por exemplo, `a == b`. Quando o interpretador Python avalia esta condição, em primeiro lugar substitui as variáveis `a` e `b` pelos seus valores. Se os valores forem iguais a condição é verdadeira. Se os valores não forem iguais a condição é falsa.

Exemplos - no ficheiro `exemplos_de_igualdade.py`

```
preco_1 = 1.0
preco_2 = 1.0
preco_3 = 1.1
if preco_1 == preco_2:
    print('os preços 1 e 2 são iguais')
else:
    print('os preços 1 e 2 são diferentes')
if preco_2 == preco_3:
    print('os preços 2 e 3 são iguais')
else:
    print('os preços 2 e 3 são diferentes')
```

Output

```
os preços 1 e 2 são iguais
os preços 2 e 3 são diferentes
```

Operador de desigualdade !=

Em muitas situações a condição que queremos avaliar é a da desigualdade entre valores.

Por exemplo, quando se altera uma *password*, é típico solicitar-se a confirmação da *password* pretendida. Se a *password* for diferente da confirmação, solicita-se que o utilizador repita o processo.

Em Python o operador de comparação de desigualdade é o sinal `!=`. É usado para definir a condição de desigualdade entre dois valores. Por exemplo, `a != b`. Se os valores de `a` e `b` forem iguais a condição é falsa. Se os valores não forem iguais a condição é verdadeira.

Exemplos - no ficheiro `exemplos_de_desigualdade.py`

```
print('password 1')
password = 'xpto'
confirmacao = 'xpt0'
if password != confirmacao:
    print('por favor, repita o processo')
else:
    print('password alterada com sucesso')
print('password 2')
password = '1234'
confirmacao = '1234'
if password != confirmacao:
    print('por favor, repita o processo')
else:
    print('password alterada com sucesso')
```

Output

Sobre funções

```
password 1  
por favor, repita o processo  
password 2  
password alterada com sucesso
```

Hello list e for

A listagem que se segue, tem preços e descontos de dez produtos.

Listagem

```
preço    1.23 7.08 3.00 0.53 2.01 4.44 9.99 2.50 0.20 7.70
desconto 8.0  5.5  3.0  3.3  9.0  9.9  2.0  2.0  5.0  5.0
```

Obejtivo

Escrever um programa que mostre o valor a pagar por cada um dos produtos e o valor do desconto respetivo.

Solução - no ficheiro `listagem.py`

```
def mostra_valores(preco, desconto):
    valor_desconto = preco * desconto/100
    valor_a_pagar = preco - valor_desconto
    print('-----')
    print('preço')
    print(preco)
    print('desconto')
    print(desconto)
    print('valor a pagar')
    print(round(valor_a_pagar, 2))
    print('valor do desconto')
    print(round(valor_desconto, 2))

preco = 1.23
desconto = 8.0
mostra_valores(preco, desconto)

preco = 7.08
desconto = 5.5
mostra_valores(preco, desconto)

preco = 3.00
desconto = 3.0
mostra_valores(preco, desconto)

preco = 0.53
desconto = 3.3
mostra_valores(preco, desconto)

preco = 2.01
desconto = 9.0
mostra_valores(preco, desconto)

preco = 4.44
desconto = 9.9
mostra_valores(preco, desconto)

preco = 9.99
desconto = 2.0
mostra_valores(preco, desconto)

preco = 2.50
desconto = 2.0
mostra_valores(preco, desconto)

preco = 0.20
desconto = 5.0
mostra_valores(preco, desconto)

preco = 7.70
desconto = 5.0
mostra_valores(preco, desconto)
```

Output

```
-----  
preço  
1.23  
desconto  
8.0  
valor a pagar  
1.13  
valor do desconto  
0.1  
-----  
preço  
7.08  
desconto  
5.5  
valor a pagar  
6.69  
valor do desconto  
0.39  
-----  
preço  
3.0  
desconto  
3.0  
valor a pagar  
2.91  
valor do desconto  
0.09  
-----  
preço  
0.53  
desconto  
3.3  
valor a pagar  
0.51  
valor do desconto  
0.02  
-----  
preço  
2.01  
desconto  
9.0  
valor a pagar  
1.83  
valor do desconto  
0.18  
-----  
preço  
4.44  
desconto  
9.9  
valor a pagar  
4.0  
valor do desconto  
0.44  
-----  
preço  
9.99  
desconto  
2.0  
valor a pagar  
9.79  
valor do desconto  
0.2  
-----  
preço  
2.5  
desconto  
2.0
```

```
valor a pagar
2.45
valor do desconto
0.05
-----
preço
0.2
desconto
5.0
valor a pagar
0.19
valor do desconto
0.01
-----
preço
7.7
desconto
5.0
valor a pagar
7.32
valor do desconto
0.39
```

Inaceitável

Aqui não acrescentamos nenhum *smile* porque esta solução é de facto inaceitável. Imagine-se o que seria este programa para vinte produtos. Terias mais quarenta linhas (quatro por produto). Para cem produtos seria completamente impossível. Quatrocentas linhas! Impossível! Esta solução é inaceitável.

Solução

Usar a estrutura de dados `list`, para armazenar os preços e os respetivos descontos. Percorrer as listas com a instrução de execução em ciclo `for`. Um ciclo `for`.

Programa - no ficheiro `listagem_for.py`

```
def mostra_valores(preco, desconto):
    valor_desconto = preco * desconto/100
    valor_a_pagar = preco - valor_desconto
    print('-----')
    print('preço')
    print(preco)
    print('desconto')
    print(desconto)
    print('valor a pagar')
    print(round(valor_a_pagar, 2))
    print('valor do desconto')
    print(round(valor_desconto, 2))

precos = [1.23, 7.08, 3.00, 0.53, 2.01, 4.44, 9.99, 2.50, 0.20, 7.70]
descontos = [8.0, 5.5, 3.0, 3.3, 9.0, 9.9, 2.0, 2.0, 5.0, 5.0]

for indice in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]:
    preco = precos[indice]
    desconto = descontos[indice]
    mostra_valores(preco, desconto)
```

Output

Não incluímos aqui o *output* porque é bastante extenso e é exatamente igual ao *output* da solução inicial inaceitável.

Como funciona

Sobre funções

A variável `precos` é uma lista de preços. A variável `descontos` é uma lista de descontos. Para criar uma lista colocam-se os seus valores, separados por vírgulas, entre parentesis retos.

O bloco de instruções executadas pelo ciclo `for` está indentado com 4 espaços relativamente à palavra `for`.

O ciclo `for` inicializa a variável `indice` com o primeiro valor da lista `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`, isto é, com o valor `0`. E executa o bloco de instruções. Em seguida reinicializa a variável `indice` com o segundo valor da lista, ou seja, com o valor `1`. E repete a execução do bloco de instruções. Repete este processo sequencialmente com cada um dos valores da lista. Até esgotar os valores da lista.

Os valores das listas `precos` e `descontos` são obtidos indexando essas listas. Para indexar uma lista usa-se o nome da lista, seguido de parentesis retos. Dentro dos parentesis retos coloca-se o índice do elemento pretendido. Os índices iniciam-se zero.

Observação

O programa `listagem_for.py` tem 20 linhas. O programa `listagem.py` tem 52. Fazem os dois exatamente a mesma coisa. O *output* de ambos é exatamente igual. Qual é o programa que prefere?

Sobre listas

As listas são valores do tipo `list`. A palavra inglesa *list* quer dizer "lista". Os valores do tipo `list` são estruturas de dados. Servem para armazenar sequências de valores. Os valores armazenados numa lista não têm que ser todos do mesmo tipo. Podem ser de tipos diferentes.

Criar uma lista

Para criar uma lista colocam-se os elementos da lista, separados por vírgulas, entre parentesis retos.

Exemplo

```
precos = [1.23, 7.08, 3.00, 0.53, 2.01, 4.44, 9.99, 2.50, 0.20, 7.70]
```

Indexar

Para se obter um valor armazenado numa lista, usa-se o seu índice na lista, e indexa-se a lista com esse índice. Para indexar uma lista coloca-se o nome da lista seguido de parentesis retos. Dentro dos parentesis retos coloca-se o índice do elemento pretendido. Os índices iniciam-se em zero. O índice do primeiro valor de uma lista é zero. O do segundo é um. O do terceiro é dois, e assim por diante.

Exemplo

```
print(precos[0])
```

Alterar elemento

As listas podem ser alteradas. Para modificar um elemento de uma lista usa-se o operador de atribuição. O sinal de igual. Do lado esquerdo coloca-se a lista indexada no elemento que se pretende alterar. Do lado direito o novo valor do elemento em causa.

Exemplo

```
precos[2] = 'esgotado'
```

Erro

Se se tentar indexar uma lista numa posição onde não existe nenhum elemento, o interpretador dá erro.

Exemplo

```
# o décimo primeiro elemento não existe  
print(outra_lista[10]) # esta linha dá erro
```

Exemplo completo - no ficheiro `exemplo_lista.py`

Sobre funções

```
# precos é uma lista
precos = [1.23, 7.08, 3.00, 0.53, 2.01, 4.44, 9.99, 2.50, 0.20, 7.70]

print(precos[0])           # este é o primeiro elemento
indice = 1
print(precos[indice])      # este é o segundo elemento
indice = 2
print(precos[indice])      # este é o terceiro elemento
print(precos[3])           # este é o quarto elemento

# alterar o terceiro elemento
precos[2] = 'esgotado'

print('aqui está a lista completa')
print(precos)

# o décimo primeiro elemento não existe
print(precos[10]) # esta linha dá erro
```

Output

```
1.23
7.08
3.0
0.53
aqui está a lista completa
[1.23, 7.08, 'esgotado', 0.53, 2.01, 4.44, 9.99, 2.5, 0.2, 7.7]
Traceback (most recent call last):
  File "/home/jbs/develop/202209_hello_python_world/pt/hello_world/hello_list_e_for/exemplo_lista.py", line 19,
    print(precos[10])
IndexError: list index out of range
```

Observação

Saliente-se que o interpretador Python executa todo o programa até à linha que dá erro. Quando não é possível executar uma dada linha, o interpretador interrompe a execução do programa nessa linha e dá erro. É o caso aqui. Não é possível obter o décimo primeiro elemento de uma lista que tem dez elementos.

Sobre o ciclo `for`

A estrutura do ciclo `for` é:

```
for variavel in sequencia:
    instrucao_1      # um
    instrucao_2      # bloco
    outras_instrucoes # de
    instrucao_n      # instruções
```

- O ciclo `for` usa uma variável e uma sequência de valores.
- A palavra `in` entre a variável e a sequência é obrigatória.
- O dois pontos `:` depois da sequência é obrigatório.
- Quando o interpretador Python executa um ciclo `for`, inicializa a variável `variavel` com o primeiro valor da sequência `sequencia`, e executa o bloco de instruções.
- Em seguida reinicializa a variável `variavel` com o segundo valor da sequência `sequencia`, e volta a executar o bloco de instruções.
- Faz isto sucessivamente até percorrer todos os valores da sequência `sequencia`.
- As instruções do bloco executado pelo ciclo `for`, distinguem-se das restantes instruções do programa porque estão indentadas com 4 espaços relativamente à palavra `for`.

Exemplo - no ficheiro `exemplo_for.py`

```
precos = [1.23, 7.08, 3.00, 0.53, 2.01, 4.44, 9.99, 2.50, 0.20, 7.70]
for preco in precos:
    print('preço = ')
    print(preco)
print('fim dos preços') # esta linha está fora do ciclo for. só é executada uma
                        # vez. depois do ciclo for ter terminado.
```

Output

```
preço =
1.23
preço =
7.08
preço =
3.0
preço =
0.53
preço =
2.01
preço =
4.44
preço =
9.99
preço =
2.5
preço =
0.2
preço =
7.7
fim dos preços
```

Hello `range`

Voltemos a analisar o ciclo `for` no programa no ficheiro `listagem_for.py`.

```
def mostra_valores(preco, desconto):
    valor_desconto = preco * desconto/100
    valor_a_pagar = preco - valor_desconto
    print('-----')
    print('preço')
    print(preco)
    print('desconto')
    print(desconto)
    print('valor a pagar')
    print(round(valor_a_pagar, 2))
    print('valor do desconto')
    print(round(valor_desconto, 2))

precos = [1.23, 7.08, 3.00, 0.53, 2.01, 4.44, 9.99, 2.50, 0.20, 7.70]
descontos = [8.0, 5.5, 3.0, 3.3, 9.0, 9.9, 2.0, 2.0, 5.0, 5.0]

for indice in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]:
    preco = precos[indice]
    desconto = descontos[indice]
    mostra_valores(preco, desconto)
```

Inaceitável

Poderíamos dizer que é inaceitável termos que escrever a lista `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`. Porque se as listas de preços e descontos tivessem milhares de elementos, teríamos que escrever milhares de números nesta lista. Para podermos indexar as listas de preços e de descontos. E ter que escrever uma lista com milhares de números, é inaceitável.

A função `range`

Por esta razão e também porque a indexação de listas é muitíssimo frequente, existe uma função para gerar sequências, que podem ser usadas para indexar listas. É a função `range`. A palavra inglesa *range* quer dizer "gama". Neste caso uma gama de valores. Em muitas situações a utilização de ciclos `for` está associada à utilização da função `range`.

A função recebe como argumento um número inteiro, `n`. E gera a sequência de `0` até `n-1`. É exatamente a gama de índices apropriada para indexar uma lista com `n` elementos.

Exemplos - no ficheiro `exemplo_range.py`

```
for n in range(10):
    print(n)
print('-----') # isto é um separador
precos = [1.23, 7.08, 3.00, 0.53, 2.01, 4.44, 9.99, 2.50, 0.20, 7.70]
for n in range(10):
    print(precos[n])
```

Output

```
0
1
2
3
4
5
6
7
8
9
-----
1.23
7.08
3.0
0.53
2.01
4.44
9.99
2.5
0.2
7.7
```

Nova versão - no ficheiro `listagem_for_v2.py`

```
def mostra_valores(preco, desconto):
    valor_desconto = preco * desconto/100
    valor_a_pagar = preco - valor_desconto
    print('-----')
    print('preço')
    print(preco)
    print('desconto')
    print(desconto)
    print('valor a pagar')
    print(round(valor_a_pagar, 2))
    print('valor do desconto')
    print(round(valor_desconto, 2))

precos = [1.23, 7.08, 3.00, 0.53, 2.01, 4.44, 9.99, 2.50, 0.20, 7.70]
descontos = [8.0, 5.5, 3.0, 3.3, 9.0, 9.9, 2.0, 2.0, 5.0, 5.0]

for indice in range(10):
    preco = precos[indice]
    desconto = descontos[indice]
    mostra_valores(preco, desconto)
```

Output

Não repetimos aqui o *output* do programa `listagem_for_v2.py` porque é bastante extenso, e é exatamente igual ao do programa `listagem_for.py`.

Inaceitável

Naturalmente podemos continuar a considerar esta solução inaceitável porque se as listas de preços e de descontos tiverem milhares de elementos, teremos que os continuar a escrever a todos. E escrever uma lista com milhares de valores é inaceitável. Calma! :-)

Resolveremos este problema mais à frente.

Hello len

O último preço e o respetivo desconto, das listagens no ficheiro `listagem_for.py` foram removidos, porque o produto foi descontinuado.

Nova versão - no ficheiro `listagem_for_v3.py`

```
def mostra_valores(preco, desconto):
    valor_desconto = preco * desconto/100
    valor_a_pagar = preco - valor_desconto
    print('-----')
    print('preço')
    print(preco)
    print('desconto')
    print(desconto)
    print('valor a pagar')
    print(round(valor_a_pagar, 2))
    print('valor do desconto')
    print(round(valor_desconto, 2))

# o último elemento desta duas listas foi removido
precos = [1.23, 7.08, 3.00, 0.53, 2.01, 4.44, 9.99, 2.50, 0.20]
descontos = [8.0, 5.5, 3.0, 3.3, 9.0, 9.9, 2.0, 2.0, 5.0]

for indice in range(10):
    preco = precos[indice]
    desconto = descontos[indice]
    mostra_valores(preco, desconto)
```

Como é que termina este programa?

Com um erro! Porquê? Porque não atualizámos o argumento da função `range`. Se a lista agora só tem 9 elementos o argumento da função `range` tem que ser `9`. Se for `10`, os índices gerados vão de `0` a `9`. E como os índices das listas se iniciam em zero, o índice `9` serve para indexar o décimo elemento. E como ele não existe, o interpretador dá erro.

Output

```
-----
preço
1.23
desconto
8.0
valor a pagar
1.13
valor do desconto
0.1
-----
preço
7.08
desconto
5.5
valor a pagar
6.69
valor do desconto
0.39
-----
preço
3.0
desconto
3.0
valor a pagar
2.91
valor do desconto
0.09
-----
preço
0.53
desconto
3.3
valor a pagar
0.51
valor do desconto
0.02
-----
preço
2.01
desconto
9.0
valor a pagar
1.83
valor do desconto
0.18
-----
preço
4.44
desconto
9.9
valor a pagar
4.0
valor do desconto
0.44
-----
preço
9.99
desconto
2.0
valor a pagar
9.79
valor do desconto
0.2
-----
preço
2.5
desconto
2.0
```


Sobre funções

```
valor a pagar
2.45
valor do desconto
0.05
-----
preço
0.2
desconto
5.0
valor a pagar
0.19
valor do desconto
0.01
Traceback (most recent call last):
  File "/home/jbs/develop/202209_hello_python_world/pt/hello_world/hello_len/listagem_for_v3.py", line 19, in
    preco = precos[indice]
IndexError: list index out of range
```

Solução

Naturalmente, que a solução passa por corrigir o argumento da função `range`. Em vez de `10`, tem que passar a ser `9`.

Mas o que seria interessante seria determinar o argumento da função `range` diretamente a partir do número de elementos das lista `precos`, por exemplo.

E isto é possível. Usando a função `len`. `len` é abreviação da palavra inglesa *length*, que quer dizer "comprimento".

Programa - no ficheiro `listagem_for_v4.py`

```
def mostra_valores(preco, desconto):
    valor_desconto = preco * desconto/100
    valor_a_pagar = preco - valor_desconto
    print('-----')
    print('preço')
    print(preco)
    print('desconto')
    print(desconto)
    print('valor a pagar')
    print(round(valor_a_pagar, 2))
    print('valor do desconto')
    print(round(valor_desconto, 2))

# o último elemento desta duas listas foi removido
precos = [1.23, 7.08, 3.00, 0.53, 2.01, 4.44, 9.99, 2.50, 0.20]
descontos = [8.0, 5.5, 3.0, 3.3, 9.0, 9.9, 2.0, 2.0, 5.0]

for indice in range(len(precos)):
    preco = precos[indice]
    desconto = descontos[indice]
    mostra_valores(preco, desconto)
```

Output

Não inserimos aqui o *output* do programa `listagem_for_v4.py` porque é bastante extenso, e é idêntico ao do programa `listagem_for_v3.py`, mas sem o erro final.

Observação

Quando se percorrem os elementos de uma lista com um ciclo `for`, é muito frequente usarem-se também as funções `range` e `len`, desta forma:

```
for variavel in range(len(uma_lista)):
```

Como funciona?

A função `len` retorna o número de elementos de qualquer valor Python que tenha elementos. Listas, como já vimos, mas também *strings*, entre outros tipos que veremos mais à frente.

Exemplos - no ficheiro `exemplos_len.py`

```
mensagem = 'Hello World'
precos = [1.23, 7.08, 3.00, 0.53, 2.01, 4.44, 9.99, 2.50, 0.20]
print('o número de carateres da string mensagem é:')
print(len(mensagem))
print('o número de elementos da lista precos é:')
print(len(precos))
```

Output

```
o número de carateres da string mensagem é:
11
o número de elementos da lista precos é:
9
```

Observação

Note-se que o espaço entre `Hello` e `World`, conta, naturalmente, como um carácter, porque é um carater.

Hello `return` e `append`

Considere-se novamente as listagens de preços e de descontos:

```
precos      = [1.23, 7.08, 3.00, 0.53, 2.01, 4.44, 9.99, 2.50, 0.20, 7.70]
descontos   = [8.0 , 5.5 , 3.0 , 3.3 , 9.0 , 9.9 , 2.0 , 2.0 , 5.0 , 5.0]
```

Objetivo

Pretende-se escrever um programa que construa uma nova lista com os valores a pagar pelos diferentes produtos. Pretende-se que o programa faça *output* das três listas. E nada mais. Assim:

```
preços:
[1.23, 7.08, 3.0, 0.53, 2.01, 4.44, 9.99, 2.5, 0.2, 7.7]
descontos:
[8.0, 5.5, 3.0, 3.3, 9.0, 9.9, 2.0, 2.0, 5.0, 5.0]
valores a pagar:
[1.13, 6.69, 2.91, 0.51, 1.83, 4.0, 9.79, 2.45, 0.19, 7.32]
```

Observação

A função `mostra_valores`, no ficheiro `listagem_for_v4.py` (e anteriores), não é adequada a esta situação. Porque esta função faz ela própria *output* dos valores que calcula. Neste caso pretende-se calcular todos os valores sem fazer qualquer *output*, e armazená-los numa lista. E só depois se pretende fazer *output*.

Solução

Criar uma nova lista `valores_a_pagar`. Definir uma nova função que receba um preço e um desconto, calcule o valor a pagar, e retorne o valor calculado. Usar a instrução `return` para retornar o valor calculado. Percorrer as listas de preços e de descontos com um ciclo `for`. Calcular cada um dos valores a pagar com a nova função e adicioná-los um a um à lista `valores_a_pagar`. Usar a função `append` para adicionar um valor à lista.

Programa - no ficheiro `listagem_valores_a_pagar.py`

```
def valor_a_pagar(preco, desconto):

    valor_desconto = preco * desconto/100
    valor_a_pagar = preco - valor_desconto
    resultado      = round(valor_a_pagar, 2)

    return resultado

precos          = [1.23, 7.08, 3.00, 0.53, 2.01, 4.44, 9.99, 2.50, 0.20, 7.70]
descontos       = [8.0 , 5.5 , 3.0 , 3.3 , 9.0 , 9.9 , 2.0 , 2.0 , 5.0 , 5.0]
valores_a_pagar = [] # inicialmente esta lista está vazia

for indice in range(len(precos)):

    preco      = precos[indice]
    desconto   = descontos[indice]
    valor      = valor_a_pagar(preco, desconto)

    valores_a_pagar.append(valor)

print('preços:')
print(precos)
print('descontos:')
print(descontos)
print('valores a pagar:')
print(valores_a_pagar)
```

Output

```
preços:
[1.23, 7.08, 3.0, 0.53, 2.01, 4.44, 9.99, 2.5, 0.2, 7.7]
descontos:
[8.0, 5.5, 3.0, 3.3, 9.0, 9.9, 2.0, 2.0, 5.0, 5.0]
valores a pagar:
[1.13, 6.69, 2.91, 0.51, 1.83, 4.0, 9.79, 2.45, 0.19, 7.32]
```

Como funciona

- A função `valor_a_pagar` recebe um preço e um desconto. Calcula o valor a pagar e arredonda-o com duas casas decimais. Usando a instrução `return`, retorna o valor obtido.
- A lista `valores_a_pagar` é inicializada com uma lista vazia.
- O ciclo `for` percorre as listas de preços e de descontos.
- A função `valor_a_pagar` é chamada para cada preço e desconto respetivo.
- O valor retornado pela função `valor_a_pagar` é armazenado na variável `valor`.
- O valor armazenado na variável `valor` é acrescentado ao fim da listas `valores_a_pagar` usando a função `append`.

Separar a computação do *output*

As funções que escrevemos até agora para calcular valores a pagar e descontos, como por exemplo a função `mostra_valores`, no ficheiro `listagem_for_v2.py`, fazem os cálculos necessários, mas também fazem o *output* dos valores calculados. Fazem duas coisas completamente distintas: cálculos e *output*. Os cálculos efetuados são bastante úteis e provavelmente serão necessários noutros contextos. Mas o *output* é muito específico. Noutros contextos o *output* poderá ter que ser diferente. Como é o caso do objetivo desta secção. Por isso para maximizar a utilidade das funções que escrevemos devemos procurar isolar umas funcionalidades das outras. Uma coisa é o

Sobre funções

cálculo de valores a pagar em produtos com desconto. Isto é muito útil. É desejável que exista uma função para efetuar estes cálculos. Outra coisa é o *output* desses valores. Isso depende do programa que estamos a fazer. É muito específico. Não deve estar misturado com os cálculos.

Sobre `return`

A palavra inglesa *return* quer dizer retorna. Uma função pode retornar um valor. Usando a instrução `return`, seguida do valor a retornar.

Duas observações importantes:

- O valor retornado pode ser armazenado numa variável. Para tal atribui-se à variável a chamada à função. Isto foi feito no exemplo anterior. Na linha:

```
valor = valor_a_pagar(preco, desconto)
```

- A instrução `return` termina a execução da função. O que quer dizer que se a instrução `return` estiver no meio do corpo da função, as instruções Python que se lhe seguem não serão executadas. A instrução `return` interrompe a execução da função.

O exemplo que se segue ilustra este segunda aspeto. Admite-se que o desconto de um produto possa ser `0.0`. Ou seja, o produto não tem desconto. Só se calcula o valor a pagar se o desconto for diferente de `0.0`.

Exemplo - no ficheiro `exemplo_return.py`

```
def valor_a_pagar(preco, desconto):

    if desconto == 0.0:
        print('nada a calcular')
        return preco

    print('a calcular o valor a pagar...')
    valor_desconto = preco * desconto/100
    valor_a_pagar = preco - valor_desconto
    resultado = round(valor_a_pagar, 2)
    print('a calcular o valor a pagar... fim!')

    return resultado

preco_chocolate = 1.23
desconto_chocolate = 8.0
valor_chocolate = valor_a_pagar(preco_chocolate, desconto_chocolate)
print('valor chocolate =')
print(valor_chocolate)

preco_saco = 0.10
desconto_saco = 0.0
valor_saco = valor_a_pagar(preco_saco, desconto_saco)
print('valor saco =')
print(valor_saco)
```

Output

```
a calcular o valor a pagar...
a calcular o valor a pagar... fim!
valor chocolate =
1.13
nada a calcular
valor saco =
0.1
```

Sobre `append`

A palavra inglesa *append* quer dizer "acrescentar". A função `append` serve para acrescentar um elemento no fim de uma lista.

A função `append` só existe no contexto das listas. Para se chamar a função `append` coloca-se o nome da lista, seguindo de um ponto `.`, seguido do nome `append`.

Como é uma função, para ser chamada tem que se colocar a seguir ao seu nome parentesis curvos. O valor a acrescentar coloca-se entre os parentesis curvos.

Isto foi feito no programa no ficheiro `listagem_valores_a_pagar.py`, na linha:

```
valores_a_pagar.append(valor)
```

Exemplo - no ficheiro `exemplo_append.py`

```
lista_de_precos = [] # esta lista está vazia
# as listas podem ter valores de tipos diferentes
# neste caso a lista vai ter strings e floats. poderiam ser outros quaisquer.
print(lista_de_precos)
lista_de_precos.append('lista') # acrescentar uma string
print(lista_de_precos)
lista_de_precos.append('de') # acrescentar uma string
print(lista_de_precos)
lista_de_precos.append('preços') # acrescentar uma string
print(lista_de_precos)
lista_de_precos.append(1.23) # acrescentar um float
print(lista_de_precos)
lista_de_precos.append(7.08) # acrescentar um float
print(lista_de_precos)
lista_de_precos.append(3.00) # acrescentar um float
print(lista_de_precos)
```

Output

```
[]
['lista']
['lista', 'de']
['lista', 'de', 'preços']
['lista', 'de', 'preços', 1.23]
['lista', 'de', 'preços', 1.23, 7.08]
['lista', 'de', 'preços', 1.23, 7.08, 3.0]
```

Hello operadores lógicos e de comparação

Considere-se novamente as listas de preços e de descontos.

Listagem

```
preço      1.23 7.08 3.00 0.53 2.01 4.44 9.99 2.50 0.20 7.70
desconto   8.0  5.5  3.0  3.3  9.0  9.9  2.0  2.0  5.0  5.0
```

Pretende-se analisar os preços mais pequenos, em particular os preços mais pequenos com descontos maiores.

Objetivo

Escrever um programa que faça *output* de todos preços menores que 2.00 e com descontos maiores ou iguais que 5.0%.

Solução

Usar os operadores de comparação menor, <, e maior ou igual, >=, e o operador lógico and.

Programa - no ficheiro `analise_valores_pequenos.py`

```
precos      = [1.23, 7.08, 3.00, 0.53, 2.01, 4.44, 9.99, 2.50, 0.20, 7.70]
descontos   = [8.0, 5.5, 3.0, 3.3, 9.0, 9.9, 2.0, 2.0, 5.0, 5.0]

print('preços menores que 2.00 e com descontos maiores ou iguais que 5.0%:')
for indice in range(10):

    preco      = precos[indice]
    desconto   = descontos[indice]

    if (preco < 2.0) and (desconto >= 5.0):
        print('preço')
        print(preco)
        print('desconto')
        print(desconto)
        print('-----')
```

Output

```
preços menores que 2.00 e com descontos maiores ou iguais que 5.0%:
preço
1.23
desconto
8.0
-----
preço
0.2
desconto
5.0
-----
```

Como funciona

- Sempre que o valor da variável `preco` é menor do que 2.0, a condição `preco < 2.0` é verdadeira.
- Por outro lado, sempre que o valor da variável `desconto` é maior ou igual que 5.0, a condição `desconto >= 5.0` é verdadeira.

Sobre funções

- `and` é o operador booleano de conjunção. Por isso sempre que ambas as condições `preco < 2.0` e `desconto >= 5.0` são verdadeiras, a conjunção também é verdadeira e o corpo do `if` é executado.

Sobre valores `bool`

Já por várias vezes referimos que uma condição pode ser verdadeira ou falsa. Por exemplo, quando falámos da execução condicional de código, vimos que a condição `quantidade == 1` é verdadeira quando o valor da variável `quantidade` for `1`. E no exemplo anterior, as condições `preco < 2.0` e `desconto >= 5.0` podem ser verdadeiras ou falsas, consoante os valores das variáveis `preco` e `desconto`. Do ponto de vista da execução de código Python isto quer dizer que a avaliação de uma condição resulta num valor que pode ser verdadeiro ou falso. `True` ou `False`. Estes valores são do tipo `bool`. `bool` é a abreviação da palavra `boolean` que se refere à álgebra de Boole, ou álgebra booleana, criada pelo matemático George Boole em 1847, e que usa apenas dois valores, verdadeiro e falso.

Podem inicializar-se variáveis diretamente com os valores booleanos `True` ou `False`. Relembre-se que o interpretador Python é case *sensitive*. O que quer dizer que as letras maiúsculas são diferentes das minúsculas. Por isso os valores `True` e `False` têm que ser escritos com a primeira letra maiúscula.

Exemplos - no ficheiro `exemplos_bool.py`

```
quantidade = 1
print(quantidade == 1)
quantidade = 2
print(quantidade == 1)
preco = 1.0
print(preco < 2.0)
preco = 2.0
print(preco < 2.0)
desconto = 4.0
print(desconto >= 5.0)
desconto = 5.0
print(desconto >= 5.0)

x = True
print(x)
y = False
print(y)
z = true # esta linha dá erro porque true com t minúsculo não existe
print(z) # esta linha não é executada
```

Output

```
True
False
True
False
False
True
True
False
Traceback (most recent call last):
  File "/home/jbs/develop/202209_hello_python_world/pt/hello_world/hello_operadores_logicos_e_de_comparacao/exen
    z = true # esta linha dá erro porque true com t minúsculo não existe
NameError: name 'true' is not defined
```

Sobre operadores de comparação

Os operadores de comparação são:

- menor: `<`
- maior: `>`
- menor ou igual: `<=`
- maior ou igual: `>=`

Os operadores de igualdade e de desigualdade são também operadores de comparação:

- igualdade: `==`
- desigualdade: `!=`

O resultado da avaliação de condições que envolvam estes operadores é um valor booleano, `True` ou `False`.

Exemplos - no ficheiros `exemplos_operadores_comparacao.py`

```
x = 1
y = 2
print(x < y)
print(x > y)
print(x = y)
print(x == y)
print(x != y)
```

Output

```
True
False
True
False
False
True
```

Sobre operadores lógicos

Os operadores lógicos são:

- negação: `not`
- conjunção: `and`
- disjunção: `or`

O resultado da avaliação de condições que envolvam estes operadores é um valor booleano, `True` ou `False`.

Exemplos - no ficheiros `exemplos_operadores_logicos.py`

```
a = True
b = False
c = a and b
d = a or b
e = not a
print(a)
print(b)
print(c)
print(d)
print(e)
```

Output

```
True
False
False
True
False
```

Hello `input`

Inaceitável

Há um problema de fundo em todos os programas que fizemos até agora. É que os dados que os programas usam, estão escritos nos próprios programas. Isto quer dizer que para alterar os dados tem que se alterar o programa. Isso é inaceitável. O programa não deveria depender dos dados que o utilizador fornece.

Imagine-se o que seria se quando quisessemos fazer contas com uma folha de cálculo, tivéssemos que alterar o programa da folha de cálculo. Isso seria totalmente inaceitável. O utilizador de uma folha de cálculo nem sabe em que linguagem está escrito o programa da folha de cálculo. Nem tem que saber.

Objetivo

Pretende-se escrever um programa que faça *output* de um preço inserido pelo utilizador. Esse preço não pode estar escrito no programa.

Solução

Usar a função `input`. A função `input` recebe como argumento uma *string*. Faz *output* dessa *string*. Lê todas as teclas pressionadas pelo utilizador, até à tecla *enter* (ou tecla *return* nalguns computadores). Retorna a *string* que contem todas as letras correspondentes às teclas pressionadas.

Programa - no ficheiro `input_preco.py`

```
preco = input('insira, por favor, o preço que pretende, seguido de return (ou enter): ')\n\nprint('o preço que inseriu é este:')\nprint(preco)
```

Output de uma execução

```
insira, por favor, o preço que pretende, seguido de return (ou enter): 2.34\no preço que inseriu é este:\n2.34
```

Output de outra execução

```
insira, por favor, o preço que pretende, seguido de return (ou enter): 9.99\no preço que inseriu é este:\n9.99
```

Observações

- Note-se que os valores `2.34` e `9.99` não estão escritos no programa. Os dados estão separados do programa. Não fazem parte do programa. E por isso para alterar os dados não é preciso alterar o programa. Basta executar o programa novamente e inserir novos dados.
- Quando há valores escritos nos programas, diz-se que esses valores estão *hard coded*. Porque estão fixos. Não podem ser alterados sem que o próprio programa seja alterado. Tipicamente só se considera aceitável usar valores *hard coded* de constantes. Por exemplo, `HORA = 60 # minutos`. Por convenção os nomes das variáveis que armazenam constantes escrevem-se só com letras maiúsculas.
- Note-se que a função `input` retorna uma *string*.

Sobre funções

Para fazermos cálculos com preços inseridos pelo utilizador teríamos em primeiro lugar que acrescentar ao programa a validação das *strings* retornadas pela função `input`. De forma a garantir que o utilizador teria inserido números decimais. Nada impede o utilizador de inserir a *string* `abc`. E `abc` não é um número decimal. Os cálculos efetuados com `abc` dariam erro.

Em segundo lugar teríamos de converter as *strings* retornadas pela função `input` em números decimais.

Optámos por não fazer tudo isto aqui, porque o programa ficaria demasiado extenso para esta secção.

Output da inserção de um número inválido

```
insira, por favor, o preço que pretende, seguido de return (ou enter): abc
o preço que inseriu é este:
abc
```

Hello `while`

Objetivo

Escrever um programa que permita ler tantos preços quantos os que o utilizador quiser. O número de preços não é fixo. Numa execução o utilizador pode querer processar três preços, e noutra cinco, ou dez. Depende das necessidades do utilizador.

Solução

Claramente teremos que usar uma instrução de execução em ciclo, para repetir o código de `input`. Mas como o número de execuções do ciclo não é conhecido, não podemos usar um ciclo `for`.

A solução é usar um ciclo `while`. O ciclo `while` executa um bloco de código, enquanto uma condição for verdadeira. A palavra inglesa *while* quer dizer "enquanto".

Programa - no ficheiro `input_precos.py`

```
resposta = 'SIM'

while resposta == 'SIM':
    preco = input('insira o preço que pretende, seguido de return (ou enter): ')

    print('o preço que inseriu é este:')
    print(preco)

    print('pretende inserir mais peços?')
    resposta = input('insira SIM ou NÃO, seguido de return (ou enter): ')
```

Output de uma execução

```
insira o preço que pretende, seguido de return (ou enter): 1.23
o preço que inseriu é este:
1.23
pretende inserir mais peços?
insira SIM ou NÃO, seguido de return (ou enter): SIM
insira o preço que pretende, seguido de return (ou enter): 9.99
o preço que inseriu é este:
9.99
pretende inserir mais peços?
insira SIM ou NÃO, seguido de return (ou enter): SIM
insira o preço que pretende, seguido de return (ou enter): 4.89
o preço que inseriu é este:
4.89
pretende inserir mais peços?
insira SIM ou NÃO, seguido de return (ou enter): NÃO
```

Output de outra execução

```
insira o preço que pretende, seguido de return (ou enter): 0.55
o preço que inseriu é este:
0.55
pretende inserir mais peços?
insira SIM ou NÃO, seguido de return (ou enter): NÃO
```

Como funciona

O ciclo `while` permite executar um bloco de código, repetidamente, **enquanto** uma condição for verdadeira. O bloco de código executado pelo ciclo `while` distingue-se do restante código do programa, porque está indentado com 4 espaços, em relação à palavra `while`. A execução do ciclo termina quando a condição for falsa.

No caso deste programa a condição é `resposta == 'SIM'`. Da primeira vez que esta condição é avaliada, ela é verdadeira. Porque a variável `resposta` foi inicializada com a string `'SIM'`. E portanto o bloco de código é executado. O bloco de código termina com a atribuição à variável `resposta`, do valor retornado pela função `input`. Portanto se o utilizador responder `SIM`, o valor `'SIM'` é atribuído à variável `resposta`. E quando a condição for novamente avaliada ela continua a ser verdadeira e por isso o bloco de código é executado mais uma vez. Quando o utilizador responder `NÃO`, a variável `resposta` passa a ser igual ao valor `'NÃO'`. A condição `resposta == 'SIM'` é avaliada novamente. Mas agora a variável `resposta` é substituída por `'NÃO'`, e portanto a condição `'NÃO' == 'SIM'` é falsa. Nesta caso o bloco de código não é executado. E a execução do ciclo `while` termina.

Sobre o ciclo `while`

A estrutura do ciclo `while` é:

```
while condicao:
    instrucao_1      # bloco
    instrucao_2      # de
    outras_instrucoes # instruções
    instrucao_n      #
```

- O dois ponto `:` depois da condição é obrigatório.
- Quando o ciclo `while` é executado, a condição `condicao` é avaliada. Se for verdadeira o bloco é executado. Se for falsa, o bloco não é executado e a execução do ciclo termina.
- Depois de executado o bloco a condição é reavaliada. E repete-se todo o processo. Se for verdadeira o bloco é executado novamente. Se for falsa o bloco não é executado e o ciclo termina. E assim, sucessivamente até a condição à ser falsa.
- As instruções do bloco executado pelo ciclo `for` distinguem-se das restantes instruções do programa porque estão indentadas com quatro espaços relativamente à palavra `while`.

Comparação entre os ciclos `for` e `while`

O ciclo `for` usa uma sequência de valores. O ciclo é executado para cada um dos valores da sequência. O que quer dizer que o número de vezes que o ciclo é executado é igual ao número de valores da sequência. Ou seja, o número de vezes que o ciclo é executado é conhecido à partida. Porque está definido pelo número de elementos da sequência

No caso do ciclo `while`, o número de vezes que o ciclo é executado depende de uma condição. Não tem que ser conhecido à partida.

É esta a principal diferença entre os dois ciclos. Quando se sabe quantas vezes se vai executar o ciclo, usa-se um ciclo `for`. Quando não se sabe, usa-se um ciclo `while`.

Hello ficheiros

Considere a lista de preços, dispostos numa coluna:

```
1.23
7.08
3.00
0.53
2.01
4.44
9.99
2.50
0.20
7.70
```

No IDLE, faça `CTRL n` para criar um novo ficheiro. Copie estes preços para o novo ficheiro. E usando `CTRL s`, grave o ficheiro com o nome `precos.txt`.

A lista de preços está agora no ficheiro `precos.txt`. Um preço em cada linha.

Objetivo

Pretende-se escrever um programa que crie o ficheiro `precos_altos.txt` com todos os preços do ficheiro `precos.txt` que não começam por zero (neste contexto, os preços que começam por zero, como por exemplo, `0.53`, não são considerados altos porque são inferiores à unidade).

Solução

Criar uma lista vazia. Abrir o ficheiro `precos.txt` em modo de leitura. Usando um ciclo `while`, ler cada uma das linhas do ficheiro `precos.txt`. Sempre que o primeiro elemento da *string* lida não seja zero, acrescentá-la à lista.

Abrir o ficheiro `precos_altos.txt` em modo de escrita. Usando um ciclo `for` escrever cada um dos elemntos da lista no ficheiro `precos_altos.txt`.

Programa - no ficheiro `ficheiros_de_precos.py`

```
lista_auxiliar = []

nome_do_ficheiro = 'precos.txt'
modo = 'r' # leitura. r de read
codificacao = 'utf8' # isto é necessário em Windows
ficheiro_precos = open(nome_do_ficheiro, modo, encoding=codificacao)

print('a ler o ficheiro de preços...')
for linha in ficheiro_precos:
    if linha[0] != '0':
        lista_auxiliar.append(linha)
print('a ler o ficheiro de preços... fim!')

ficheiro_precos.close()

nome_do_ficheiro = 'precos_altos.txt'
modo = 'w' # escrita. w de write
codificacao = 'utf8' # isto é necessário em Windows
ficheiro_precos_altos = open(nome_do_ficheiro, modo, encoding=codificacao)

print('a escrever o ficheiro de preços altos...')
for linha in lista_auxiliar:
    ficheiro_precos_altos.write(linha)
print('a escrever o ficheiro de preços altos... fim!')

ficheiro_precos_altos.close()
```

Output

```
a ler ficheiro de preços...
a ler ficheiro de preços... fim!
a escrever ficheiro de preços altos...
a escrever ficheiro de preços altos... fim!
```

Output principal

O *output* principal deste programa está no ficheiro `precos_altos.txt` . Poderá abri-lo com o IDLE para verificar que tem este conteúdo.

```
1.23
7.08
3.00
2.01
4.44
9.99
2.50
7.70
```

Tal como era pretendido os preços `0.53` e `0.20` não estão no ficheiro `precos_altos.txt` .

Como funciona

Sobre ficheiros

Os ficheiros são recursos do computador, geridos pelo sistema operativo do computador.

Os ficheiros abrem-se usando a função `open`, e fecham-se usando a função `close`. Quando já não são usados, os ficheiros devem ser sempre fechados. Senão ficam a ocupar recursos do computador desnecessariamente.

Hello tuple

Considere-se novamente as listagens de preços e de descontos:

```
precos    = [1.23, 7.08, 3.00, 0.53, 2.01, 4.44, 9.99, 2.50, 0.20, 7.70]
descontos = [8.0 , 5.5 , 3.0 , 3.3 , 9.0 , 9.9 , 2.0 , 2.0 , 5.0 , 5.0]
```

Objetivo

Pretende-se escrever um programa que construa duas novas listas. Uma com os valores a pagar pelos diferentes produtos. Outra com os valores dos descontos dos diferentes produtos. Pretende-se que o programa faça *output* das quatro listas. E nada mais. Assim:

```
preços:
[1.23, 7.08, 3.0, 0.53, 2.01, 4.44, 9.99, 2.5, 0.2, 7.7]
descontos:
[8.0, 5.5, 3.0, 3.3, 9.0, 9.9, 2.0, 2.0, 5.0, 5.0]
valores a pagar:
[1.13, 6.69, 2.91, 0.51, 1.83, 4.0, 9.79, 2.45, 0.19, 7.32]
valores dos descontos:
[0.1, 0.39, 0.09, 0.02, 0.18, 0.44, 0.2, 0.05, 0.01, 0.39]
```

Solução

Criar uma nova função `valor_do_desconto` . Usar a função `valor_a_pagar` para obter os valores a pagar pelos diferentes produtos, e usar a nova função `valor_do_desconto` para obter os valores dos descontos respetivos.

Programa - no ficheiro `listagem_valores_a_pagar_e_descontos.py`

```

def valor_a_pagar(preco, desconto):

    valor_desconto = preco * desconto/100
    valor_a_pagar = preco - valor_desconto
    resultado      = round(valor_a_pagar, 2)

    return resultado

def valor_do_desconto(preco, desconto):

    valor_desconto = preco * desconto/100
    # no caso desta função a próxima linha não é necessária
    #valor_a_pagar = preco - valor_desconto
    resultado      = round(valor_desconto, 2)

    return resultado

precos          = [1.23, 7.08, 3.00, 0.53, 2.01, 4.44, 9.99, 2.50, 0.20, 7.70]
descontos       = [8.0 , 5.5 , 3.0 , 3.3 , 9.0 , 9.9 , 2.0 , 2.0 , 5.0 , 5.0]
valores_a_pagar = [] # inicialmente esta lista está vazia
valores_dos_descontos = [] # inicialmente esta lista está vazia

for indice in range(len(precos)):

    preco          = precos[indice]
    desconto       = descontos[indice]
    valor          = valor_a_pagar(preco, desconto)
    valor_desconto = valor_do_desconto(preco, desconto)

    valores_a_pagar.append(valor)
    valores_dos_descontos.append(valor_desconto)

print('preços:')
print(precos)
print('descontos:')
print(descontos)
print('valores a pagar:')
print(valores_a_pagar)
print('valores dos descontos:')
print(valores_dos_descontos)

```

Output

```

preços:
[1.23, 7.08, 3.0, 0.53, 2.01, 4.44, 9.99, 2.5, 0.2, 7.7]
descontos:
[8.0, 5.5, 3.0, 3.3, 9.0, 9.9, 2.0, 2.0, 5.0, 5.0]
valores a pagar:
[1.13, 6.69, 2.91, 0.51, 1.83, 4.0, 9.79, 2.45, 0.19, 7.32]
valores dos descontos:
[0.1, 0.39, 0.09, 0.02, 0.18, 0.44, 0.2, 0.05, 0.01, 0.39]

```

Inaceitável

Esta solução é inaceitável porque as duas funções são praticamente iguais. Particamemente só muda o valor retornado. Num caso, é o valor a pagar pelo produto, e no outro, o valor do desconto. Se por algum motivo se decidir alterar a forma com são feitos os cálculos, por exemplo, do valor do desconto, esta solução implica fazer essa alteração em dois sítios.

Observação

O que se pretende é que uma só função retorne dois valores. E isso é possível? Estritamente falando, não é. A instrução `return` retorna apenas um único valor. Mas esse valor pode ser uma estrutura de dados. Um valor que contenha em si vários valores. Habitualmente os valores retornados não deverão poder ser alterados fora da

função. Porque são produto da execução da função. São da responsabilidade da função. Nada fora da função os deverá poder alterar. Por isso a estrutura de dados que normalmente se usa nestes casos é o tuplo.

Solução

Criar uma só função que faz todos os cálculos e retorna um tuplo com dois valores. Um, o valor a pagar. Outro, o valor do desconto respetivo.

Programa - no ficheiro `listagem_valores_a_pagar_e_descontos_v2.py`

```
def valor_a_pagar_e_valor_desconto(preco, desconto):

    valor_desconto = preco * desconto/100
    valor_a_pagar = preco - valor_desconto
    resultado_1 = round(valor_a_pagar, 2)
    resultado_2 = round(valor_desconto, 2)

    resultado = (resultado_1, resultado_2)

    return resultado

precos = [1.23, 7.08, 3.00, 0.53, 2.01, 4.44, 9.99, 2.50, 0.20, 7.70]
descontos = [8.0, 5.5, 3.0, 3.3, 9.0, 9.9, 2.0, 2.0, 5.0, 5.0]
valores_a_pagar = [] # inicialmente esta lista está vazia
valores_dos_descontos = [] # inicialmente esta lista está vazia

for indice in range(len(precos)):

    preco = precos[indice]
    desconto = descontos[indice]
    valores = valor_a_pagar_e_valor_desconto(preco, desconto)
    # valores é um tuplo. o primeiro elemento é o valor a pagar. o segundo é
    # o valor do desconto
    valor_a_pagar = valores[0]
    valor_desconto = valores[1]

    valores_a_pagar.append(valor_a_pagar)
    valores_dos_descontos.append(valor_desconto)

print('preços:')
print(precos)
print('descontos:')
print(descontos)
print('valores a pagar:')
print(valores_a_pagar)
print('valores dos descontos:')
print(valores_dos_descontos)
```

Output

Não incluímos aqui o *output* porque é exatamente igual ao da solução inicial.

Como funciona

Os tuplos são estruturas de dados semelhantes às listas. Apenas com uma diferença, depois de criados não podem ser alterados. À semelhança das listas, os seus elementos obtêm-se usando índices. Os índices iniciam-se em zero. Portanto a função `valor_a_pagar_e_valor_desconto` retorna um tuplo com dois valores. O primeiro é o valor a pagar e o segundo o valor do desconto. O retorno da função é armazenado na variável `valores`. O elemento no índice zero da variável `valores` é o valor a pagar. O elemento no índice um da variável `valores` é o valor do desconto.

Sobre tuplos

Os tuplos são valores do tipo `tuple`. A palavra inglesa *tuple* quer dizer "tuplo". Os valores do tipo `tuple` são estruturas de dados. Servem para armazenar sequências de valores.

São em tudo semelhantes às listas, mas ao contrário das listas, depois de serem criados não podem ser alterados. Depois de serem criados não se lhes pode acrescentar, alterar ou remover elementos.

São úteis em situações em que se pretende criar uma sequência de valores que não possa ser alterada. Uma sequência *read-only*. É muito frequente serem usados em funções porque permitem retornar vários valores.

Tal como nas listas, os valores armazenados num tuplo não têm que ser todos do mesmo tipo. Podem ser de tipos diferentes.

Tal como nas listas os valores dos tuplos são obtidos por indexação.

Criar um tuplo

Para criar um tuplo colocam-se os elementos do tuplo, separados por vírgulas, entre parentesis curvos.

Exemplo

```
resultado = (7.32, 0.39)
```

Indexar

Tal como nas listas, para se obter um valor armazenado num tuplo, usa-se o seu índice no tuplo, e indexa-se o tuplo com esse índice. Para indexar um tuplo coloca-se o nome do tuplo seguido de parentesis retos. Dentro dos parentesis retos coloca-se o índice do elemento pretendido. Os índices iniciam-se em zero. O índice do primeiro valor de um tuplo é zero. O do segundo é um. O do terceiro é dois, e assim por diante.

Exemplo

```
print(resultado[0])
```

Alterar elemento

Depois de criados os tuplos não podem ser alterados. O exemplo que se segue dá erro.

Exemplo

```
resultado[1] = 0.0 # esta linha dá erro
```

Erro

Tal como nas listas, se se tentar indexar um tuplo numa posição onde não existe nenhum elemento, o interpretador dá erro.

Exemplo

```
# o terceiro elemento não existe
print(resultado[2]) # esta linha dá erro
```

Exemplo completo - no ficheiro `exemplo_tuplo.py`

Sobre funções

```
# resultado é um tuplo
resultado = (7.32, 0.39)

print(resultado[0])      # este é o primeiro elemento
indice = 1
print(resultado[indice]) # este é o segundo elemento

print('aqui está a tuplo completo')
print(resultado)

# alterar o segundo elemento
resultado[1] = 0.0 # esta linha dá erro

# o terceiro elemento não existe
print(resultado[2]) # esta linha dá erro
```

Output

```
7.32
0.39
aqui está a tuplo completo
(7.32, 0.39)
Traceback (most recent call last):
  File "exemplo_tuplo.py", line 12, in
    resultado[1] = 0.0 # esta linha dá erro. tem que ser comentada
TypeError: 'tuple' object does not support item assignment
```

Observação

Saliente-se mais uma vez que o interpretador Python executa todo o programa até à linha que dá erro. Quando não é possível executar uma dada linha, o interpretador interrompe a execução do programa nessa linha e dá erro. É o caso aqui. Não é possível alterar um tuplo. Nem é possível obter o terceiro elemento de um tuplo que tem dois elementos.

Hello `set`

Voltemos à lista de descontos:

```
[8.0 , 5.5 , 3.0 , 3.3 , 9.0 , 9.9 , 2.0 , 2.0 , 5.0 , 5.0]
```

Há descontos repetidos nesta listas. Alguns produtos têm o mesmo desconto que outros. Os descontos `2.0` e `5.0` estão repetidos. Suponhamos que queremos obter uma lista de todos os descontos diferentes, isto é, queremos obter uma lista de descontos sem repetições.

Objetivo

Pretense-se escrever um programa que obtenha uma lista de descontos sem repetições.

Solução

Naturalmente que para uma lista tão pequena, a solução mais natural, é retirar os elementos repetidos à mão. Mas queremos uma solução que resolva o problema com listas grandes. Com milhares de elementos.

No conceito matemático de conjunto não há repetições de elementos. O conceito de conjunto é suportado em Python pela estrutura de dados `set`. A palavra inglesa *set*, quer dizer "conjunto".

A solução para este problema passa então por criar um conjunto a partir da lista original, retirando assim os elementos repetidos. Depois criar uma lista a partir do conjunto obtido. Isto porque se pretende que o resultado final continue a ser uma lista.

A criação de um conjunto a partir de uma lista faz-se com a função `set`.

A criação de uma lista a partir de um conjunto faz-se com a função `list`.

Tudo isto se consegue fazer numa única linha de código.

programa - no ficheiro `lista_sem_repeticoes.py`

```
descontos = [8.0 , 5.5 , 3.0 , 3.3 , 9.0 , 9.9 , 2.0 , 2.0 , 5.0 , 5.0]

print(descontos)
conjunto = set(descontos)
print(conjunto)
lista_sem_repeticoes = list(conjunto)
print(lista_sem_repeticoes)
# numa única linha
lista_sem_repeticoes = list(set(descontos))
print(lista_sem_repeticoes)
```

Output

```
[8.0, 5.5, 3.0, 3.3, 9.0, 9.9, 2.0, 2.0, 5.0, 5.0]
{2.0, 3.0, 3.3, 5.5, 5.0, 8.0, 9.0, 9.9}
[2.0, 3.0, 3.3, 5.5, 5.0, 8.0, 9.0, 9.9]
[2.0, 3.0, 3.3, 5.5, 5.0, 8.0, 9.0, 9.9]
```

Como funciona

A variável `descontos` é uma lista. A chamada `set(descontos)` retorna um conjunto com os elementos da lista `descontos`. Como os conjuntos não têm repetições, os descontos repetidos são descartados. O conjunto retornado é armazenado na variável `conjunto`. A chamada `list(conjunto)` retorna uma lista com elementos do

`conjunto` `conjunto` . Como o conjunto não tem repetições, a lista retornada também não tem repetições.

Sobre valores `set`

O tipo dos valores conjunto é `set`. A palavra inglesa *set* quer dizer "conjunto". Os conjuntos em Python são estruturas de dados. Servem para armazenar elementos. Os elementos de um conjunto podem ser de tipos diferentes.

À semelhança do conceito matemático de conjunto:

- Os conjunto em Python não têm elementos repetidos.
- A ordem dos elementos dos conjuntos Python não se define (não existe).
- Um conjunto escreve-se como uma sequência de elementos separados por vírgulas, entre chavetas.

Também à semelhança do conceito matemático de conjunto, os conjuntos Python suportam as operações de pertença, inclusão (conjunto contido noutro), união, interseção, diferença e diferença exclusiva.

Exemplos - no ficheiro `exemplos_conjuntos.py`

```
produtos_loja_1 = {'chocolate', 'água', 'café', 'chocolate'}
produtos_loja_2 = {'pão', 'água', 'café', 'pão'}

# conjunto de todos os produtos
uniao = produtos_loja_1 | produtos_loja_2
# produtos vendidos em ambas as lojas
intersecao = produtos_loja_1 & produtos_loja_2
# produtos exclusivos da loja 1
diferenca = produtos_loja_1 - produtos_loja_2
# produtos exclusivos na loja 1 ou na loja 2
diferenca_exclusiva = produtos_loja_1 ^ produtos_loja_2

print(produtos_loja_1)
print(produtos_loja_2)
print('-----')
print(uniao)
print(intersecao)
print(diferenca)
print(diferenca_exclusiva)
```

Output

```
{'água', 'café', 'chocolate'}
{'água', 'café', 'pão'}
-----
{'água', 'pão', 'café', 'chocolate'}
{'água', 'café'}
{'chocolate'}
{'chocolate', 'pão'}
```

Hello dicionários

A lista `nomes`, que se segue, têm nomes de produtos. A lista `preços` tem preços de produtos. As duas listas estão alinhadas, isto é, o preço de cada produto está na lista de preços no índice do nome do produto, na lista de nomes. Por exemplo, como `pão` está no índice zero, o preço do pão está em `precos[0]`, ou seja, o preço do pão é `1.23`.

Listas

```
nomes = ['pão', 'água', 'café', 'sal', 'leite', 'kiwi', 'maça', 'pera']
precos = [1.23, 7.08, 3.00, 0.53, 2.01, 4.44, 9.99, 2.50]
```

Objetivo

Escrever uma função que dado o nome de um produto, retorne o seu preço.

Solução - no ficheiro `nome_preco.py`

```
def get_preco(nome, lista_nomes, lista_precos):

    indice_produto = None

    for indice in range(len(lista_nomes)):
        if lista_nomes[indice] == nome:
            indice_produto = indice

    if indice_produto != None:
        return lista_precos[indice_produto]
    else:
        return None

nomes = ['pão', 'água', 'café', 'sal', 'leite', 'kiwi', 'maça', 'pera']
precos = [1.23, 7.08, 3.00, 0.53, 2.01, 4.44, 9.99, 2.50]

# testes
produto = 'pão'
preco = get_preco(produto, nomes, precos)
print(produto)
if preco != None:
    print(preco)
else:
    print('preço não existente')

produto = 'sal'
preco = get_preco(produto, nomes, precos)
print(produto)
if preco != None:
    print(preco)
else:
    print('preço não existente')

produto = 'pera'
preco = get_preco(produto, nomes, precos)
print(produto)
if preco != None:
    print(preco)
else:
    print('preço não existente')

produto = 'cacau'
preco = get_preco(produto, nomes, precos)
print(produto)
if preco != None:
    print(preco)
else:
    print('preço não existente')
```

Output

```
pão
1.23
sal
0.53
pera
2.5
cacau
preço não existente
```

Versão melhorada - no ficheiro `nome_preco_v2.py`

```
def get_preco(nome, lista_nomes, lista_precos):

    indice_produto = None

    for indice in range(len(lista_nomes)):
        if lista_nomes[indice] == nome:
            indice_produto = indice

    if indice_produto != None:
        return lista_precos[indice_produto]
    else:
        return None

nomes = ['pão', 'água', 'café', 'sal', 'leite', 'kiwi', 'maça', 'pera']
precos = [1.23, 7.08, 3.00, 0.53, 2.01, 4.44, 9.99, 2.50]

# testes

def get_mensagem(preco):
    if preco != None:
        return preco
    else:
        return 'preço não existente'

# produtos a listar
produtos = ['pão', 'sal', 'pera', 'cacau']
for produto in produtos:
    preco = get_preco(produto, nomes, precos)
    mensagem = get_mensagem(preco)
    print(produto)
    print(mensagem)
```

Output

```
pão
1.23
sal
0.53
pera
2.5
cacau
preço não existente
```

Alternativa

Aquilo que gostaríamos de fazer aqui, seria indexar uma estrutura de dados de preços, usando o nome de um produto.

Será que existe alguma estrutura de dados que permitir usar índices que não sejam números inteiros? Que sejam *strings*, por exemplo?

Sim, existe, são os dicionários.

Versão com dicionário - no ficheiro `nome_preco_v3.py`

```
def get_preco(nome, nomes_precos):

    if nome in nomes_precos:
        return nomes_precos[nome]
    else:
        return None

# chave: nome de produto
# valor: preço de produto
nomes_precos = {
    'pão': 1.23,
    'água': 7.08,
    'café': 3.00,
    'sal': 0.53,
    'leite': 2.01,
    'kiwi': 4.44,
    'maça': 9.99,
    'pera': 2.50
}

# testes

def get_mensagem(preco):
    if preco != None:
        return preco
    else:
        return 'preço não existente'

# produtos a listar
produtos = ['pão', 'sal', 'pera', 'cacau']
for produto in produtos:
    preco = get_preco(produto, nomes_precos)
    mensagem = get_mensagem(preco)
    print(produto)
    print(mensagem)
```

Output

```
pão
1.23
sal
0.53
pera
2.5
cacau
preço não existente
```

Como funciona

Os dicionários armazenam pares chave/valor (*key/value*). Indexando um dicionário com uma dada chave, obtém-se o valor correspondente a essa chave.