Programação

11.º ano

11

Ministério da Educação

Recursos digitais acessíveis por QR Code no manual. Versão digital em www.escolavirtual.cv

escola virtual
um serviço Porto Editora

Explora o manual digital do teu livro

Exercícios Interativos
Para resolução com *feedback* imediato.

Vídeos e interatividades
Explicam a matéria de forma motivadora.

Jogos
Exploram os conceitos curriculares de forma lúdica.

Áudios
Dão vida aos textos e ajudam a reforçar as competências linguísticas.

QuizEV
Desafiam-te a mostrares o que sabes. Podes, também, jogar com os teus amigos.

www.escolavirtual.cv

Programação

11.º ano

Explora o teu manual digital

https://escolavirtual.cv

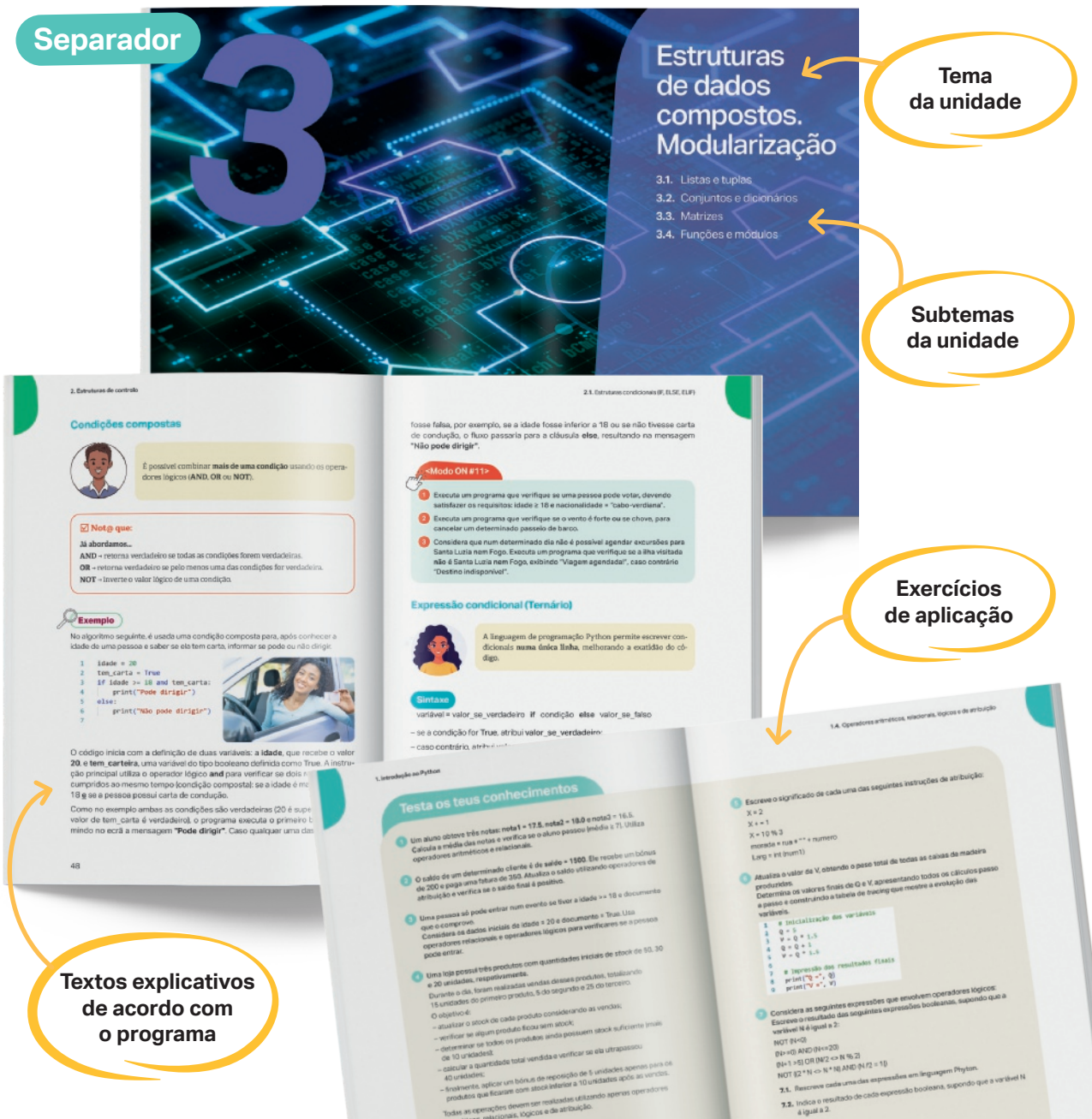
Acesso e condições de utilização em
www.escolavirtual.cv

Ministério da Educação

Conhece o teu caderno

Este manual foi desenvolvido para apoiar a tua iniciação à programação em Python®, está organizado em quatro capítulos, de acordo com o plano curricular do ensino secundário. O primeiro capítulo, dedicado à **Introdução ao Python**, apresenta os conceitos fundamentais da linguagem e estabelece as bases necessárias para começares a programar. No segundo capítulo, sobre **Estruturas de controlo**, são exploradas as sequências, seleções e repetições. O terceiro capítulo aborda as **Estruturas de dados compostos** e a **Modularização**, introduzindo listas, tuplas, dicionários e funções, o que te permitirá desenvolver programas mais organizados, claros e eficientes. Por fim, o capítulo do **Projeto integrador** reúne os conhecimentos adquiridos ao longo do manual e propõe a sua aplicação prática através de um projeto final.

Cada tema e subtema são compostos por…



Índice

Introdução ao Python

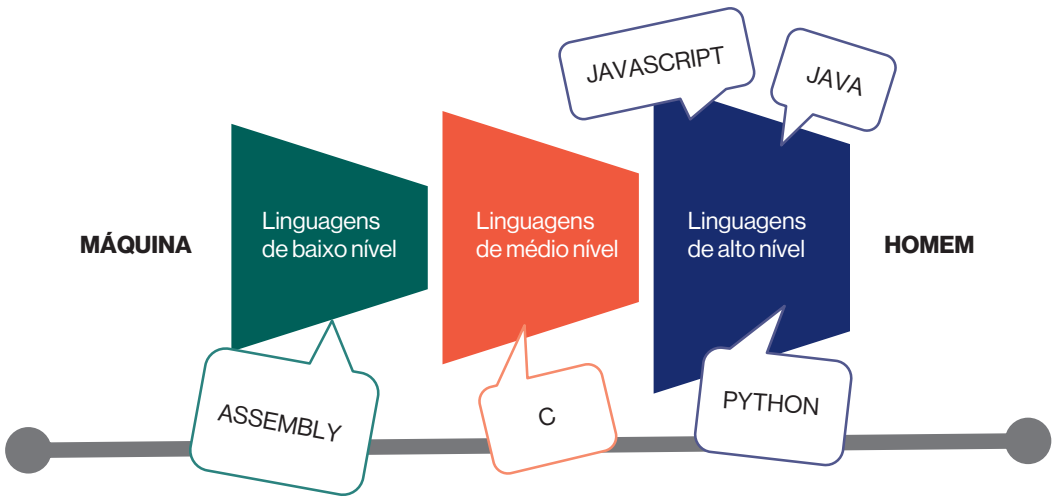
1 Introdução ao Python

No final deste capítulo, deverás ser capaz de:

- Conhecer o ambiente e linguagem Python.
- Entender a estrutura básica de um programa Python.
- Identificar e trabalhar com variáveis e vários tipos de dados e saber realizar conversões.
- Realizar operações matemáticas simples.

1.1. Ambiente de desenvolvimento e linguagem Python

Na informática, a classificação das linguagens de programação por "níveis" refere-se à distância que existe entre o código que o programador escreve e o código que o processador executa.



As **linguagens de baixo nível** estão quase totalmente ligadas à arquitetura física do computador. Embora permitam um controlo absoluto sobre o *hardware* e uma velocidade de execução máxima, são extremamente complexas para os seres humanos, uma vez que as suas instruções não se assemelham à nossa linguagem comum, mas sim a operações diretas do processador.

As **linguagens de médio nível** oferecem estruturas de programação mais legíveis e organizadas, mas mantêm a capacidade de manipular diretamente a memória do computador.

As **linguagens de alto nível** foram desenhadas para serem intuitivas e próximas da linguagem humana. São as linguagens mais populares atualmente, devido à sua facilidade de aprendizagem.

A tabela seguinte sistematiza as principais características de cada nível de linguagem, bem como a sua proximidade à máquina ou ao ser humano e alguns exemplos representativos. Esta organização permite compreender como as linguagens de programação evoluíram, tornando-se mais intuitivas e orientadas para a produtividade do programador.

Nível de linguagem	Proximidade	Características principais	Exemplos práticos
Baixo nível	Máquina	Difícil leitura humana; controlo total do *hardware*; execução instantânea.	Assembly Código Máquina
Médio nível	Híbrida	Equilibra a legibilidade com o acesso direto à memória; alta eficiência.	C C++
Alto nível	Humana	Fácil de ler e escrever; independente do *hardware*; foco na produtividade.	Python Java JavaScript

Atualmente existem muitas linguagens de programação, cada uma criada com objetivos diferentes, como desenvolver aplicações, criar jogos, trabalhar com dados ou controlar dispositivos. Entre essas linguagens, aparece a linguagem Phyton, amplamente usada em diversas áreas.

Manual Interativo

Vídeo
Python

O que é Python?

Python é uma linguagem de programação moderna, simples e poderosa, criada por Guido van Rossum, cuja designação ficou a dever-se ao gosto do autor pela série *Monty Python's Flying Circus*.
É uma **linguagem de programação de alto nível**, o que significa que foi projetada para ser próxima da linguagem humana, facilitando a leitura e a escrita do código.

Ao contrário de linguagens de baixo nível, como o Assembly, em que cada instrução corresponde diretamente a operações do processador, o Python permite que o programador se concentre na lógica do problema em vez de se preocupar com detalhes complexos de *hardware*.

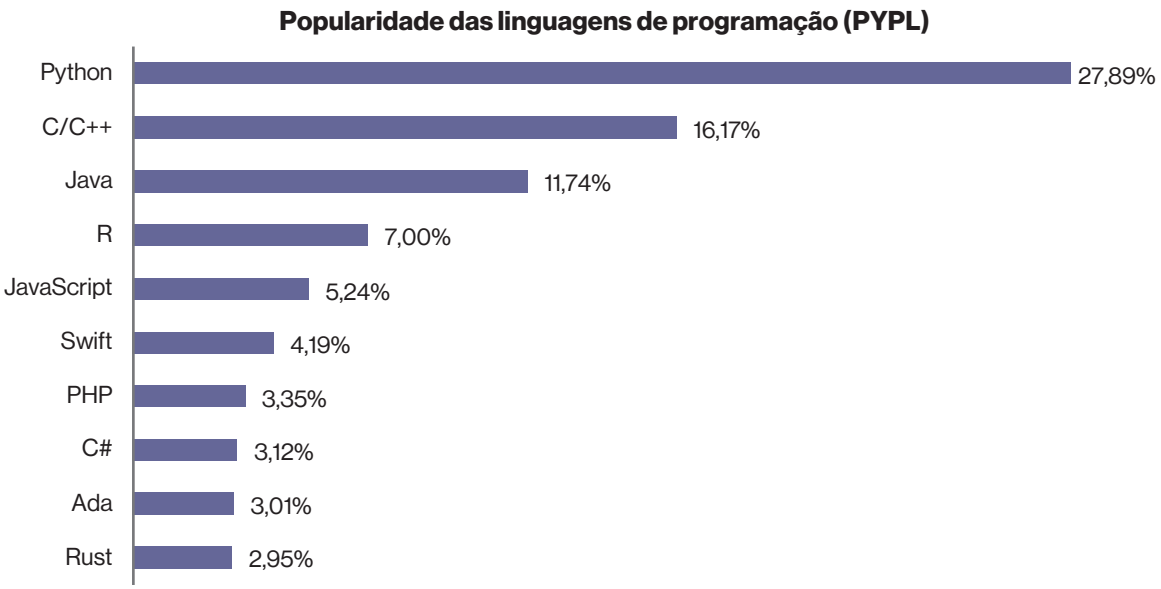
Isso torna a linguagem ideal tanto para iniciantes, que podem aprender conceitos de programação de forma intuitiva, quanto para profissionais, que consegue desenvolver projetos complexos de maneira mais rápida e eficiente.

Segundo o índice PYPL – *PopularitY of Programming Language*, de 2026, o Python detém 27,89% da quota de mercado como linguagem de programação mais procurada.

<Modo ON #1>

Durante quase 30 anos, o criador do Python teve o título oficial de "Ditador benevolente vitalício".
Quem é este programador holandês e o que este título estranho diz sobre a forma como as linguagens de programação evoluem com o tempo?

No gráfico seguinte, podes consultar que outras linguagens são atualmente usadas.



Disponível em https://pypl.github.io/PYPL.html [consult. jan 2026]

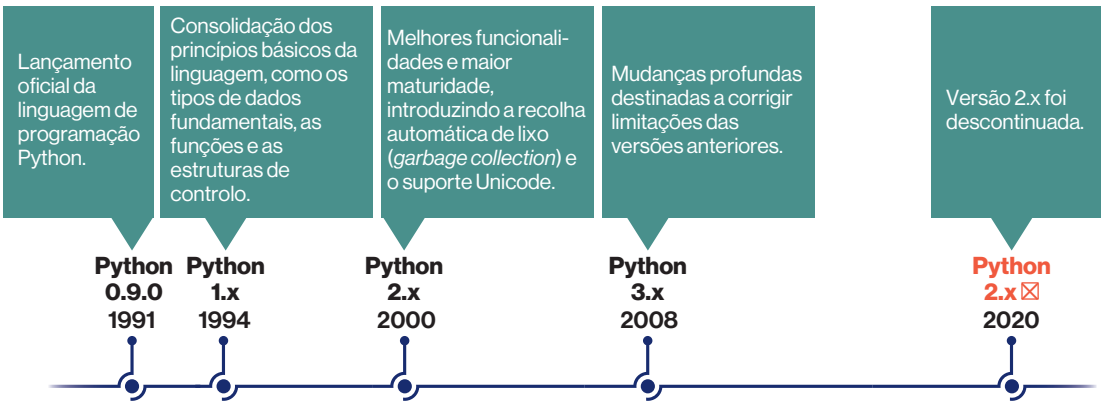
Manual Interativo

Link
Popularidade das linguagens de programação (PYPL)

Python é uma linguagem ***open source***, ou seja, o código-fonte da sua implementação é público e gratuito. Qualquer pessoa pode descarregar, utilizar, modificar e distribuir a linguagem sem custos.
Esta característica fomentou uma comunidade global muito ativa, que cria bibliotecas, ferramentas e documentação para facilitar o uso do Python em diferentes áreas, desde educação e ciência de dados até desenvolvimento *web*, automação e inteligência artificial.
O facto de ser *open source* também garante transparência e flexibilidade, permitindo que estudantes, professores e programadores contribuam para o seu aperfeiçoamento e adaptação a novas necessidades tecnológicas.

Desde o início, a linguagem foi concebida com um objetivo muito claro: ser fácil de ler e de escrever, aproximando-se da linguagem humana. Essa filosofia de *design* tornou o Python especialmente adequado ao ensino, enquanto permitiu que se tornasse uma ferramenta robusta e versátil para uso profissional.

Ao longo dos anos, o Python evoluiu significativamente.



Hoje, o Python continua a crescer e a adaptar-se às necessidades tecnológicas atuais. A sua sintaxe continua simples e intuitiva, mantendo o espírito original da linguagem, mas o seu ecossistema expandiu-se para incluir milhares de bibliotecas e ferramentas.

A sua adoção global deve-se ao facto de o Python ser compatível com:

Sabias que...

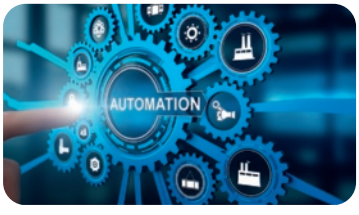
Além das versões principais, também existem **variações e implementações específicas da linguagem Python**, criadas para responder a necessidades particulares. Entre as mais conhecidas destacam-se:

- **CPython**, escrita em C. É a versão mais instalada nos computadores.
- **Jython**, versão da linguagem Python desenvolvida para correr sobre a Máquina Virtual Java (JVM), permitindo integrar código Python com aplicações Java.
- **IronPython**, implementação criada para funcionar na plataforma .NET da Microsoft®, facilitando a integração com aplicações e bibliotecas desse ecossistema.
- **MicroPython**, versão muito leve da linguagem Python, criada para dispositivos pequenos, como microcontroladores e placas eletrónicas (por exemplo, ESP32 ou micro:bit), muito utilizada em robótica educativa.
- **PyPy**, implementação focada em desempenho, com um compilador *just-in-time* (JIT), que torna a execução mais rápida em muitos casos.

O Python está presente em diversas áreas, como:

Inteligência artificial

Educação



Automação

Indústria

Desenvolvimento *web*

Ciência de dados

O Python é uma linguagem de programação ***case sensitive*** (distingue letras maiúsculas de minúsculas), orientada a objetos e reconhecida pelo seu **código autodescritivo**, simples e de fácil leitura.

Ambiente de desenvolvimento

Um **ambiente de desenvolvimento** é o conjunto de ferramentas que permite ao programador criar, testar e executar programas de forma organizada e eficiente. Funciona como um "espaço de trabalho" completo, reunindo tudo o que é necessário para transformar ideias em *software* funcional.

No caso da linguagem Python, este ambiente é composto por vários elementos que trabalham em conjunto para facilitar o processo de programação.

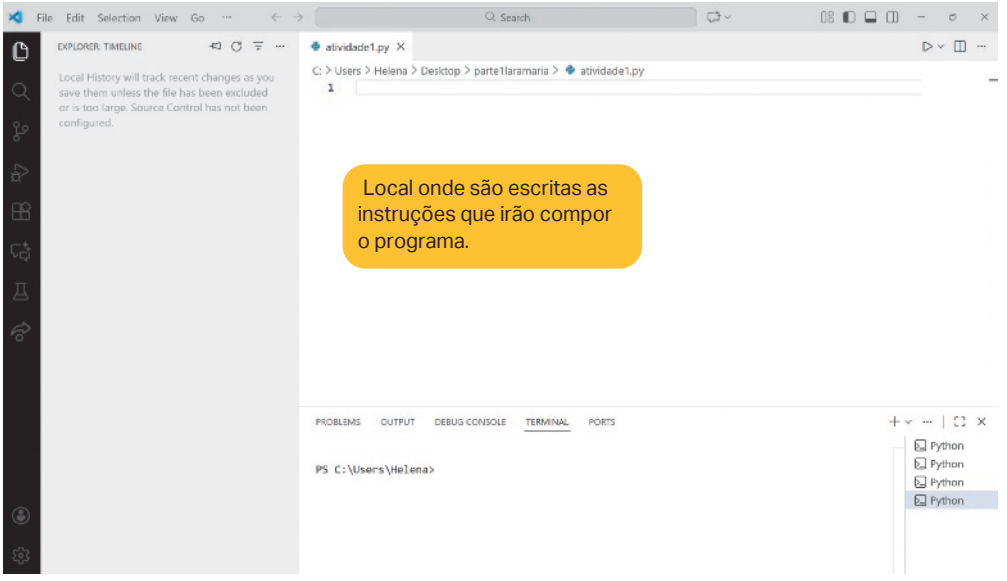
<Modo ON #2>

Se comparássemos a escrita de um programa de computador à preparação de uma receita culinária, quem seria o "cozinheiro", o que seriam os "ingredientes" e qual seria o papel da "linguagem de programação" nesse processo?

Um dos elementos centrais de um ambiente de desenvolvimento é o **editor de código Integrated Development and Learning Environment – IDLE**.

Os editores podem variar, desde aplicações simples e intuitivas, ideais para iniciantes, até ambientes mais avançados, que oferecem recursos como sugestão de código, destaque de erros e possibilidade de autocompletar o código.

IDLE Visual Studio Code®



Outro componente essencial é o **interpretador Python**, responsável por ler e executar o código linha a linha. Transforma as instruções escritas pelo programador em ações que o computador consegue compreender. Para interagir com o interpretador e executar os programas, utiliza-se frequentemente o **terminal ou consola**, uma interface baseada em texto que permite introduzir comandos, correr ficheiros Python e visualizar resultados imediatamente.

Quando executamos um programa Python,

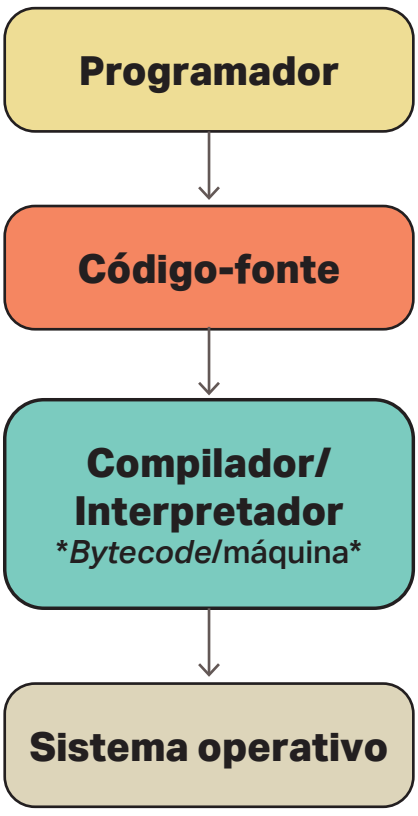
```
1   a = 10
2   b = 5
3   soma = a + b
4   print("Resultado:", soma)
```

O código-fonte (.py) é convertido em *bytecode*, uma forma intermediária mais simples e rápida de processar do que o código original.

Este *bytecode* não é diretamente compreendido pelo computador.

O interpretador assume um papel central na tradução do código.

O interpretador envia instruções correspondentes do *bytecode* para o CPU, que executa o programa no nível do código máquina, ou seja, nas instruções que o *hardware* entende.



Todo esse processo ocorre de forma automática e em tempo de execução, sem a necessidade de criar um ficheiro executável separado, como acontece em linguagens compiladas.

É exatamente este funcionamento que caracteriza o Python como uma linguagem interpretada, proporcionando flexibilidade, rapidez na escrita, teste e execução de programas.

Por fim, um ambiente de desenvolvimento completo inclui também **bibliotecas**, que são coleções de funcionalidades adicionais criadas para ampliar as capacidades da linguagem. Em vez de construir todas as funcionalidades do zero, o programador pode recorrer a essas bibliotecas para resolver problemas

específicos de forma rápida e eficiente, utilizando recursos para áreas como a Matemática, os gráficos, etc.

Manual Interativo

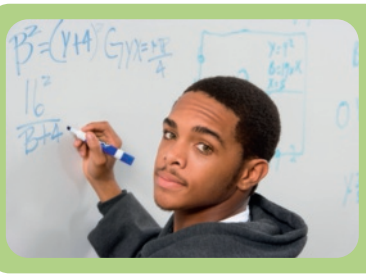
Link
Site oficial do Python®

Imagina que já criaste um programa que consegue somar uma lista grande de valores.

Se agora precisares de um novo programa para calcular a média, não precisas de escrever a lógica da soma novamente.

Na programação, trabalhamos com bibliotecas.

Chamas a função de Soma que já existe na tua "biblioteca" pessoal ou do sistema.



O novo programa apenas tem de realizar uma tarefa extra, "dividir esse resultado pela quantidade de valores".

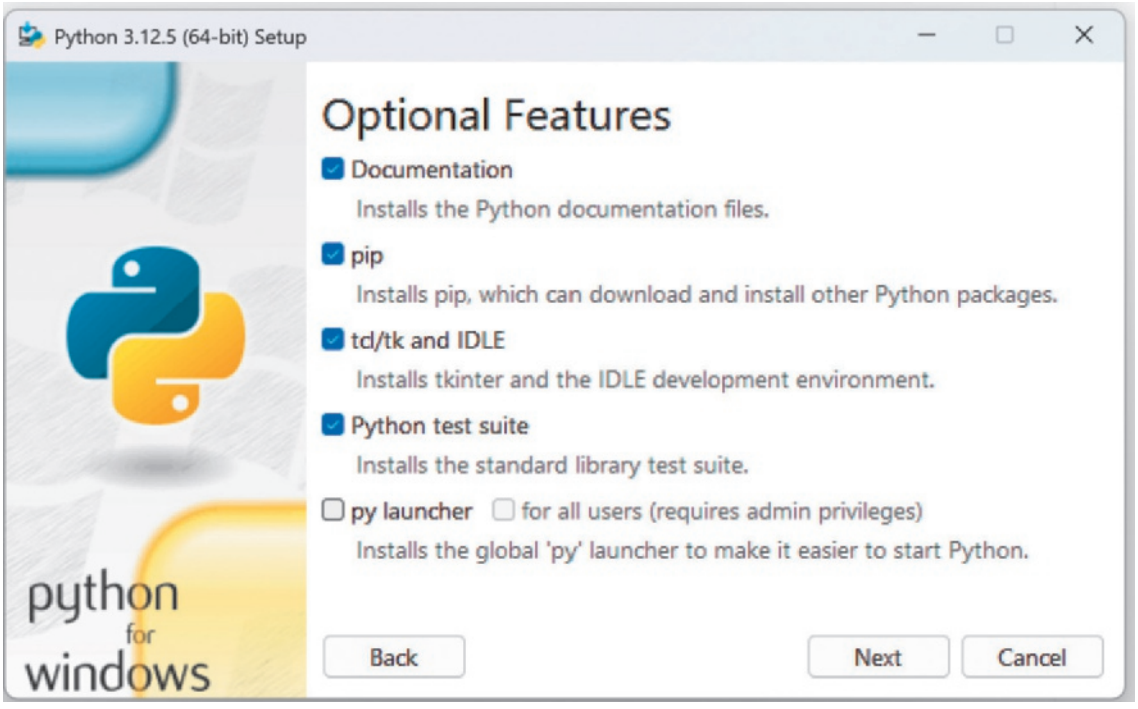
Porque é isto importante?

- Escreves menos código.
- Se o programa da "soma" já foi testado e funciona bem, evitas erros ao reutilizá-lo.
- Podes usar "peças" criadas por outros programadores para construir algo muito maior.

Instalar e configurar o Python

Para configurar um ambiente de desenvolvimento completo em Python, é recomendado instalar tanto o **Visual Studio Code (VS Code)** quanto o próprio **Python**.

O primeiro passo é instalar o **Python**, que deve ser descarregado diretamente do *site* oficial https://www.python.org/. Durante a instalação, é importante selecionar a opção **"Add Python to PATH"**, permitindo que o interpretador seja reconhecido pelo sistema em qualquer diretório.



Manual Interativo

Link
Site oficial do VSCode®

Em seguida, deve instalar-se o **Visual Studio Code**, disponível no *site* oficial https://code.visualstudio.com/. É aconselhável adicionar a extensão oficial Python do VS Code, que proporciona a integração completa com o interpretador Python, suporte a depuração, execução de código linha a linha e gestão de ambientes virtuais.

O **VS Code** é um editor de código (IDLE) leve, versátil e altamente configurável, que suporta diversas linguagens de programação e oferece recursos como destaque de sintaxe, autocompletar e integração com sistemas de controlo de versão.

<Modo ON #3>

Laboratório digital: instalação

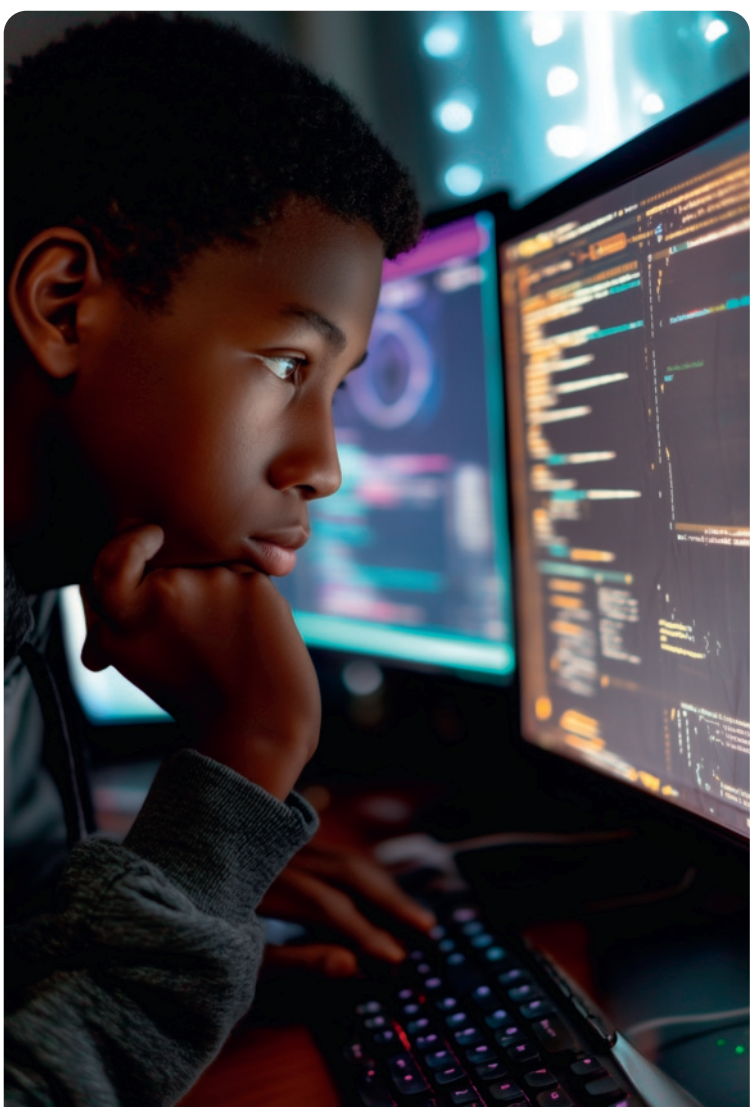
Faz a instalação do Python e do VS Code no computador da escola ou outro dispositivo. Instala também a extensão Python no VS Code. Testa a instalação com o código seguinte:

```
print("Vamos aprender a programar em Python")
```

Nota: Podes usar aspas duplas " ", ou aspas simples, ' '. As aspas tipográficas, “ ”, não são aceites em Python.

Nos casos em que não é possível instalar o *software* diretamente nos computadores, existem **alternativas *online*** que permitem programar em Python sem necessidade de instalação. Alguns exemplos são:

- **Replit®**: ambiente completo de programação acessível através do navegador;
- **Google Colab®**: ideal para projetos mais longos, que oferece recursos avançados e a possibilidade de armazenar e partilhar facilmente os trabalhos na nuvem;
- **Programiz®**: oferece um ambiente de programação simples e acessível, ideal para iniciantes aprenderem Python de forma prática.



Testa os teus conhecimentos

1 **Para cada questão, assinala a opção correta.**

1.1. Quem criou a linguagem Python e qual a origem do seu nome?

(A) Dennis Ritchie; inspirado numa espécie de serpente.

(B) Guido van Rossum; inspirado na série *Monty Python's Flying Circus*.

(C) James Gosling; inspirado num café da Indonésia.

(D) Bjarne Stroustrup; inspirado num herói de banda desenhada.

1.2. O que caracteriza uma linguagem ***open source***?

(A) O código é secreto e apenas empresas autorizadas podem usar.

(B) É uma linguagem paga, mas com suporte técnico gratuito.

(C) O código-fonte da sua implementação é público, gratuito e pode ser modificado por qualquer pessoa.

(D) É uma linguagem que só funciona em sistemas operativos de código aberto (Linux).

1.3. O que significa dizer que o Python é *case sensitive*?

(A) Que o programa corrige erros de ortografia automaticamente.

(B) Que a linguagem distingue entre letras maiúsculas e minúsculas (por exemplo, Soma é diferente de soma).

(C) Que a linguagem exige o uso de maiúsculas em todos os comandos.

(D) Que o código é sensível a espaços no final das linhas.

2 **Classifica em verdadeira (V) ou falsa (F) cada uma das afirmações seguintes.**

(A) O Python é considerado uma linguagem de baixo nível porque comunica diretamente com o *hardware*.

(B) O Python 2 foi oficialmente descontinuado no ano de 2020.

(C) O interpretador Python lê e executa o código linha a linha.

(D) O *bytecode* é uma forma intermediária de código que o CPU executa diretamente sem ajuda do interpretador.

(E) O Visual Studio Code (VS Code) é um exemplo de um editor de código (IDLE).

(F) As bibliotecas permitem reutilizar funcionalidades já criadas.

Testa os teus conhecimentos

3 **Completa as frases com as palavras corretas: interpretada, alto nível, bibliotecas, *bytecode*, CPython.**

a) O Python é uma linguagem de _____, pois a sua sintaxe é próxima da linguagem humana.

b) A implementação-padrão do Python, escrita em C e a mais utilizada em computadores, chama-se _____.

c) Antes de ser executado, o código-fonte (.py) é convertido em _____, uma representação mais simples para o interpretador.

d) Ao contrário de linguagens como o C++, o Python é uma linguagem _____, executando o código em tempo real.

e) As _____ são coleções de funcionalidades prontas a usar que evitam que o programador tenha de escrever tudo do zero.

4 **Faz corresponder a implementação do Python, na coluna A, ao seu objetivo principal, na coluna B.**

Coluna A	Coluna B
1. MicroPython	**(A)** Criado para funcionar na plataforma .NET da Microsoft.
2. Jython	**(B)** Focado em desempenho, usando um compilador just-in-time (JIT).
3. IronPython	**(C)** Versão leve para microcontroladores (robótica).
4. PyPy	**(D)** Desenvolvido para correr sobre a máquina virtual Java (JVM).

5 **Explica a diferença fundamental entre uma linguagem de alto nível (como a Python) e uma de baixo nível (como a Assembly).**

6 **Durante a instalação do Python no Windows®, porque é importante selecionar a opção Add Python to PATH?**

7 **Menciona duas opções *online* para programar em Python, caso não seja possível instalar um *software* no computador.**

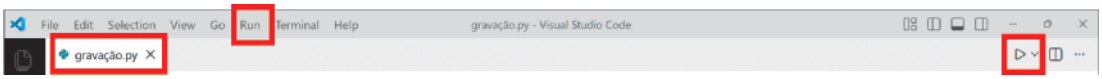
1.2. Estrutura básica de um programa

Manual Interativo

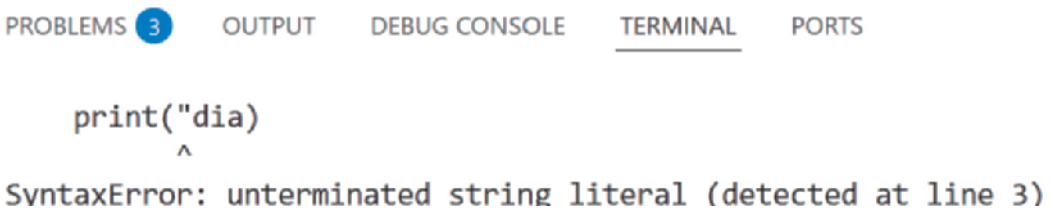
Vídeo
Introdução à linguagem de programação Python – O que é?; Desenvolvimento de programas em Python

Programas em Python

Um programa em Python é um conjunto de instruções numeradas, guardadas num ficheiro cuja **extensão** é **.py**. Para executar um programa em Python, clica no ícone **Play** ou **Menu Run** > **Start Debugging F5**.



O resultado da interpretação do código, com ou sem sinalização de erros, aparece, no caso do Visual Studio Code, na parte inferior do ecrã, no separador **TERMINAL**.



Comentários

Os comentários em linguagens de programação são excertos incluídos no código que **não são executados pelo computador**.

Em Python, um comentário é criado colocando-se o símbolo **#** antes do texto. Os comentários servem para **melhorar a clareza, a compreensão e a comunicação** do código, ajudando outros programadores a entenderem a lógica ou a intenção por detrás de determinadas partes do programa.

Exemplo

```
1   print("Bem-vindos ao estudo de Python!")  # exemplo de comentário não interpretado
```

Indentação

O Python utiliza a tecla 

para determinar blocos de código. A linha 3 do exemplo, por estar indentada, indica que pertence à instrução da linha 2.

Exemplo

```
1   idade=20
2   if idade>= 18:
3       print("És maior de idade")
```

Estrutura típica de um programa

A estrutura típica de um programa em Python segue uma organização simples e clara. Geralmente, começa pela **declaração ou inicialização de variáveis**, feita diretamente com a atribuição de valores, já que a linguagem não exige definir tipos.

Em seguida, ocorre o **processamento**, que inclui as operações, cálculos e a lógica implementada pelo programador para manipular esses dados. Por fim, o programa realiza a **saída de dados**, normalmente utilizando a função **print()**, que exibe os resultados para o utilizador no ecrã.

Exemplo

```
1   a = 10                      # inicialização da variável a
2   b = 5                       # inicialização da variável b
3   soma = a + b                # processamento
4   print("Resultado:", soma)   # saída de dados
```

☑ Not@ que:

No exemplo, as variáveis *a* e *b* assumem os valores inteiros 10 e 5, respetivamente. Para variáveis que assumem valores do tipo texto, devem ser adicionadas aspas duplas " " ou aspas simples ' '.

Formatação de strings

A formatação de **strings** (conjunto de *caracteres**) em Python utiliza o método **.format()**.

* Cáracter assume a grafia *caractere* no contexto de programação.

Esta técnica permite controlar com precisão como o texto é apresentado, sendo muito útil para criar tabelas ou alinhar dados na consola.

Sintaxe

{índice:preencher alinhar largura .precisão}.format (**string**)

Preencher	Alinhar	Largura	Precisão
Define o *caractere* usado para ocupar os espaços vazios (o padrão é o espaço).	Define a orientação do texto: < (esquerda) > (direita) ^ (centro)	Define a largura mínima total do campo.	Define a largura máxima ou o número de *caracteres*/casas decimais.

Exemplo

```
1   s = "Python"                    #ou 'Python'
2
3   print("{0:>15}".format(s))      # alinha à direita com 15 espaços e caracteres
4
5   print("{0:*>10}".format(s))     # alinha à direita com preenchimento de símbolos *, para 10 caracteres
6
7   print("{0:^16}".format(s))      # alinha ao centro usando 5 espaços à esquerda e 5 à direita
8
9   print("{0:.2}".format(s))       # imprime apenas as primeiras duas letras
```

OUTPUT

```
          Python
****Python
      Python
Py
```

<Modo ON #4>

Desenvolve um programa em Python que reproduza exatamente o *output* fornecido. Inclui os comentários explicativos para cada instrução do código.

```
#escrita da frase "Primeiro programa de Python"
#inicialização da varíavel nome em "Ana"
#escrita do conteúdo da variável nome
```

OUTPUT PRETENDIDO

```
Primeiro Programa de Python
Ana
```

Testa os teus conhecimentos

1 **Para cada questão, assinala a opção correta.**

1.1. O que significa dizer que o Python é uma linguagem *open source*?

(A) O Python só pode ser usado em escolas.

(B) O código da implementação da linguagem Python pode ser usado, estudado e modificado livremente.

(C) O Python é uma linguagem paga.

(D) O Python só funciona com ligação à Internet.

1.2. Para que serve o IDLE em Python?

(A) Executar apenas programas já compilados.

(B) Criar jogos avançados.

(C) Escrever e executar programas em Python.

(D) Converter código Python para outras linguagens.

1.3. O que é um ficheiro com extensão .py?

(A) Um ficheiro de texto comum.

(B) Um ficheiro de imagem.

(C) Um ficheiro que contém o código-fonte de um programa Python.

(D) Um ficheiro de instalação da linguagem Python.

2 **Comenta, instrução a instrução, o seguinte código de programação:**

```
1   nome = "Carla"
2
3   print("{0:<1}".format(nome))
4   print("{0:.3}".format(nome))
5   print("{0:^40}".format(nome))
6   print("{0:_>22}".format(nome))
```

1.3. Variáveis e tipos de dados

O que são variáveis?

Em Python, **definir variáveis** significa criar nomes que armazenam valores que podem mudar durante o programa. A linguagem é dinâmica, ou seja, não é necessário indicar o tipo da variável – ele é determinado automaticamente pelo Python de acordo com o valor atribuído.

Manual Interativo

Vídeo
Introdução à linguagem de programação Python – Aspectos básicos de Python variáveis – variáveis e tipos de dados

Exemplo

```
1   idade= 17
2   nome="Carla"
```

Regras na definição dos nomes das variáveis

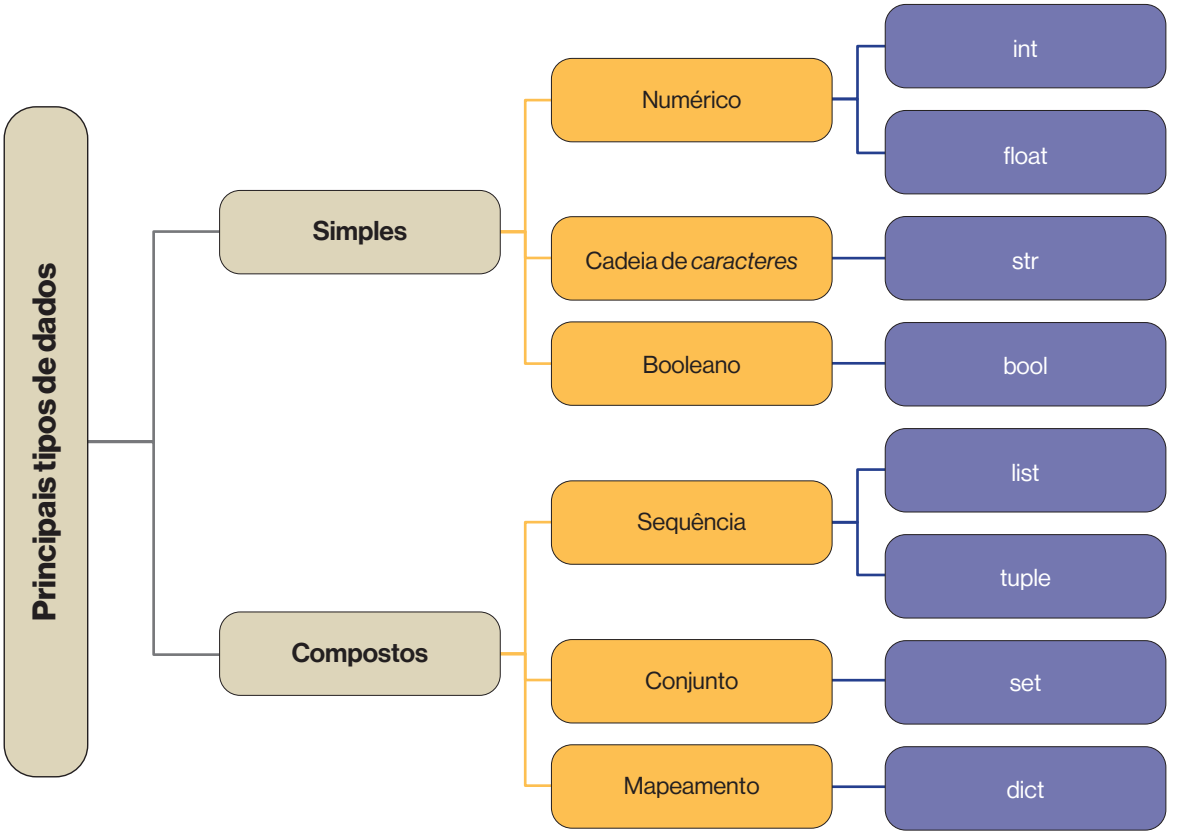
- Não podem começar com números.
- Não podem usar espaços nem símbolos (+, –, *, %, \, /...), com exceção do *underscore* (_).
- Devem ter nomes significativos.

Exemplos de nomes de variáveis válidos e inválidos em Python

Nome da variável	Válido?	Observação
nome	✓	Começa com letra, só letras
_idade	✓	Começa com *underscore*, permitido
nota_final1	✓	Letras, números e *underscore*
Idade	✓	Diferencia maiúsculas e minúsculas (idade ≠ Idade)
1nome	x	Não pode começar com número
nome-aluno	x	O hífen não é permitido
nome aluno	x	Os espaços não são permitidos
for	x	Palavra reservada da linguagem Python
True	x	Palavra reservada da linguagem Python
idade2	✓	Letras e números permitidos, desde que não comece com número
***nome**	x	Não pode conter símbolos, com exceção do *underscore*.

Tipos de dados em Python

Os **tipos de dados** são categorias que definem que tipo de valor uma variável pode armazenar e que operações podem ser feitas com esse valor.



Tipo de dado	Exemplo	Descrição
int	idade = 25	Números inteiros, sem parte decimal.
float	altura = 1.75	Números reais, com parte decimal.
str	nome = "Ana"	Cadeia de *caracteres* (texto).
bool	tem_carta = True	Valores lógicos, **True** (1) ou **False** (0).
list	frutas = ["maçã", "laranja"]	Lista ordenada de elementos, mutável.
tuple	coordenadas = (10, 20)	Tupla ordenada de elementos, imutável.
set	numeros = {1, 2, 3}	Conjunto de elementos únicos, sem ordem.
dict	aluno = {"nome": "Ana", "idade": 25}	Dicionário com pares chave:valor.

Conversão entre tipos

A **conversão de tipos** (ou *type casting*) é o processo que transforma um valor de um tipo de dado em outro tipo de dado. Em programação, isso é muito útil, porque diferentes operações exigem tipos específicos e nem sempre os dados estão no formato que precisamos.

Função de conversão	Exemplo	Resultado	Output
int()	int("10")	10	<class 'int'>
float()	float(5)	5.0	<class 'float'>
str()	str(25)	"25"	<class 'str'>
bool()	bool(0)	False	<class 'bool'>

☑ Not@ que:

type é uma função que retorna o tipo de dado de um valor ou variável.

Exemplo

```
1   # Conversão de string para inteiro
2   x = int("10")
3   print(x)          # Saída: 10
4   print(type(x))    # Saída: <class 'int'>
5
6   # Conversão de inteiro para float
7   y = float(5)
8   print(y)          # Saída: 5.0
9   print(type(y))    # Saída: <class 'float'>
10
11  # Conversão de inteiro para string
12  z = str(25)
13  print(z)          # Saída: "25"
14  print(type(z))    # Saída: <class 'str'>
15
16  # Conversão de inteiro para booleano
17  w = bool(0)
18  print(w)          # Saída: False
19  print(type(w))    # Saída: <class 'bool'>
20
21  # Conversão de float para inteiro
22  z=int(2.3)
23  print(z)          # Saída: 2
24  print(type(z))    # Saída: <class 'int'>
```

<Modo ON #5>

Cria um programa que declare as variáveis qtd = "20" (**string**), preco = "12.50" (**string**) e *stock* = 0 (inteiro).

O teu objetivo é converter a quantidade para inteiro, o preço para **float** e o *stock* para booleano.

Após as conversões, calcula o valor total da venda (quantidade * preço), converte esse resultado final para **string** e apresenta-o no ecrã juntamente com o tipo de cada variável convertida, utilizando a função **type()**.

Testa os teus conhecimentos

1 **Cria um programa em Python que armazene informações pessoais em variáveis, como nome, idade, altura, se a pessoa é solteira ou não e o nome da escola.**

O programa deve declarar cada informação numa variável adequada e, em seguida, exibir todas as informações no ecrã, indicando claramente o que cada valor representa.

2 **Cria 10 variáveis com atribuição e identifica, em comentário, o tipo automático atribuído pelo Python.**

3 **Para cinco variáveis, mostra como converter os seus valores entre tipos diferentes.**

```
# Criando 5 variáveis com tipos iniciais
nome = "Carlos"         # string
idade = "30"            # string
altura = 1.75           # float
solteiro = False        # booleano
nota = 18               # inteiro
```

4 **Qual é o tipo de dados usado para representar números inteiros em Python?**

(A) float **(B)** str
(C) int **(D)** bool

5 **Qual dos seguintes valores é do tipo float?**

(A) 10 **(B)** "3.5"
(C) 3.5 **(D)** True

6 **Qual é o tipo de dados de True e False em Python?**

(A) int **(B)** str
(C) bool **(D)** float

7 **Qual das opções representa corretamente um valor do tipo str?**

(A) 25 **(B)** 3.14
(C) False **(D)** "Python"

8 **Se executares o código x =int("15"), qual será o tipo da variável x?**

1.4. Operadores aritméticos, relacionais, lógicos e de atribuição

Operadores aritméticos

As **operações aritméticas** são utilizadas para manipular números, permitindo adicionar, subtrair, multiplicar, dividir, elevar à potência e até obter o resto de uma divisão.

Operação	Símbolo	Exemplo	Resultado	Descrição
Adição	**+**	10 + 5	15	Soma dois valores.
		"Rui" + "Pires"	Rui Pires	Concatena (junta) **strings**
Subtração	–	10 – 5	5	Subtrai o segundo valor do primeiro.
Multiplicação	*	10 * 5	50	Multiplica dois valores.
Divisão	/	10 / 5	2.0	Divide e retorna o número decimal (**float**).
Divisão inteira	//	10 // 3	3	Divide e retorna apenas a parte inteira.
Módulo (resto)	%	10 % 3	1	Retorna o resto da divisão.
Potenciação	**	2 ** 3	8	Eleva um valor à potência de outro.

Operadores relacionais

Os **operadores relacionais** são usados para comparar valores. Eles retornam sempre um resultado booleano **True (1)** ou **False (0)**.

Operação	Símbolo	Exemplo	Resultado	Descrição
Igual a	==	a == b	True/False	Verifica se dois valores são iguais.
Diferente de	!=	a != b	True/False	Verifica se os valores são diferentes.
Maior que	>	a > b	True/False	Verifica se o primeiro valor é maior que o segundo.
Menor que	<	a < b	True/False	Verifica se o primeiro valor é menor que o segundo.
Maior ou igual a	>=	a >= b	True/False	Verifica se o primeiro valor é maior ou igual ao segundo.
Menor ou igual a	<=	a <= b	True/False	Verifica se o primeiro valor é menor ou igual ao segundo.

Operadores lógicos

Os **operadores lógicos** são fundamentais para que um programa possa avaliar condições e escolher diferentes caminhos de execução.

Operação	Símbolo	Exemplo	Resultado	Descrição
E lógico	and	True and False	False	Retorna **True** somente se ambas as condições forem verdadeiras.
OU lógico	or	True or False	True	Retorna **True** se pelo menos uma das condições for verdadeira.
Negação lógica	not	not True	False	Inverte o valor lógico, **True** passa a **False** e vice-versa.

Operadores de atribuição

Os **operadores de atribuição** são utilizados para guardar valores em variáveis e também para os atualizar de forma prática.

Operação	Símbolo	Exemplo	Resultado	Descrição
Atribuição simples	=	x = 10	x passa a 10	Atribui um valor a uma variável.
Adição e atribuição	+=	x += 5	x = x + 5	Soma um valor e atualiza a variável.
Subtração e atribuição	–=	x –= 3	x = x – 3	Subtrai um valor e atualiza a variável.
Multiplicação e atribuição	*=	x *= 2	x = x * 2	Multiplica e atualiza a variável.
Divisão e atribuição	/=	x /= 4	x = x / 4	Divide e atualiza (resultado **float**).
Divisão inteira e atribuição	//=	x //= 4	x = x // 4	Realiza divisão inteira e atualiza.
Módulo e atribuição	%=	x %= 2	x = x % 2	Calcula o resto da divisão e atualiza.
Potenciação e atribuição	**=	x **= 3	x = x ** 3	Eleva à potência e atualiza.

Precedência de operadores

Em Python, a **atribuição (=)** é sempre o operador de menor prioridade, ou seja, todas as operações são avaliadas antes que o valor seja atribuído a uma variável.

A **potenciação (**)**, por sua vez, é avaliada da direita para a esquerda, o que significa que expressões como 2 ** 3 ** 2 são interpretadas como 2 ** (3 ** 2), resultando em 512.

Além disso, as **comparações** podem ser encadeadas de maneira intuitiva, permitindo expressões como 1 < x < 5, que verificam se *x* está entre 1 e 5.

Para ter controlo total sobre a ordem de execução das operações, é sempre possível utilizar **parênteses**, que têm a prioridade máxima e garantem que as expressões internas sejam avaliadas primeiro.

Nível	Operadores	Descrição
1 (mais forte)	()	Parênteses – forçam a ordem de avaliação.
2	**	Potenciação.
3	*, /, //, %	Multiplicação, divisão, divisão inteira e módulo.
4	+, –	Adição e subtração.
5	<, <=, >, >=, !=, ==	Operadores relacionais (comparação).
6	not	Negação lógica.
7	and	**E** lógico.
8	or	**OU** lógico.
9	if ... else ...	Expressão condicional.
10 (mais fraco)	=, +=, -=, *=, /=, etc.	Operadores de atribuição.

Testa os teus conhecimentos

1. **Um aluno obteve três notas: nota1 = 17.5, nota2 = 18.0 e nota3 = 16.5. Calcula a média das notas e verifica se o aluno passou (média ≥ 7). Utiliza operadores aritméticos e relacionais.**

2. **O saldo de um determinado cliente é de saldo = 1500. Ele recebe um bónus de 200 e paga uma fatura de 350. Atualiza o saldo utilizando operadores de atribuição e verifica se o saldo final é positivo.**

3. **Uma pessoa só pode entrar num evento se tiver a idade >= 18 e um documento que o comprove.**
   Considera os dados iniciais de idade = 20 e documento = True. Usa operadores relacionais e operadores lógicos para verificares se a pessoa pode entrar.

4. **Uma loja possui três produtos com quantidades iniciais de *stock* de 50, 30 e 20 unidades, respetivamente.**

   Durante o dia, foram realizadas vendas desses produtos, totalizando 15 unidades do primeiro produto, 5 do segundo e 25 do terceiro.

   O objetivo é:

   – atualizar o *stock* de cada produto considerando as vendas;

   – verificar se algum produto ficou sem *stock*;

   – determinar se todos os produtos ainda possuem *stock* suficiente (mais de 10 unidades);

   – calcular a quantidade total vendida e verificar se ela ultrapassou 40 unidades;

   – finalmente, aplicar um bónus de reposição de 5 unidades apenas para os produtos que ficaram com *stock* inferior a 10 unidades após as vendas.

   Todas as operações devem ser realizadas utilizando apenas operadores aritméticos, relacionais, lógicos e de atribuição.

   OUTPUT PRETENDIDO

```
Stocks finais: 35 25 0
Algum produto sem Stock? True
Todos com stock suficiente? False
Total vendido: 45
Vendas acima de 40 unidades? True
```

5 **Escreve o significado de cada uma das seguintes instruções de atribuição:**

X = 2

X + = 1

X = 10 % 3

morada = rua + " " + numero

Larg = int (num1)

6 **Atualiza o valor de V, obtendo o peso total de todas as caixas de madeira produzidas.**
Determina os valores finais de Q e V, apresentando todos os cálculos passo a passo e construindo a tabela de *tracing* que mostre a evolução das variáveis.

```
1   # Inicialização das variáveis
2   Q = 5
3   V = Q * 1.5
4   Q = Q + 1
5   V = Q * 1.5
6
7   # Impressão dos resultados finais
8   print("Q =", Q)
9   print("V =", V)
```

7 **Considera as seguintes expressões que envolvem operadores lógicos:**

NOT (N<0)

(N>=0) AND (N<=20)

(N+1 >5) OR (N/2 <> N % 2)

NOT ((2 * N <> N * N) AND (N /2 = 1))

7.1. Rescreve cada uma das expressões em linguagem Phyton.

7.2. Indica o resultado de cada expressão booleana, supondo que a variável N é igual a 2.

1.5. Entrada e saída de dados

Manual Interativo

Vídeo
Introdução à linguagem de programação Python – Função input; Função print

Função input()

A **função input()** permite receber dados do utilizador durante a execução de um programa.
Quando é chamada, o programa aguarda que o utilizador escreva algo.

Exemplo

Mostra a mensagem: Como te chamas? Aguarda a resposta. Guarda o que o utilizador escreveu na variável nome.

```
1    nome = input("Como te chamas? ")
```

OUTPUT

```
Como te chamas?                Como te chamas? Carla
```

Conversão após input

Tudo o que o utilizador escreve é lido como texto (**string**). Para usar números, é preciso converter o valor.

Exemplo

Conversão para inteiro:

```
1    idade = int(input("Idade: "))
```

Solicita um dado **idade** ao utilizador, que é recebido sempre como uma **string** (texto). Converte esse texto num número inteiro para permitir possíveis operações matemáticas.

Conversão para número decimal:

```
3    altura = float(input("Altura (em metros): "))
```

Solicita um dado **altura** ao utilizador, que é recebido sempre como uma **string**, e converte esse texto num número real.

Conversão explícita para texto:

```
1   x=bool(input("Tens carta de condução? Introduz 0 se não tens e 1 se tens "))
2   #converte 0 em False e 1 em True
3   print(x)
```

OUTPUT

```
Tens carta de condução? Introduz 0 se não tens e 1 se tens 1
True
```

Função print()

A **função print()** mostra informação no ecrã. Pode exibir texto, variáveis e resultados.

Exemplo

```
1   nome=input()
2   print("Olá", nome)
3   print() # separa automaticamente os elementos com uma linha em branco
4   print("Fim.")
```

OUTPUT

```
Carla
Olá Carla

Fim.
```

A função **input()** capta a informação digitada para a variável nome, em formato **string**. A função **print()** exibe "Olá" seguido do nome introduzido pelo utilizador. O uso de um **print()** vazio serve para inserir uma linha em branco. Escreve "Fim." na linha seguinte.

<Modo ON #6>

Escreve um programa em Python que peça ao utilizador para introduzir dois números, que deverão ser guardados nas variáveis a e b. Como o computador recebe estes valores como texto, deves convertê-los para números inteiros para conseguires somá-los.

Depois de calculares a soma, mostra o resultado no ecrã através de uma mensagem.

Formatação de saída

O Python facilita a criação de **strings** dinâmicas através das **f-strings**, uma forma moderna e prática de interpolar variáveis diretamente no texto.

Exemplo

O **f** indica que dentro da **string** podem existir variáveis entre **{ }**. É a forma mais usada e recomendada no Python atual.

```
1   nome=input()
2   idade=int(input())
3   print(f"O aluno {nome} tem {idade} anos.")
```

OUTPUT

```
João
17
O aluno João tem 17 anos.
```

<Modo ON #7>

Escreve um programa em Python para ajudar um condutor a calcular o custo do abastecimento. O programa deve pedir ao utilizador o tipo de combustível (gasolina ou gasóleo), o preço por litro (que deve ser convertido para **float**) e a quantidade de litros abastecidos (também em **float**).

Depois de calculares o custo total, utiliza uma **f-string** para exibir no ecrã uma mensagem que resuma a operação, mostrando o nome do combustível e o valor total a pagar.

Testa os teus conhecimentos

1 Cria um programa que pergunte ao utilizador: o seu nome; o destino da viagem; quantos dias vai durar a viagem; quanto dinheiro tem disponível para gastar. Depois, o programa deve calcular quanto dinheiro poderá gastar por dia e mostrar uma mensagem formatada (**f-strings**) com todas as informações.

2 Cria um programa que pergunte ao utilizador: o seu nome e a sua idade em anos. Depois, o programa deve calcular a idade aproximada em meses, semanas e dias e mostrar todas as informações de forma clara e formatada conforme o *output*.

OUTPUT PRETENDIDO

```
Qual é o teu nome? Carla
Quantos anos tens? 55
Carla, tens 55 anos.
Isto corresponde aproximadamente a 660 meses, 2860 semanas e 20075 dias.
```

3 Após usar **input()**, como podemos converter o valor introduzido para um número inteiro?

(A) input(int())

(B) int(input())

(C) input() + int

(D) str(input())

4 Qual das opções utiliza corretamente uma **f-string** para mostrar o valor da variável idade?

(A) print("Tenho idade anos")

(B) print("Tenho", idade, "anos")

(C) print(f"Tenho {idade} anos")

(D) print("Tenho {idade} anos")

5 Qual é a forma correta de converter um valor introduzido pelo utilizador para **float**?

(A) float(input())

(B) input(float())

(C) int(input())

(D) str(input())


2

Estruturas de controlo

2.1. Estruturas condicionais (IF, ELSE, ELIF)

2.2. Estruturas de repetição (FOR, WHILE)

2.3. Controlo de fluxos (BREAK, CONTINUE)

2 Estruturas de controlo

No final deste capítulo, deverás ser capaz de:

- Entender e utilizar estruturas condicionais.
- Entender e utilizar estruturas de repetição.
- Controlar a execução de ciclos.
- Criar programas interativos.

Manual Interativo

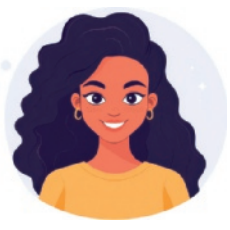
Vídeo
Introdução à linguagem de programação Python – if-elif-else

2.1. Estruturas condicionais (IF, ELSE, ELIF)

As estruturas condicionais são recursos presentes em linguagens de programação que permitem que um programa adote decisões com base em certas condições, possibilitando que o fluxo de execução siga caminhos diferentes dependendo de uma situação específica.

O funcionamento básico de uma estrutura condicional consiste em, se determinada condição for verdadeira, executar um bloco de comandos; caso contrário, realizar outro bloco ou simplesmente ignorar a ação.

Estrutura IF



O comando **IF** permite **executar um bloco de código** somente se uma condição for verdadeira.

Sintaxe

```
if condição:
    bloco_de_código
```

– condição: expressão que retorna **True** (verdadeira) ou **False** (falsa);

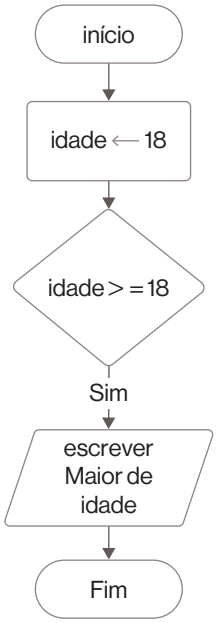
– bloco de código: conjunto de instruções que será executado caso a condição seja verdadeira;

– indentação (tabulação): em Python, é obrigatória para definir o início e o fim do bloco de código.

Exemplo

No algoritmo seguinte, é seguida uma estrutura condicional **IF** para verificar se uma determinada idade é superior ou igual a 18 anos, determinando se a pessoa é maior de idade.

Fluxograma



Linguagem Python

```
1   idade = 18
2   if idade >= 18:
3       print("Maior de idade")
```

O código começa por definir a variável **idade** com o valor **18**. Em seguida, utiliza a instrução **if** para avaliar se a condição **idade >= 18** é satisfeita. Como o valor da variável é exatamente 18, a expressão lógica resulta em verdadeiro, o que permite ao programa entrar no bloco de código indentado e executar o comando para imprimir a mensagem "**Maior de idade**" no ecrã.

Se a idade fosse inferior a 18, o programa simplesmente terminaria sem realizar qualquer ação ou exibir qualquer mensagem.

<Modo ON #8>

1. Executa um programa que verifique se a temperatura (introduzida pelo utilizador) é superior a 30 °C e, se tal acontecer, escreva "Dia muito quente" no ecrã.

OUTPUT

```
Qual a temperatura atual? 44
Dia muito quente
```

2. Executa um programa que verifique se a idade de uma pessoa é maior do que 18 anos e escreva no ecrã "Pode votar nas próximas eleições".

3. Executa um programa que verifique se um número de passageiros num *ferry* é igual à lotação máxima de 100 pessoas e escreva "Lotação máxima atingida".

4. Executa um programa que verifique se uma ilha corresponde a "Sal" e escreva "Destino turístico popular" no ecrã.

Estrutura ELSE

O **ELSE** define um bloco alternativo que será executado quando a condição **IF** for falsa.

Sintaxe

if condição:
  bloco_se_verdadeiro
else:
  bloco_se_falso

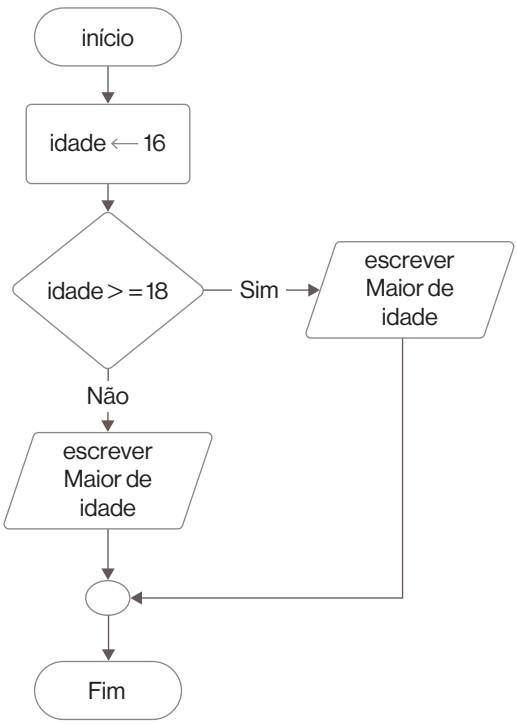
– se a condição for **True**, executa o bloco_se_verdadeiro;

– caso contrário, executa o bloco_se_falso.

Exemplo

No algoritmo seguinte, é seguida uma estrutura condicional **IF ELSE** para verificar, em função da idade, se uma pessoa é maior de idade ou menor de idade.

Fluxograma



Linguagem Python

```
1   idade = 16
2   if idade >= 18:
3       print("Maior de idade")
4   else:
5       print("Menor de idade")
```

Este exemplo apresenta a estrutura condicional **IF ELSE** completa, utilizada para decidir entre dois caminhos alternativos.

O código começa por definir a variável **idade** com o valor **16**. Em seguida, a instrução **if** avalia se a condição **idade >= 18** é verdadeira. Como o valor 16 não cumpre este requisito, o programa ignora o primeiro bloco de código e salta diretamente para a cláusula **else**. Esta cláusula funciona como uma alternativa obrigatória quando a condição inicial falha, resultando na execução da instrução que imprime a mensagem "**Menor de idade**" no ecrã. Ao contrário do **IF** simples, esta estrutura garante que o programa forneça sempre uma resposta ao utilizador, independentemente de a condição ser ou não satisfeita.

<Modo ON #9>

1. Executa um programa que verifique se o vento é forte (> 20 km/h); caso contrário, mostre a mensagem "Vento moderado" no ecrã.
2. Executa um programa que verifique se uma lancha está cheia; caso contrário, escreva "Ainda há lugares" no ecrã.
3. Executa um programa que verifique se a praia está aberta; caso contrário, escreva "Fechada por maré alta" no ecrã.

Estrutura ELIF

O **ELIF** (abreviatura de "ELSE IF") permite testar **várias condições em sequência** de forma organizada.

Sintaxe

```
if condição1:
    bloco1
elif condição2:
    bloco2
elif condição3:
    bloco3
else:
    bloco_final
```

– avalia a condição1, se for **True**, executa bloco1 e ignora todo o resto;

– se a condição1 for **False**, passa para condição2, se for **True**, executa bloco2;

– se nenhuma das anteriores for verdadeira, testa condição3, se for **True**, executa bloco3;

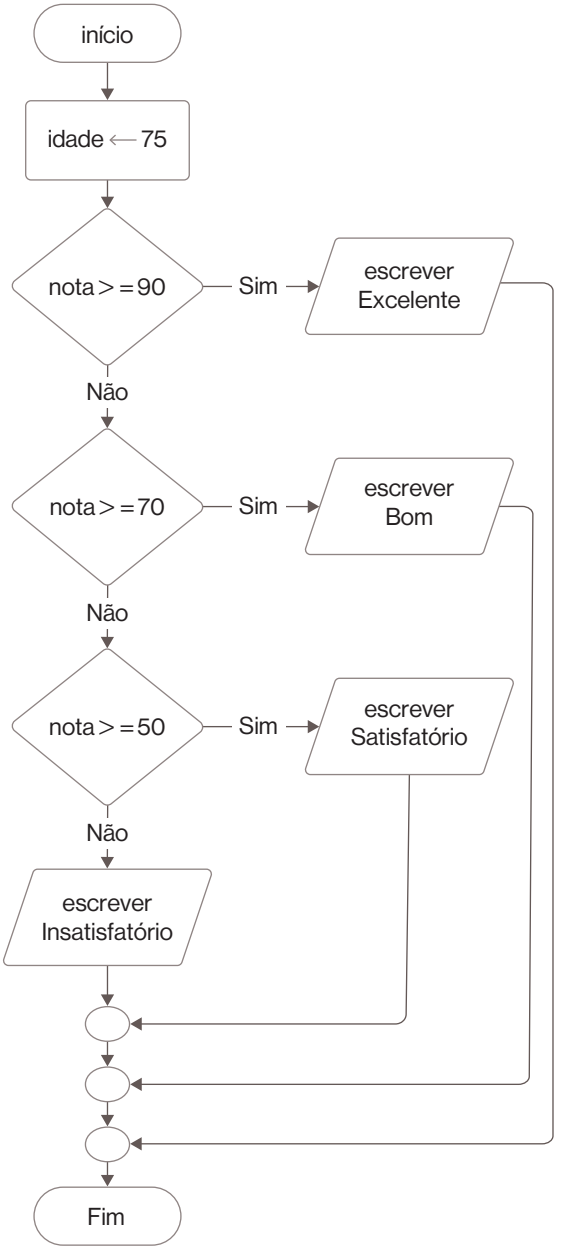
– se todas as condições forem **False**, executa o **bloco_final** do **else**;

– **importante**: apenas um bloco é executado — o primeiro cuja condição é **True**.

Exemplo

No algoritmo seguinte, é seguida uma estrutura condicional encadeada **ELIF** para atribuir uma classificação qualitativa (Excelente, Bom, Satisfatório, Insatisfatório) em função da nota de um teste numa escala de 0 a 100.

Fluxograma



Linguagem Python

```
1  nota = 75
2  if nota >= 90:
3      print("Excelente")
4  elif nota >= 70:
5      print("Bom")
6  elif nota >= 50:
7      print("Satisfatório")
8  else:
9      print("Insatisfatório")
```

Este exemplo evidencia a utilização da estrutura condicional encadeada recorrendo à cláusula **ELIF** (abreviatura de ELSE IF) para avaliar múltiplas condições em cascata.

O programa começa por atribuir o valor **75** à variável **nota**. A execução segue então uma ordem hierárquica de testes: primeiro, o **if** verifica se a nota é igual ou superior a 90; como esta condição é falsa, o código passa para o primeiro **elif**, que testa se a nota é igual ou superior a 70. Uma vez que 75 cumpre este requisito, o programa executa o bloco correspondente, imprimindo a mensagem **"Bom"** no ecrã.

É importante notar que, assim que uma condição é satisfeita, o Python ignora todas as verificações seguintes (como o **elif nota >= 50** ou o **else**), garantindo que seja exibida apenas uma resposta. Caso nenhuma das condições numéricas fosse satisfeita, o bloco **else** final seria ativado, exibindo **"Insatisfatório"** como último recurso.

<Modo ON #10>

1. **Realiza um programa que verifique se um número introduzido de turistas num determinado espaço público é baixo (< 50), médio (50-100) ou alto (> 100).**

2. **Realiza um programa que verifique qual o dia da semana, se for 1 escreve "Segunda", se for 2 escreve "Terça", sucessivamente até 7 ser "Domingo"; se for outro número, indica "Dia inválido".**

3. **Numa determinada disciplina um aluno tem três momentos de avaliação: nota1, nota2 e nota3 (escala de 0 a 20), sendo a média destes três momentos relevante para a aprovação da disciplina.**
   Além disso, ele precisa ter frequência mínima de 75% das atividades letivas para ser aprovado.
   A aprovação é definida da seguinte forma:
   – o aluno passa se média ≥ 12 e frequência ≥ 75%;
   – o aluno fica em recuperação se 9 ≤ média < 12 e frequência ≥ 75%;
   – caso contrário, o aluno reprova.

   Calcula a média das notas e determina a situação do aluno ("Aprovado", "Em recuperação" ou "Reprovado") usando operadores relacionais e lógicos. Atualiza a média com bónus de 1 ponto se a frequência for 100% usando operadores de atribuição.

Condições compostas

É possível combinar **mais de uma condição** usando os operadores lógicos (**AND**, **OR** ou **NOT**).

☑ Not@ que:

Já abordamos...

AND → retorna verdadeiro se todas as condições forem verdadeiras.

OR → retorna verdadeiro se pelo menos uma das condições for verdadeira.

NOT → inverte o valor lógico de uma condição.

Exemplo

No algoritmo seguinte, é usada uma condição composta para, após conhecer a idade de uma pessoa e saber se ela tem carta, informar se pode ou não dirigir.

```
1   idade = 20
2   tem_carta = True
3   if idade >= 18 and tem_carta:
4       print("Pode dirigir")
5   else:
6       print("Não pode dirigir")
7
```

O código inicia com a definição de duas variáveis: a **idade**, que recebe o valor **20**, e **tem_carteira**, uma variável do tipo booleano definida como **True**. A instrução principal utiliza o operador lógico **and** para verificar se dois requisitos são cumpridos ao mesmo tempo (condição composta): se a idade é maior ou igual a 18 **e** se a pessoa possui carta de condução.

Como no exemplo ambas as condições são verdadeiras (20 é superior a 18 e o valor de tem_carta é verdadeiro), o programa executa o primeiro bloco, imprimindo no ecrã a mensagem **"Pode dirigir"**. Caso qualquer uma das condições

fosse falsa, por exemplo, se a idade fosse inferior a 18 ou se não tivesse carta de condução, o fluxo passaria para a cláusula **else**, resultando na mensagem **"Não pode dirigir"**.

<Modo ON #11>

1. **Executa um programa que verifique se uma pessoa pode votar, devendo satisfazer os requisitos: idade ≥ 18 e nacionalidade = "cabo-verdiana".**
2. **Executa um programa que verifique se o vento é forte ou se chove, para cancelar um determinado passeio de barco.**
3. **Considera que num determinado dia não é possível agendar excursões para Santa Luzia nem Fogo. Executa um programa que verifique se a ilha visitada não é Santa Luzia nem Fogo, exibindo "Viagem agendada!", caso contrário "Destino indisponível".**

Expressão condicional (Ternário)

A linguagem de programação Python permite escrever condicionais **numa única linha**, melhorando a exatidão do código.

Sintaxe

variável = valor_se_verdadeiro **if** condição **else** valor_se_falso

– se a condição for **True**, atribui **valor_se_verdadeiro**;

– caso contrário, atribui **valor_se_falso**.

Exemplo

Exemplo de uma estrutura de decisão que verifique se uma pessoa é ou não maior de idade em função da sua idade.

```
1   idade = 16
2   mensagem = "Maior de idade" if idade >= 18 else "Menor de idade"
3   print(mensagem)
```

Inicialmente, o programa define uma variável chamada **idade** com o valor **16**. A variável **mensagem** é então criada para armazenar o resultado de uma avaliação lógica imediata. Verifica-se se a condição central **idade >= 18** é verdadeira e, se fosse, seria atribuído à variável o texto posicionado à esquerda **"Maior de idade"**.

Contudo, como **a condição é falsa** (visto que 16 é menor que 18), o programa executa a instrução após a cláusula **else**, atribuindo à mensagem o valor **"Menor de idade"**. Por fim, a instrução **print** exibe esse resultado no ecrã.

<Modo ON #12>

1. Executa um programa que determine "Vento forte" ou "Vento fraco" com base na velocidade do vento > 20 km/h.
2. Executa um programa que defina "Dia de praia" ou "Dia em casa" com base no tempo (sol ou chuva).
3. Executa um programa que indique "Feriado" ou "Dia normal" com base numa variável booleana.

Estrutura MATCH/CASE

MATCH/CASE refere-se a uma estrutura de seleção múltipla, usada quando queremos executar **diferentes blocos de código** consoante o valor de uma expressão.

Quando usar CASE em vez de IF/ELIF?

Usa-se **CASE** quando:

- há muitas opções possíveis baseadas no mesmo valor;
- o código ficaria muito longo com **ELIF**;
- se quer maior clareza e organização;
- se trabalha com padrões (estruturas, tuplas, tipos).

Observa:

com IF/ELIF	com CASE

if valor == 1:
  ...
elif valor == 2:
  ...
elif valor == 3:
  ...
else:
  ...
	match valor:
	  case 1:
	  ...
	  case 2:
	  ...
	  case 3:
	  ...
	  case _:
	  ...

> ☑ **Not@ que:**
>
> O símbolo "_" em "**case** _" funciona como caso-padrão, equivalente ao else na estrutura **IF/ELSE**.

Sintaxe

```
match expressão:
  case padrão1:
    bloco1
  case padrão2:
    bloco2
  case padrão3:
    bloco3
  case _:
    bloco_final
```

– **match:** avalia uma única expressão (apenas uma vez);

– **case:** define um padrão possível;

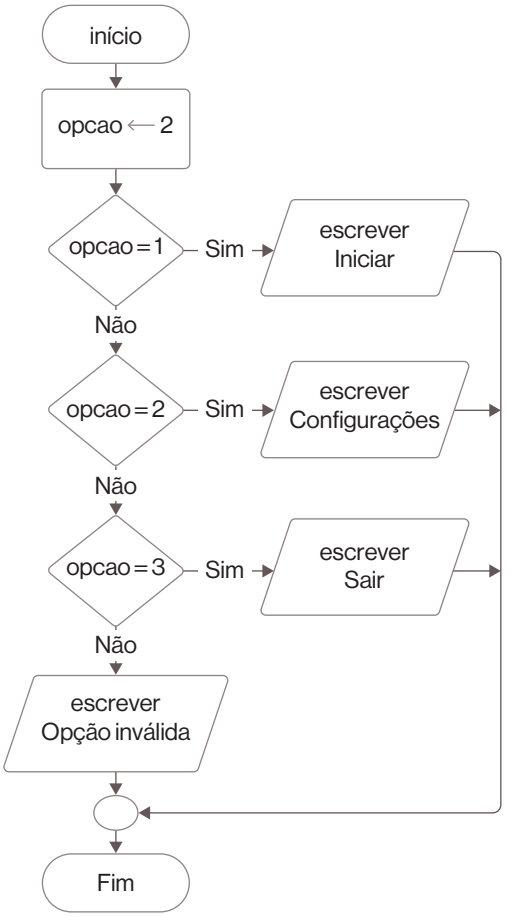
– **padrão:** valor ou estrutura a comparar, comparando esse valor com vários casos possíveis para executar o bloco correspondente;

– **_:** corresponde a qualquer valor (equivalente ao else); se nenhum caso coincidir, executa um caso por defeito.

Exemplo

Neste exemplo, é usada a estrutura **MATCH/CASE** para avaliar o conteúdo de uma variável "opção" e procurar um padrão correspondente entre os casos definidos.

Fluxograma



Linguagem Python

```
1   opcao = 2
2
3   match opcao:
4       case 1:
5           print("Iniciar")
6       case 2:
7           print("Configurações")
8       case 3:
9           print("Sair")
10      case _:
11          print("Opção inválida")
```

O programa inicia com a definição da variável **opcao**, à qual é atribuído o valor **2**. Através da instrução **match**, o Python avalia o conteúdo desta variável e procura um padrão correspondente entre os casos a seguir definidos.

No primeiro cenário, o **case 1**, o programa imprimiria **"Iniciar"**, mas como o valor é 2, ele avança para o **case 2**, onde encontra a correspondência exata e executa a instrução para imprimir **"Configurações"** no ecrã. Caso o valor fosse **3**, seria ativado o **case 3** para exibir **"Sair"**.

Por fim, o código inclui um **case** _, que considera qualquer valor que não tenha sido previsto nos casos anteriores, resultando na mensagem **"Opção inválida"**, garantindo que o programa tem sempre uma resposta mesmo perante entradas inesperadas.

Iniciar	Configurações	Sair	Opção inválida

<Modo ON #13>

1 **Cria um programa que peça ao utilizador um número inteiro entre 1 e 3. Utiliza MATCH/CASE para apresentar:**

– 1 → "Novo jogo"

– 2 → "Carregar jogo"

– 3 → "Sair"

– qualquer outro valor → "Opção inválida"

NOVO JOGO	CARREGAR JOGO	SAIR	OPÇÃO INVÁLIDA

2 **Cria um programa que leia uma nota inteira entre 0 e 20 e atribua uma classificação qualitativa, usando MATCH/CASE com condições (if), caso a nota seja:**

– < 10 → "Reprovado"

– 10 a 13 → "Suficiente"

– 14 a 16 → "Bom"

– 17 a 20 → "Muito Bom"

– fora do intervalo → "Nota inválida"

3 Um aluno vai para a escola utilizando diferentes meios de transporte. Cria um programa que:
peça ao utilizador para escolher um meio de transporte através de um número:
– 1 → Autocarro
– 2 → Táxi
– 3 → Bicicleta
– 4 → A pé
Usa **MATCH/CASE** para mostrar a mensagem:
– "Transporte escolhido: Autocarro"
– "Transporte escolhido: Táxi"
– "Transporte escolhido: Bicicleta"
– "Transporte escolhido: A pé"
– se o número não for válido, mostrar "Opção inválida"

4 Cria um programa que peça ao utilizador para escrever o turno do dia ("manhã", "tarde" ou "noite"). Usa **MATCH/CASE** para mostrar a saudação correta:
– "manhã" → "Bom dia"
– "tarde" → "Boa tarde"
– "noite" → "Boa noite"
– se o utilizador escrever outro valor, o programa deve mostrar "Turno inválido"

Testa os teus conhecimentos

[Tarefa integradora 1]

Tema – Gestão de água em Cabo Verde

Objetivo:

Aplicar conhecimentos sobre estruturas condicionais em Python (**IF**, **ELSE**, **ELIF**, operadores lógicos e expressão ternária).

Nota:

Antes de escreveres o código, é aconselhável planeares e desenhares o teu algoritmo no papel, de forma a organizares o raciocínio e antecipares possíveis erros. Além disso, deves segmentar o código com comentários, para explicar cada parte e facilitar a sua compreensão.

Contexto:

A água é um recurso escasso em Cabo Verde. Para promover o uso responsável, pretende-se desenvolver partes de um programa que ajude as famílias a acompanharem o seu consumo de água mensal.

Pretende-se criar um programa em Python que, com base nos dados fornecidos pelo utilizador, seja capaz de:

- avaliar se o consumo mensal de água está dentro do limite recomendado;
- aplicar tarifas diferentes com base em escalões de consumo;
- emitir mensagens específicas usando condições simples, compostas, **elif** e expressão ternária.

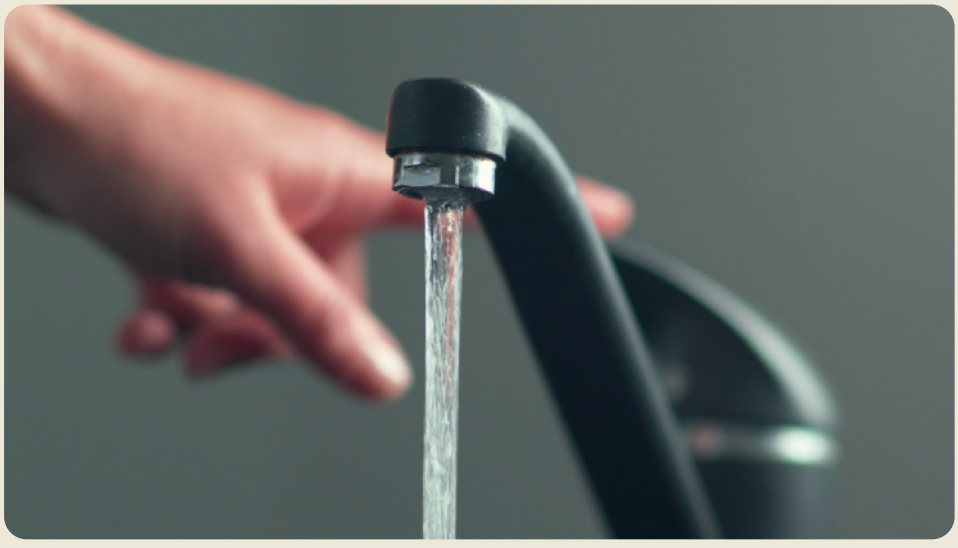
Testa os teus conhecimentos

O programa deve receber:

- o consumo mensal de água (em litros);
- a ilha onde o utilizador reside (por exemplo, Santiago, Sal, São Vicente, Santo Antão...);
- a situação do agregado familiar: "normal" ou "vulnerável".

Com base nessas informações, o programa deve aplicar:

- **IF**: para indicar se o consumo está abaixo ou acima do limite recomendado (6000 L/mês por agregado);
- **ELSE**: caso esteja acima, emitir alerta de consumo excessivo;
- **ELIF**: aplicar diferentes escalões tarifários:
  – até 6000 L: tarifa base
  – entre 6001 e 10 000 L: tarifa média
  – acima de 10 000 L: tarifa agravada
- **Condições compostas**: se o agregado for vulnerável e estiver dentro do escalão-base, aplicar desconto especial;
- **Expressão ternária**: se a ilha for Sal ou Boa Vista, definir a variável mensagem_ilha, com "Ilha com forte *stress* hídrico". Caso contrário, definir como "Ilha com *stress* hídrico moderado".



[Tarefa integradora 2]

Tema – Gestão do consumo de energia elétrica em Cabo Verde

Objetivo:

Aplicar conhecimentos sobre estruturas condicionais (**IF**, **ELIF**, **ELSE**, operadores lógicos, expressão ternária e **MATCH/CASE**).

Antes de escrever o código, deves planear o algoritmo no papel, identificar decisões e condições e segmentar o código com comentários explicativos.

Contexto:

Em Cabo Verde, a eletricidade tem um custo elevado e a produção depende, em grande parte, de recursos limitados. Para incentivar o uso responsável de energia, pretende-se criar um programa simples que ajude as famílias a analisarem o seu consumo mensal de eletricidade.

Dados introduzidos pelo utilizador:

O programa deve pedir:

- o consumo mensal de energia (em kWh);
- a ilha de residência;
- o tipo de habitação – "normal" e "social".

Regras a aplicar:

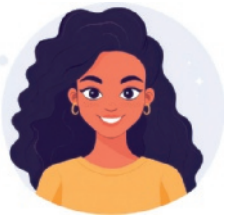
- **IF/ELSE**: caso se verifique que o consumo está dentro do limite recomendado (150 kWh/mês); se estiver acima, emitir um aviso de consumo elevado;
- **ELIF** – Escalões de consumo:
  – até 150 kWh → consumo baixo
  – de 151 a 300 kWh → consumo médio
  – acima de 300 kWh → consumo elevado
- **Condições compostas**: caso a habitação seja social e os consumos sejam baixos, aplicar benefício social;
- **MATCH/CASE** – tipo de energia principal: o programa deve pedir ainda o tipo de energia mais usada na casa: "solar", "elétrica" ou "mista".
  – Usar **MATCH/CASE** para apresentar uma mensagem adequada;
- **Expressão ternária** – situação energética da ilha: caso a ilha seja Sal ou Boa Vista, definir "Ilha com elevado custo energético". Caso contrário, "Ilha com custo energético moderado".

2.2. Estruturas de repetição (FOR, WHILE)

Manual Interativo

Vídeo
Introdução à linguagem de programação Python – Ciclo for; Ciclo while

Em programação, precisamos frequentemente de executar o mesmo conjunto de instruções várias vezes recorrendo a estruturas de repetição, também chamadas de **loops**.



As **estruturas de repetição** são comandos que permitem **executar um bloco de código várias vezes**, de forma automática, enquanto uma condição for verdadeira ou para cada elemento de uma sequência.
Servem para evitar repetir o código manualmente e tornar os programas mais eficientes.

Estrutura FOR

A estrutura **FOR** é usada para **repetir um bloco de código para cada elemento** de uma sequência (lista, string, tupla, range).
É uma das estruturas mais poderosas e simples da linguagem.

☑ Not@ que:

O **FOR** da linguagem Python é um ciclo **que percorre elementos** e não um ciclo que controla um índice como em muitas linguagens de programação.

Sintaxe

```
for variável in sequência:
    bloco_de_código
```

– variável: variável que recebe cada item da sequência;

– sequência: algo que pode ser percorrido (iterado) (lista, string, range, tuplo…);

– bloco_de_código: o que será executado em cada repetição.

Quando aplicada a sequências numéricas, a estrutura **FOR** é frequentemente combinada com a **função range()**, que permite gerar um intervalo de números inteiros com base em parâmetros de início, fim e passo (incremento ou decremento).

Durante o processo, a ordem dos elementos é rigorosamente respeitada. Se estivermos a percorrer uma **lista de palavras ou os caracteres de uma frase**, o ciclo processará cada item individualmente, do primeiro ao último, até que a sequência se esgote.

Esta abordagem torna o código mais legível e eficiente, pois automatiza a transição entre os elementos e elimina a necessidade de gerir manualmente o incremento de um índice ou verificar a condição de paragem, como acontece no ciclo **while**, estudado no próximo ponto.

Sintaxe

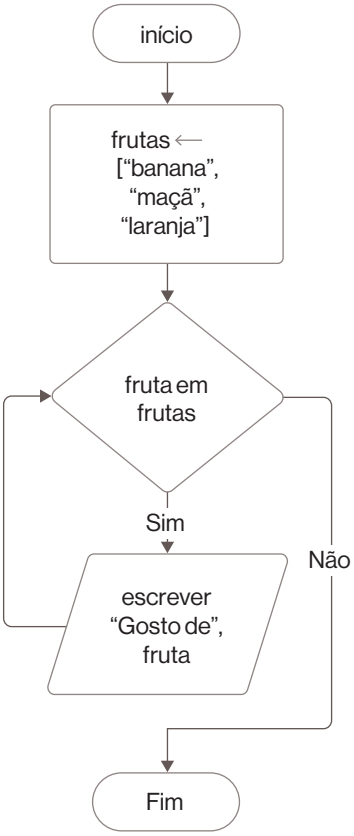
FOR em lista

```
for variável in list:
    bloco_de_código
```

Exemplo

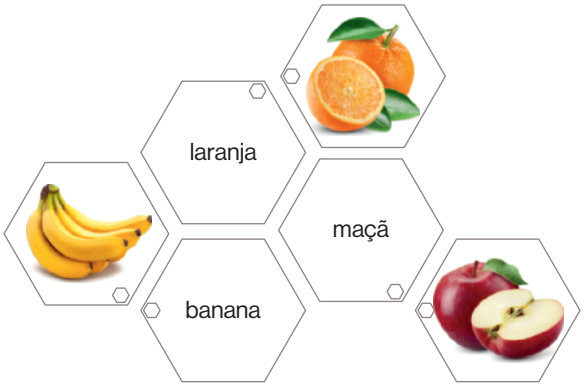
FOR com uma lista

Através do recurso a uma estrutura de repetição, pretende-se que o programa, com base numa lista de frutas, apresente uma mensagem repetida do tipo "Gosto de X", percorrendo as frutas da lista para definir a fruta "X".



Traçagem

	Frutas	Saída
1.º passo	**banana** maçã laranja	Gosto de banana
2.º passo	banana **maçã** laranja	Gosto de maçã
3.º passo	banana maçã **laranja**	Gosto de laranja



Linguagem Python

```
1   frutas = ["banana", "maçã", "laranja"]
2
3   for fruta in frutas:
4       print("Gosto de", fruta)
```

Output

```
Gosto de banana
Gosto de maçã
Gosto de laranja
```

Através da iteração sobre a **lista frutas**, a **variável de controlo fruta** assume o valor de cada elemento da sequência. Isto permite processar e mostrar cada item individualmente, concatenando-o (juntando) com a expressão "Gosto de" em cada repetição.

☑ Not@ que:

Os tipos de dados **tuple**, **dict** e **set** também podem ser usados no ciclo **for**, à semelhança do tipo **list**, como será apresentado no capítulo 3.

<Modo ON #14>

Cria uma lista com cinco cidades de Cabo Verde. Em seguida, escreve um programa que percorra essa lista utilizando a estrutura de repetição FOR e mostre cada cidade acompanhada da mensagem: "Gostaria de visitar: <cidade>".

OUTPUT PRETENDIDO

```
Gostaria de visitar: Praia
Gostaria de visitar: Mindelo
Gostaria de visitar: Santa Maria
Gostaria de visitar: Espargos
Gostaria de visitar: São Filipe
```

A **função range()** é utilizada em conjunto com a estrutura de repetição **FOR** para produzir uma série de números inteiros, baseando-se nos limites configurados nos seus argumentos.

Sintaxe

range ([start,] stop [,step])

– **start**: parâmetro opcional identificado pelas **[ ]**, primeiro número da sequência, por defeito é igual a 0;

– **stop**: parâmetro obrigatório, último número do intervalo, excluído da sequência;

– **step**: parâmetro opcional, pode ser positivo (incremento) ou negativo (decremento). Determina o passo a ser seguido no próximo valor da sequência. Por defeito é igual a 1.

Exemplos de formatos que a função range pode assumir e respetivas sequências de números gerados:

Função	Sequência	Interpretação
range(6)	0,1,2,3,4,5	**apenas o stop**: começa, por padrão, no 0 e vai até 5 (6 itens).
range(2,10)	2,3,4,5,6,7,8,9	**início e fim**: começa no 2 e para antes de chegar a 10.
range(2,10,2)	2,4,6,8	**com step**: começa no 2, aumenta de 2 em 2 e para antes de 10.
range(10,2,-1)	10,9,8,7,6,5,4,3	**step regressivo**: começa no 10 e "desce" de um em um até ao 3 (para antes do 2).
range(0,10,-2)	vazia	a direção do **step** deve ser a mesma direção do início para o fim.

Exemplo

FOR com um range

```
1   # Mostrar números de 1 a 5
2   for i in range(1, 6):
3       print("Número:", i)
```

Output

```
Número: 1
Número: 2
Número: 3
Número: 4
Número: 5
```

Neste exemplo, a função **range**(1, 6) é utilizada para gerar uma sequência numérica de 1 a 5. O intervalo começa no 1 (**start**) e termina em 6 (**stop**), sendo este último valor exclusivo. Como o incremento (**step**) não foi especificado, a função assume o valor-padrão de 1. Durante o ciclo, a variável de controlo **i** assume cada valor da sequência, exibindo no ecrã a mensagem **"Número:"** seguida do respetivo número.

<Modo ON #15>

Escreve um programa em Python que peça ao utilizador um número inteiro positivo *n*.

O programa deve então utilizar um ciclo for com range() para mostrar a tabuada do número escolhido, desde 1 até 10.

OUTPUT PRETENDIDO

```
Introduz um número inteiro positivo: 12

Tabuada do 12:
1  x 12 = 12
2  x 12 = 24
3  x 12 = 36
4  x 12 = 48
5  x 12 = 60
6  x 12 = 72
7  x 12 = 84
8  x 12 = 96
9  x 12 = 108
10 x 12  = 120
```

Exemplo

FOR com uma string

```
1   palavra = "Python"
2
3   for letra in palavra:
4       print(letra)
```

Output

```
P
y
t
h
o
n
```

Neste exemplo, a **variável palavra** (do tipo **str**, **string** – cadeia de caracteres) atua como o objeto iterável que controla o ciclo. Durante a execução, cada *caractere* da **string** é atribuído sequencialmente à **variável de controlo letra**, permitindo que cada letra seja exibida individualmente no ecrã.

Sintaxe

FOR em string

for variável in str:
    bloco_de_código

<Modo ON #16>

Escreve um programa em Python que peça ao utilizador uma frase.

O programa deve utilizar um ciclo **for** para contar quantas vogais existem na frase (a, e, i, o, u – maiúsculas ou minúsculas). No final, o programa deve apresentar o total de vogais encontradas.

OUTPUT PRETENDIDO

```
Introduz uma frase: Python é interessante
A frase contém 6 vogais.
```

Estrutura WHILE

O **WHILE** é uma **estrutura de repetição** em Python que executa repetidamente um bloco de código enquanto uma condição lógica for verdadeira.
É ideal quando não sabemos de antemão quantas vezes o ciclo deve ser repetido, ao contrário do **for**, que normalmente percorre uma sequência com número conhecido de elementos.

Sintaxe

```
while condição:
    # bloco de código a executar enquanto a condição for verdadeira
    Instruções
```

– condição: expressão lógica que retorna **True** ou **False**;

– instruções: código que será repetido enquanto a condição for **True**;

– bloco de código: deve ser indentado corretamente (normalmente quatro espaços ou tecla *Tab*).

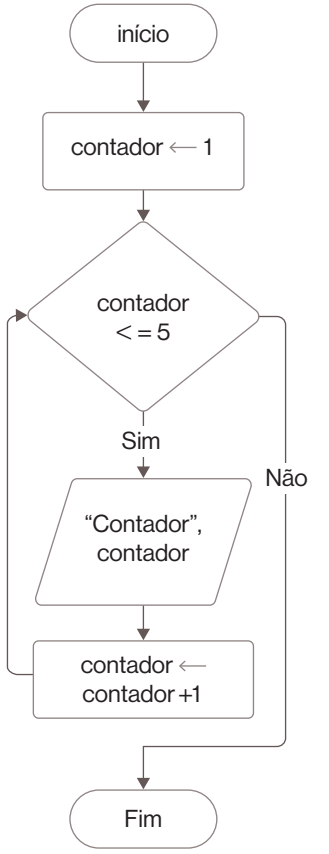
Exemplo

WHILE

Traçagem

	Contador	Contador ⟵ Contador +1	Saída
1.º passo	1	---	Contador: 1
2.º passo	---	2	Contador: 2
3.º passo	---	3	Contador: 3
4.º passo	---	4	Contador: 4
5.º passo	---	5	Contador: 5
6.º passo	---	6	---

Fluxograma



Linguagem Python

```python
contador = 1

while contador <= 5:
    print(f"Contador: {contador}")
    contador += 1  # atualiza a variável para evitar loop infinito
```

Output

```
Contador: 1
Contador: 2
Contador: 3
Contador: 4
Contador: 5
```

O código demonstra o funcionamento de um ciclo **while**, começando pela inicialização da variável contador com o valor **1**. Esta variável é fundamental, pois serve de base para a condição de controlo do **loop**, que determina que o bloco de código interno deve ser executado enquanto o valor do contador for **menor ou igual a 5**.

Durante cada repetição, o programa utiliza uma **f-string** para imprimir no ecrã a mensagem **"Contador:"** acompanhada pelo valor atual da variável. Logo após a impressão, a instrução **contador += 1** (ou seja, contador=contador+1) incrementa o valor da variável em uma unidade. Este passo é vital para a lógica do programa, pois garante que a condição de paragem seja eventualmente atingida, evitando que o sistema entre num **loop infinito**. O ciclo termina assim que o contador atinge o valor **6**, momento em que a condição lógica deixa de ser verdadeira.

<Modo ON #17>

1 Escreve um programa em Python que peça ao utilizador para introduzir números inteiros positivos.
O programa deve continuar a pedir números enquanto o utilizador não digitar 0.
No final, o programa deve mostrar:
– quantos números foram introduzidos (excluindo o zero);
– a soma de todos os números introduzidos.

OUTPUT PRETENDIDO

```
Introduz um número inteiro (0 para terminar): 2
Introduz um número inteiro (0 para terminar): 3
Introduz um número inteiro (0 para terminar): 4
Introduz um número inteiro (0 para terminar): 2
Introduz um número inteiro (0 para terminar): 0
Foram introduzidos 4 números.
A soma dos números é 11.
```

2 Escreve um programa que peça ao utilizador um número inteiro positivo e faça uma contagem regressiva até zero, apresentando cada número no ecrã.
No final, o programa deve mostrar a mensagem "FIM!".

OUTPUT PRETENDIDO

```
Introduz um número inteiro positivo: 5
5
4
3
2
1
0
FIM!
```

3 Escreve um programa que peça ao utilizador um número entre 1 e 10. O programa deve continuar a pedir o número enquanto o utilizador não inserir um valor válido.

OUTPUT PRETENDIDO

```
Introduz um número entre 1 e 10: 4
Número válido introduzido: 4
```

```
Introduz um número entre 1 e 10: 17
Número inválido! Tenta novamente.
```

4 **Escreve um programa que tenha uma palavra secreta. O programa deve pedir ao utilizador para adivinhar a palavra. O ciclo continua enquanto a palavra introduzida estiver errada.**
No final, o programa deve congratular o utilizador.

OUTPUT PRETENDIDO

```
Adivinha a palavra secreta: PYTHON
Errado! Tenta novamente.
```

```
Adivinha a palavra secreta: python
Parabéns! A palavra está correta.
```

5 **Cria um programa que exiba um menu com três opções:**

[1] – Saudação (imprime "Olá!")

[2] – Data (imprime "2026")

[3] – Sair

O programa deve continuar a pedir uma opção ao utilizador até que ele escolha a opção 3. Se ele escolher 1 ou 2, o programa executa a ação e volta a mostrar o menu.

6 **Cria um programa que permita ao utilizador inserir vários números positivos. Quando o utilizador inserir um número negativo, o programa deve parar e exibir a média aritmética de todos os números positivos inseridos anteriormente.**

7 **Escreve um programa que peça ao utilizador uma nota de exame (entre 0 e 20). Enquanto o utilizador digitar um valor inválido (por exemplo, -5 ou 25), o programa deve exibir "Nota inválida! Digite novamente:" e continuar a pedir o valor até que este esteja no intervalo correto.**

8 **Digita o seguinte código no Python e comenta-o instrução a instrução:**

```
2   n = int(input("Número: "))
3   f = 1
4   i = n
5   while i > 1:
6       f *= i
7       i -= 1
8   print(f)
```

Testa os teus conhecimentos

[Tarefa integradora 1]

Tema – Gestão e exploração de informações sobre pratos típicos de Cabo Verde, utilizando listas, estruturas de repetição, condicionais, funções e **strings**.

Objetivo:

Desenvolver um programa que recolha informação do utilizador, analisa elementos armazenados numa lista e permita ao utilizador interagir através de tentativas até acertar numa resposta correta.

Contexto:

Imagina que estás a preparar um pequeno sistema que trabalha com pratos típicos de Cabo Verde. O programa deve recolher vários pratos, mostrar informações detalhadas sobre cada um e desafiar o utilizador a adivinhar um deles.

Regras a aplicar:

- guardar pratos numa lista: o programa deve criar uma lista com cinco posições fixas;
- analisar cada prato, usando um ciclo **for**. Para cada prato introduzido, mostrar a mensagem "Um prato típico de Cabo Verde: <prato>";
- contar quantas letras o prato tem;
- verificar se o prato começa com uma vogal (maiúscula ou minúscula);
- contabilizar quantos pratos começam com vogal;
- jogo das adivinhas com ciclo **while**: o programa deve pedir ao utilizador que tente adivinhar um dos pratos existentes na lista. O ciclo só termina quando o utilizador acertar. A cada tentativa errada, deve mostrar: "Tenta outra vez!". Contar o número total de tentativas;
- no final, o programa deve mostrar:
  - quantas tentativas foram necessárias para acertar;
  - a lista completa dos pratos;
  - quantos pratos começam com uma vogal;
  - depois de um jogo, este pergunta se quer jogar novamente.

OUTPUT PRETENDIDO

```
=== Mini-Projeto: Pratos Típicos de Cabo Verde ===

Quantos pratos queres introduzir? 3
Prato 1: cachupa
Prato 2: caldo de peixe
Prato 3: jagacida

Informações sobre os pratos:

Prato: cachupa
  - Letras (sem espaços): 7
  - Vogais: 3
  - Consoantes: 4

Prato: caldo de peixe
  - Letras (sem espaços): 12
  - Vogais: 6
  - Consoantes: 6

Prato: jagacida
  - Letras (sem espaços): 8
  - Vogais: 4
  - Consoantes: 4

Total de pratos que começam com vogal: 0

Tenta adivinhar um dos pratos!
Adivinha o prato secreto: frango grelhado
Tenta outra vez!
Adivinha o prato secreto: cachupa

Parabéns! Acertaste depois de 2 tentativas.
Lista completa dos pratos:
- cachupa
- caldo de peixe
- jagacida

Queres jogar novamente? (sim/não): sim

Quantos pratos queres introduzir? []
```

2.3. Controlo de fluxos (BREAK, CONTINUE)

Ao trabalhar com estruturas de repetição, pode surgir a necessidade de alterar o comportamento normal de um ciclo, interrompendo completamente o ciclo antes do fim ou mesmo ignorando uma determinada iteração e passar para a seguinte.

Controlo de fluxo BREAK

O **BREAK** é uma instrução de controlo de fluxo em Python usada para **interromper imediatamente um ciclo** (**for** ou **while**), mesmo que a condição do ciclo ainda não tenha terminado.
Quando o Python encontra um **break** dentro de um **for** ou **while**, ele para o ciclo naquele exato momento e continua a execução a partir da linha seguinte ao ciclo.
Serve para encerrar um **loop** quando já não é necessário continuar a percorrê-lo.

Sintaxe

Dentro da estrutura **FOR**

for variável in sequência:
  if condição:
    break

Exemplo

BREAK (dentro do FOR)

```
1   ilhas = ["Fogo","São Vicente", "Boa Vista",  "Santiago", "Sal"]
2
3   for ilha in ilhas:
4       print("A verificar:", ilha)
5       if ilha[0] == "S":   # Encontrou uma ilha começada por S
6           print("Primeira ilha começada por S encontrada:", ilha)
7           break            # Interrompe o ciclo
```

Output

```
A verificar: Fogo
A verificar: São Vicente
Primeira ilha começada por S encontrada: São Vicente
```

O programa começa por definir uma lista de ilhas de Cabo Verde. Ao iniciar o ciclo, a variável de controlo ilha percorre cada elemento da lista sequencialmente. Para cada item, o programa, primeiro imprime a mensagem **"A verificar:"** seguida do nome da ilha atual. Imediatamente a seguir, existe uma estrutura condicional que verifica se a primeira letra da ilha (**ilha[0]**) é igual a **"S"**.

O ponto crucial ocorre quando o ciclo chega a "São Vicente": como esta ilha começa por "S", a condição torna-se verdadeira, o programa imprime que a primeira ilha correspondente foi encontrada e executa o comando **break**. Esta instrução força a interrupção imediata de todo o ciclo, o que significa que as ilhas restantes na lista ("Boa Vista", "Santiago" e "Sal") nunca chegarão a ser verificadas ou impressas no ecrã.

Sintaxe

Dentro da estrutura **WHILE**

while condição:
  if condição:
    break

Exemplo

BREAK (dentro do WHILE)

```
1   print("Introduz números positivos. Se introduzires um número negativo, o programa termina.")
2
3   total = 0
4
5   while True:
6       num = int(input("Número: "))
7
8       if num < 0:     # condição para interromper o ciclo
9           print("Número negativo detetado. A encerrar...")
10          break       # interrompe o while
11
12      total = total + num
13
14  print("Soma final:", total)
```

Output

```
Introduz números positivos. Se introduzires um número negativo, o programa termina.
Número: 12
Número: 4
Número: 5
Número: -2
Número negativo detetado. A encerrar...
Soma final: 21
```

O programa começa por definir uma variável total com o valor **0**, que servirá para acumular a soma dos números introduzidos. Ao entrar no ciclo **while True**, a linguagem Python executa o bloco de código repetidamente sem uma condição de paragem inicial. Em cada volta, é solicitado ao utilizador que introduza um número através do comando **input**.

O ponto determinante do código é a estrutura condicional interna: se o utilizador introduzir um número negativo (**num < 0**), o programa imprime uma mensagem de encerramento e executa o comando **break**. Este comando interrompe imediatamente o ciclo infinito, saltando para a última linha do código para exibir a **"Soma final"**. Se o número for positivo, ele é somado ao total e o ciclo recomeça, pedindo um novo valor. É uma técnica comum para criar programas que correm continuamente até receberem um sinal específico de paragem.

Controlo de fluxo CONTINUE

O **CONTINUE** é uma instrução de controlo de fluxo usada dentro de ciclos (**for** ou **while**).
Quando é executada, **interrompe apenas a iteração atual do ciclo** e faz o programa **passar imediatamente para a próxima iteração**, ignorando todo o código que estaria abaixo dela dentro do ciclo.

Em resumo:
– salta o resto do código da repetição atual;
– continua o ciclo normalmente na próxima repetição;
– não termina o ciclo (ao contrário do **break**);
– serve para encerrar um **loop** quando já não é necessário continuar a percorrê-lo.

Sintaxe

Dentro da estrutura **FOR**

for variável in sequência:
  if condição:
    continue

Exemplo

CONTINUE (dentro do FOR)

```
1   for n in range(1, 11):
2       if n % 2 != 0:  # se for ímpar
3           continue    # ignora números ímpares
4       print("Número par:", n)
```

Output

```
Número par: 2
Número par: 4
Número par: 6
Número par: 8
Número par: 10
```

O ciclo **for** que percorre os números de 1 a 10. Para cada número, o programa verifica se ele é ímpar usando a condição **n % 2 != 0**. Quando essa condição é verdadeira, a instrução **continue** é acionada, fazendo com que o ciclo ignore o restante do código da iteração atual e passe imediatamente para o próximo número. Dessa forma, o **print** só é executado para os números pares, mostrando apenas os números 2, 4, 6, 8 e 10.

Sintaxe

Dentro da estrutura **WHILE**

while condição:
  if condição:
    continue

O **CONTINUE** aparece sozinho, sempre dentro de um ciclo, e salta diretamente para a próxima iteração, ignorando o código que viria depois dele na mesma passagem do ciclo.

Exemplo

CONTINUE (dentro do WHILE)

```
1   print("Introduz números positivos. Digita 0 para terminar.")
2
3   total = 0
4
5   while True:
6       num = int(input("Número: "))
7       if num < 0:
8           print("Número inválido! Só positivos são permitidos.")
9           continue   # Ignora o restante do ciclo e volta para o input
10      if num == 0:
11          break      # Sai do ciclo quando o utilizador digita 0
12      total = total + num
13
14  print("Soma dos números válidos:", total)
```

Output

```
Introduz números positivos. Digita 0 para terminar.
Número: 12
Número: 3
Número: 4
Número: 0
Soma dos números válidos: 19
```

No exemplo, o **while** é infinito (**while True**) e pede ao utilizador números positivos. Se o número for negativo, o **continue** faz com que o resto do ciclo seja ignorado, voltando imediatamente para a próxima iteração, sem somar o número. Se o número for zero, o **break** termina o ciclo. Os números positivos são somados normalmente.

<Modo ON #18>

Escreve um programa que simule o acesso a um cofre digital pedindo uma palavra-passe ao utilizador dentro de um **loop** infinito, mas que utiliza o comando **break** para encerrar o programa e exibir a mensagem "Acesso concedido" apenas quando a senha digitada for "Python123".

Testa os teus conhecimentos

[Tarefa integradora 1]

Tema – Jogo de adivinhação de números

Objetivo:

Criar um programa que desafie o aluno a adivinhar um número escolhido pelo computador.

Contexto:

O jogo simula uma situação em que o utilizador tenta adivinhar um número secreto.

Descrição do programa:

- o computador "pensa" num número aleatório entre 1 e 50;
- o utilizador tenta adivinhar esse número;
- o programa deve informar se o palpite é maior ou menor que o número secreto;
- o ciclo continua até o utilizador acertar;
- no final, o programa mostra o número de tentativas feitas.

Regras a aplicar:

- número secreto: definido no início do programa (por exemplo, 25);
- tentativas inválidas: números negativos ou fora do intervalo permitido (1 a 50) devem ser ignorados. O ciclo continua para pedir um novo palpite;
- fim do jogo: se o utilizador acertar no número secreto, o ciclo termina. Se o utilizador digitar 0, o programa encerra imediatamente;
- *feedback* ao utilizador: se o palpite for menor que o número secreto deve mostrar: "O número é maior!". Se o palpite for maior, tem de mostrar: "O número é menor!". Se acertar, mostrar: "Parabéns! Acertaste!";
- contador de tentativas: apenas os palpites válidos (entre 1 e 50) são contados. Mostrar no final o total de tentativas válidas realizadas.

Manual Interativo

Vídeos

Maioria absoluta em Python

Juros simples e juros compostos em Python

Testa os teus conhecimentos

OUTPUT PRETENDIDO

```
=== Jogo de Adivinhação ===
Tenta adivinhar o número secreto entre 1 e 50.
Se quiseres sair, digita 0.
Tenta adivinhar o número: 33
O número é menor!
Tenta adivinhar o número: 30
O número é menor!
Tenta adivinhar o número: 20
O número é maior!
Tenta adivinhar o número: 25
Parabéns! Acertaste!
Número de tentativas válidas: 4
```

[Tarefa integradora 2]

Tema – Tabuada interativa

Objetivo:

Criar um programa que permita ao utilizador ver a tabuada de qualquer número inteiro positivo, aplicando conceitos de ciclo **for**, **while**, **break** e **continue**.

Contexto:

O programa simula uma calculadora educativa de tabuadas, permitindo que o utilizador pratique multiplicações e aprenda a controlar o fluxo do programa de forma interativa.

O programa deve receber:

- um número inteiro positivo fornecido pelo utilizador;
- a opção de sair do programa digitando 0.

Regras a aplicar:

- entrada do número: o utilizador deve digitar um número inteiro positivo. Se o número digitado for negativo ou inválido, o programa deve mostrar uma mensagem de erro e pedir outro número. Se o número for 0, o programa encerra imediatamente;
- criação da tabuada de 1 a 10. Mostrar cada resultado no formato: 1 x 3 = 3;
- repetição do programa: após mostrar a tabuada, o programa pergunta se o utilizador quer ver outra tabuada. Se a resposta for diferente de sim, o programa termina.

OUTPUT PRETENDIDO

```
=== Calculadora de Tabuada ===
Se quiseres sair, digita 0.
Introduz um número positivo: 3
1  x 3 = 3
2  x 3 = 6
3  x 3 = 9
4  x 3 = 12
5  x 3 = 15
6  x 3 = 18
7  x 3 = 21
8  x 3 = 24
9  x 3 = 27
10 x 3  = 30
Tabuada concluída.

Queres ver outra tabuada? (sim/não): sim
Introduz um número positivo:
```


3

Estruturas de dados compostos. Modularização

3.1. Listas e tuplas

3.2. Conjuntos e dicionários

3.3. Matrizes

3.4. Funções e módulos

3 Estruturas de dados compostos. Modularização

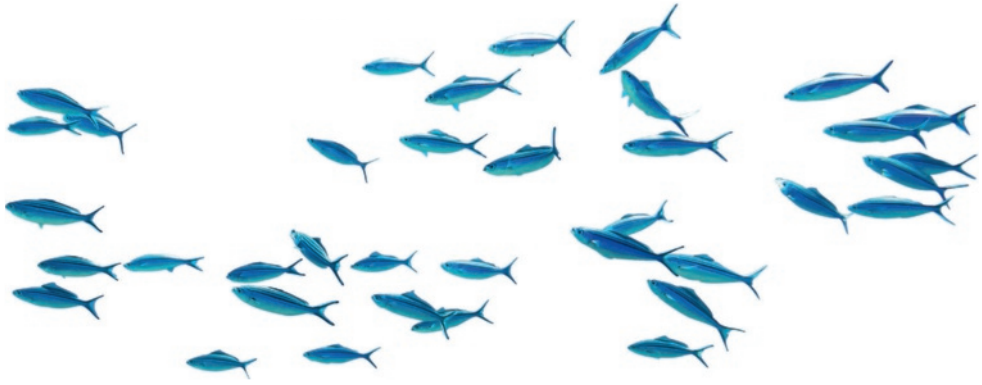
No final deste capítulo, deverás ser capaz de:

- Distinguir, criar e manipular listas e tuplas.
- Utilizar operações básicas (*append*, *remove*, *pop*, *insert*, *sort*).
- Criar e usar dicionários.
- Utilizar métodos para manipular dados em dicionários (*keys*, *values*, *items*).
- Construir e usar operações com matrizes.
- Entender o conceito de função.
- Saber organizar programas em partes reutilizáveis.

3.1. Listas e tuplas

Em programação, raramente trabalhamos com um único dado.

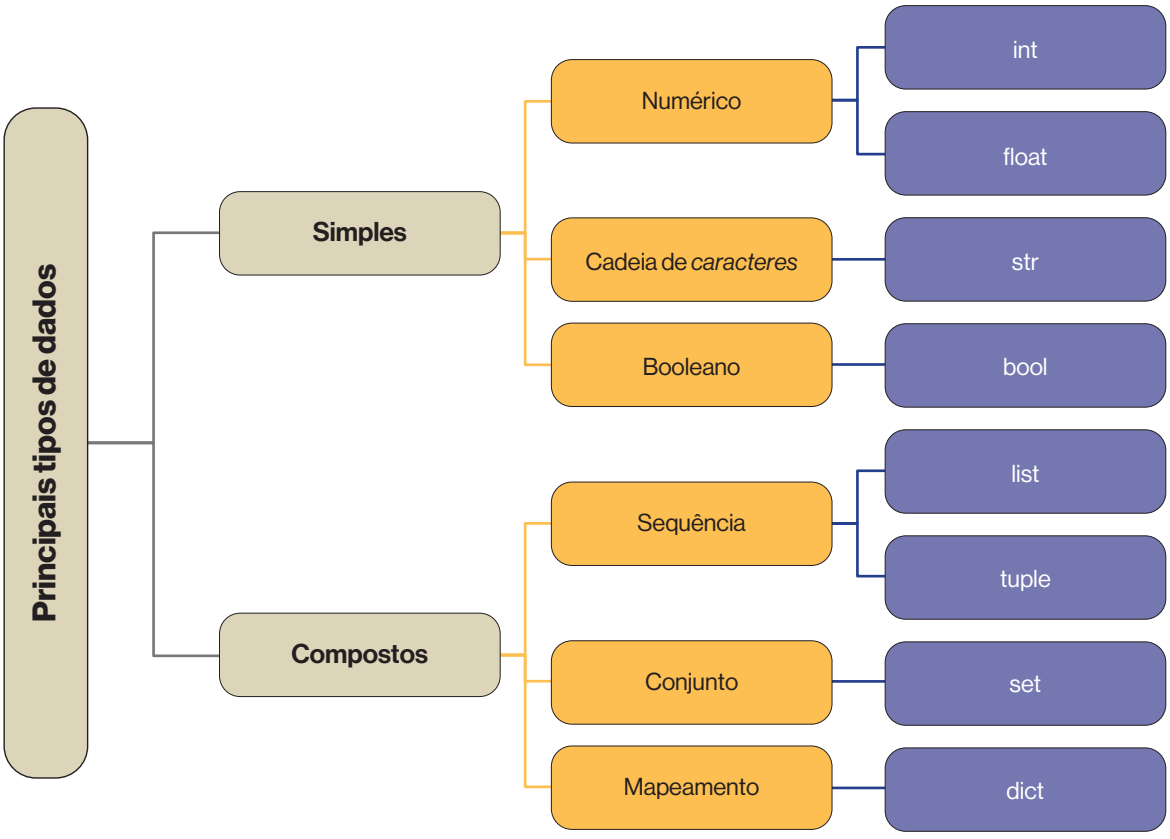
Normalmente, lidamos com conjuntos de dados, pelo que precisamos de guardar vários valores relacionados numa única variável, por exemplo, guardar as notas de um aluno, os nomes de jogadores de uma equipa de futebol ou as coordenadas de um ponto. Imagina que queres guardar os nomes de 50 peixes vendidos no mercado, não faz sentido criares 50 variáveis distintas, vais, por isso, usar uma **estrutura de dados**.



Estruturas de dados

As **estruturas de dados** (dados compostos) são o modo de organizar informações na memória do computador.
Em Python, existem diversos tipos: listas, tuplas, conjuntos, dicionários, entre outros. Cada um tem características próprias que os tornam mais ou menos adequados para tarefas específicas.

No capítulo anterior, desenvolvemos a criação de programas com dados do tipo simples; neste capítulo, iremos focar-nos nos dados compostos.



As listas e as tuplas são estruturas de dados fundamentais em Python. Permitem armazenar vários valores numa única variável, facilitando a organização e o tratamento da informação.

Listas (*list*)

As **listas** são coleções ordenadas e os seus elementos podem ser alterados, adicionados ou removidos. Podem conter elementos de qualquer tipo.
Para criar uma lista, devemos usar os parênteses retos [ ], com os itens no seu interior separados por vírgulas.

As **listas** são muito utilizadas em situações como o armazenamento de notas de alunos, listas de nomes, resultados de cálculos e qualquer conjunto de dados que possam variar ao longo do tempo.

Exemplo

```
# Criar uma lista de nomes (Strings)
alunos = ["Aline", "Beto", "Carlos"]

# Criar uma lista de números (Inteiros/Floats)
notas = [12, 15.5, 18, 10]

# Criar uma lista vazia (para preencher depois)
carrinho_compras = []
```

O exemplo começa por definir a lista **alunos**, que armazena um conjunto de **strings** representando nomes de pessoas.

Na instrução seguinte, mostra-se que uma lista pode conter diferentes tipos de dados numéricos simultaneamente, como acontece na variável **notas**, que mistura números inteiros e valores decimais (**floats**).

Por último, o programa mostra como declarar uma lista vazia, denominada **carrinho_compras**, uma técnica essencial para preparar uma estrutura que será preenchida dinamicamente durante a execução do código.

☑ Not@ que:

Para acederes a um item específico, deves usar o nome da lista seguido do índice entre parênteses retos **[ ]**.

Em programação, a contagem dos índices começa sempre no **zero**.

O índice negativo permite aceder aos elementos a partir do fim da lista.
O último elemento da lista é o -1.

Exemplo

```
ilha = ["Sal", "Boa Vista", "Maio"]

print(ilha[0])  # Imprime: Sal
print(ilha[1])  # Imprime: Boa Vista
print(ilha[2])  # Imprime: Maio
```

O exemplo apresenta o conceito de **indexação**, que começa por definir uma lista chamada **ilha** que contém três elementos. A contagem das posições numa lista inicia-se sempre no **zero**, o que significa que o primeiro item, **"Sal"**, é acedido através do índice **[0]**. O programa utiliza a instrução **print(ilha[0])** para exibir este primeiro valor e, sucessivamente, utiliza os índices **[1]** e **[2]** para aceder e mostrar **"Boa Vista"** e **"Maio"**, respetivamente.

Elemento →	Sal	Boa Vista	Maio
Índice →	0	1	2

Manipulação das listas

O Python disponibiliza vários métodos que facilitam a manipulação das listas. Os métodos são funções associadas à própria lista. Como, por exemplo:

append()
Adiciona um elemento no final da lista

insert()
Insere um elemento numa posição específica

index()
Indica a posição (índice) de um elemento numa lista

remove()
Remove um elemento pelo seu valor

pop()
Remove um elemento pela sua posição

len()
Devolve o número de elementos da lista


```
# Criação da lista inicial com dois materiais escolares
materiais = ["Caneta", "Lápis"]
print("Lista inicial:", materiais)

# Insere "Pasta"
materiais.append("Pasta")
print("Após a utilização do append:", materiais)

# Insere "Borracha" na posição 1
materiais.insert(1, "Borracha")
print("Após a utilização do insert:", materiais)

# Remove o item "Lápis" da lista
materiais.remove("Lápis")
print("Após a utilização do remove:", materiais)

# Remove o elemento que está na posição 0
material_removido = materiais.pop(0)
print("Material removido com pop:", material_removido)
print("Após a utilização do pop:", materiais)

# Devolve o número de elementos da lista
quantidade = len(materiais)
print("Número de materiais na lista:", quantidade)
```

OUTPUT

```
Lista inicial: ['Caneta', 'Lápis']
Após a utilização do append: ['Caneta', 'Lápis', 'Pasta']
Após a utilização do insert: ['Caneta', 'Borracha', 'Lápis', 'Pasta']
Após a utilização do remove: ['Caneta', 'Borracha', 'Pasta']
Material removido com pop: Caneta
Após a utilização do pop: ['Borracha', 'Pasta']
Número de materiais na lista: 2
```

O programa inicia com a criação de uma lista denominada **materiais**, à qual são atribuídos dois elementos iniciais: **"Caneta"** e **"Lápis"**.

De seguida, é utilizado o **método append()**, que tem como função adicionar um novo elemento ao final da lista. Ao executar a instrução **materiais.append("Pasta")**, o interpretador de Python identifica a última posição ocupada na lista e insere o novo elemento imediatamente a seguir sem alterar a ordem dos elementos previamente existentes.

> ☑ **Not@ que:**
>
> Esta operação evidencia a natureza mutável e dinâmica das listas em Python, que podem crescer ou ser modificadas durante a execução do programa.

Após a inserção do novo elemento, o **método insert()** pode ser utilizado para adicionar um item numa posição específica da lista. Ao usar **materiais.insert(1, "Borracha")**, o Python desloca os elementos a partir da posição indicada para a direita, inserindo o novo valor sem eliminar os dados existentes.

Em seguida, o **método remove()** permite eliminar um elemento da lista com base no seu valor. Quando se executa **materiais.remove("Lápis")**, o interpretador procura a primeira ocorrência desse valor na lista e remove-a.

Já o **método pop()** remove um elemento com base na sua posição (índice). Ao utilizar, **materiais.pop(0)**, o elemento localizado na primeira posição da lista é removido e devolvido, podendo ser armazenado numa variável. Este procedimento torna o **método pop()** especialmente útil quando se pretende não só remover mas também reutilizar o valor eliminado.

Por fim, a **função len()** é aplicada para determinar o número total de elementos presentes na lista no momento da execução. Esta função **len (materiais)** devolve

um valor inteiro que representa o comprimento da lista, permitindo controlar o tamanho da estrutura de dados e apoiar decisões lógicas no programa, como ciclos ou condições.

Exemplo

```
1   frutas = ["Maçã", "Banana", "Laranja"]
2
3   # Descobrir a posição da Banana
4   posicao = frutas.index("Banana")
5
6   print("A Banana está no índice:", posicao)
```

OUTPUT

```
A Banana está no índice: 1
```

O exemplo cria a lista **frutas** com três elementos: **"Maçã"**, **"Banana"** e **"Laranja"**. A linha **frutas.index("Banana")** procura a posição de **"Banana"** na lista e retorna o índice **1**, porque é o segundo elemento. Esse valor é guardado na variável **posicao** e depois exibido no ecrã com **print**.

<Modo ON #19>

1 **Vais organizar um torneio de futebol entre as ilhas. Precisas de um programa para gerir a lista de equipas inscritas.**

- **a)** Cria uma lista chamada equipas, com os nomes: "Santiago", "São Vicente" e "Sal".
- **b)** Verifica quantas equipas estão inscritas inicialmente.
- **c)** Uma nova equipa da ilha do "Fogo" chegou para se inscrever. Usa o método adequado para a adicionar ao final da lista.
- **d)** Alguém escreveu o nome da segunda equipa (índice 1) errado. Altera o valor no índice 1 de "São Vicente" para "Mindelenses".
- **e)** A equipa do "Sal" teve um problema no transporte e desistiu. Deve ser retirada da lista.
- **f)** Apresenta a lista final e a quantidade total de equipas que irão participar no torneio.

2 **Cria um programa que manipule uma lista de números inteiros. Começa por criar uma lista com cinco números à escolha e que mostre o primeiro e o último elemento. De seguida, altera o terceiro elemento da lista para um novo valor e apresenta a lista atualizada. Depois, adiciona um novo elemento no final da lista e mostra novamente a lista. Por fim, elimina um elemento específico e apresenta a lista final. Em cada etapa, adiciona comentários claros que indiquem o que está a ser feito e o estado atual da lista para acompanhar passo a passo todas as alterações.**

OUTPUT PRETENDIDO

```
Lista inicial: [10, 20, 30, 40, 50]
Primeiro elemento: 10
Último elemento: 50
Após alterar o terceiro elemento: [10, 20, 35, 40, 50]
Após adicionar um novo elemento: [10, 20, 35, 40, 50, 60]
Após eliminar um elemento: [10, 35, 40, 50, 60]
Lista final: [10, 35, 40, 50, 60]
```

Tuplas (*tuple*)

As **tuplas** são estruturas de dados semelhantes às listas, mas com uma diferença fundamental: são imutáveis. Isto significa que, depois de criadas, não é possível alterar os seus valores.
As tuplas são criadas com parênteses curvos ( ).

As **tuplas** são ideais para representar dados fixos, como dias da semana, meses do ano ou coordenadas.

Sabias que...

É muito comum percorrer os elementos de uma **lista** ou **tupla** utilizando ciclos. O ciclo **for** é o mais utilizado para esta finalidade

Exemplo

```
1   dias = ('segunda', 'terça', 'quarta', 'quinta', 'sexta')
2   for d in dias:
3       print(d)
```

OUTPUT

```
segunda
terça
quarta
quinta
sexta
```

Esse código utiliza um ciclo **for** para percorrer a **tupla dias** de forma sequencial. A cada repetição (iteração) do ciclo, a variável temporária **d** assume o valor do próximo elemento da coleção — começa por **'segunda'** e termina em **'sexta'** (sendo equivalente colocar-se **'** ou **"** na **string**). O comando **print(d)**, dentro do bloco, instrui o Python a exibir cada um desses dias individualmente, um por baixo do outro.

☑ Not@ que:

A escolha entre **lista** e **tupla** depende do problema a resolver.
Se os dados não devem ser alterados, a **tupla** é a melhor opção.
As **tuplas** são mais seguras e ocupam menos memória do que as **listas**.

<Modo ON #20>

1 **Cria uma tupla com cinco províncias de Cabo Verde: Santo Antão, São Vicente, Sal, Santiago e Fogo.**

- **a)** Visualiza o segundo elemento da tupla.
- **b)** Tenta alterar um dos elementos da tupla, "Sal" para "Brava", e observa o erro gerado pelo Python.
- **c)** Converte a tupla para lista.
- **d)** Determina o número de elementos da lista, confirmando que se mantêm os cinco elementos inicialmente definidos.

2 **Desenvolve um programa que, dado uma tupla composta por diversos números inteiros (positivos e negativos),**

```
numeros = (12, -5, 45, 78, 2, -10, 33)
```

seja capaz de identificar e escrever no ecrã o maior e o menor elemento contidos na mesma.

3 **Escreve um programa que analise uma tupla de temperaturas.**

```
temperaturas = (15.5, 18.0, 21.2, 14.8, 19.5, 22.0, 16.5)
```

O programa deve calcular a média e identificar as temperaturas acima dessa média.
No final, o programa deve exibir no ecrã a média (com duas casas decimais), a temperatura mais alta e a lista das temperaturas que ficaram acima da média.

Operações comuns das listas e tuplas

Algumas operações são comuns a várias estruturas de dados, em particular às **listas** e **tuplas**, permitindo aceder a elementos, verificar a sua existência, contar itens ou percorrer toda a coleção.

Além das operações, o Python disponibiliza várias funções nativas que permitem realizar cálculos e análises rapidamente.

As **listas**, por serem estruturas mutáveis, possuem ainda métodos específicos para adicionar ou remover elementos, ordenar ou inverter a ordem dos mesmos.

Vamos ver algumas dessas funções, aplicadas sempre que possível a **tuplas** e a **listas**, e organizadas de acordo com a ação que se pretende executar.

O que faz	Lista	Tupla	Exemplo
Criar	**l = [10, 20, 30]** (cria uma lista com três valores 10, 20, 30)	**t = (10, 20, 30)** (cria uma tupla com três valores 10, 20, 30)	[10, 20, 30] (10, 20, 30)

O que faz	Lista	Tupla	Exemplo
Retorna o valor de um elemento pelo índice	**a_minha_lista [0]** (aponta para o item da posição 0)	**t[1]** (aponta para o item da posição 1)	l[0] → 10 t[1] → 20
Substitui o valor de um elemento	**a_minha_lista [1] = 5** (atribui o valor 5 ao item que está na posição 1)	X	[10, 20, 30] l[1]=5 → [10, 5, 30]
Insere um novo valor no final ou em posição específica	**a_minha_lista append(4)** (adiciona o valor 4 na última posição) **a_minha_lista insert(1, 9)** (insere o valor 9 na posição 1)	X	[10, 20, 30] **l.append(4)** → [10, 20, 30, 4] **l.insert(1, 9)** → [10, 9, 20, 30]
Retorna a posição do primeiro elemento igual ao valor	**a_minha_lista index(30)** (índice do valor 30)	**t.index(30)** (índice do valor 30)	**l.index(30)** → 2 **t.index(30)** → 2

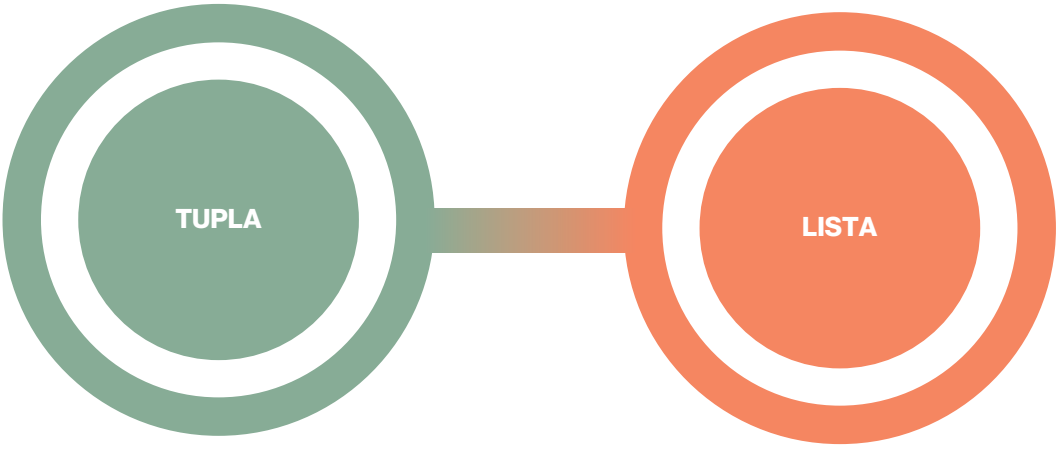
O que faz	Lista	Tupla	Exemplo
Conta quantas vezes um elemento aparece	**l.count(20)** (número de vezes que surge o valor 20)	**t.count(20)** (número de vezes que surge o valor 20)	**l.count(20)** → 1
Retorna o número de elementos	**len(l)** (quantos elementos tem a lista l)	**len(t)** (quantos elementos tem a tupla t)	[10, 20, 30] **len(l)** → 3

O que faz	Lista	Tupla	Exemplo
Verifica se um elemento está presente	**20 in l** (o valor 20 existe na lista l?)	**30 in t** (o valor 30 existe na tupla t?)	**20 in l** → **True** **30 in t** → **True**
Itera sobre cada elemento da estrutura	**for x in l:** (procura x na lista l)	**for x in t:** (procura x na tupla t)	–

O que faz	Lista	Tupla	Exemplo
Remove o primeiro elemento que corresponda ao valor indicado	**l.remove(20)** (apaga o valor 20)	X	[10, 20, 30] **l.remove(20)** → [10, 30]
Remove e retorna o elemento de um índice específico	**Valor = l.pop(1)** (remove da lista o valor que está na posição 1 e guarda-o em memória)	X	[10, 20, 30] **l.pop(1)** → 20 [10, 30]
Remove o elemento ou a lista inteira pelo índice	**del l[1]** (remove o valor que está na posição 1) **del l** (remove toda a lista)	X	[1, 2, 3]; **del** l[1] → [1, 3]

O que faz	Lista	Tupla	Exemplo
Retorna o menor elemento	**min(l)** (valor mais pequeno da lista l)	**min(t)** (valor mais pequeno da tupla t)	[30, 10, 20] **min(l)** → 10
Retorna o maior elemento	**max(l)** (valor maior da lista l)	**max(t)** (valor maior da tupla l)	[30, 10, 20] **max(l)** → 30
Retorna a soma dos elementos (numéricos)	**sum(l)** (soma dos valores da lista l)	**sum(t)** (soma dos valores da tupla t)	[10, 20, 30] **sum(l)** → 60

O que faz	Lista	Tupla	Exemplo
Organiza os elementos em ordem crescente	**l.sort()** (organiza os valores da lista l de forma crescente)	X	[30, 10, 20] **l.sort()** → [10, 20, 30]
Retorna uma nova lista ordenada sem modificar a original	**x=sorted(l)** (cria uma nova lista x com os valores da lista l organizados de forma crescente)	**x=sorted(t)** (cria uma nova tupla x com os valores da lista l organizados de forma crescente)	[30, 10, 20] **x=sorted(l)** → [10, 20, 30]
Inverte a ordem atual dos elementos sem alterar os valores	**l.reverse()** (organiza os valores da lista l de forma decrescente)	X	[10, 20, 30] **l.reverse()** → [30, 20, 10]



No exemplo seguinte, vamos aplicar algumas das operações apresentadas, utilizando **listas** e **tuplas** para manipular conjuntos de dados.

Exemplo

```
# Dados iniciais (Tupla)
notas_oficiais = (12, 15, 8, 18, 12, 10)
print("Tupla",notas_oficiais)

# Conversão para Lista e Append
notas_trabalho = list(notas_oficiais)
notas_trabalho.append(14)
print("Lista",notas_trabalho)

# Insert (índice 1, valor 20)
notas_trabalho.insert(1, 20)
print("Lista atualizada",notas_trabalho)
```

```
13
14  # Max, Min, Sum
15  print(f"Maior: {max(notas_trabalho)}")
16  print(f"Menor: {min(notas_trabalho)}")
17  print(f"Soma: {sum(notas_trabalho)}")
18
19  # Count e Index
20  print(f"O 12 aparece {notas_trabalho.count(12)} vezes.")
21  print(f"O 18 está no índice: {notas_trabalho.index(18)}")
22
23  # Remove (pelo valor)
24  notas_trabalho.remove(8)
25  print("Lista atualizada",notas_trabalho)
26
27  # Pop (retira o último valor)
28  nota_descartada = notas_trabalho.pop()
29  print("Lista atualizada",notas_trabalho)
30
31  # Del (apaga o índice 3)
32  del notas_trabalho[3]
33  print("Lista atualizada",notas_trabalho)
34
35  # Sorted (Cria uma cópia ordenada)
36  print(f"Lista para visualização: {sorted(notas_trabalho)}")
37  print(f"Lista original ainda está assim: {notas_trabalho}")
38
39  # Sort e Reverse
40  notas_trabalho.sort() # Ordena de forma crescente
41  print("Lista atualizada",notas_trabalho)
42  notas_trabalho.reverse() # Inverte os valores de forma decrescente
43  print("Lista atualizada",notas_trabalho)
```

OUTPUT

```
Tupla (12, 15, 8, 18, 12, 10)
Lista [12, 15, 8, 18, 12, 10, 14]
Lista atualizada [12, 20, 15, 8, 18, 12, 10, 14]
Maior: 20
Menor: 8
Soma: 109
O 12 aparece 2 vezes.
O 18 está no índice: 4
Lista atualizada [12, 20, 15, 18, 12, 10, 14]
Lista atualizada [12, 20, 15, 18, 12, 10]
Lista atualizada [12, 20, 15, 12, 10]
Lista para visualização: [10, 12, 12, 15, 20]
Lista original ainda está assim: [12, 20, 15, 12, 10]
Lista atualizada [10, 12, 12, 15, 20]
Lista atualizada [20, 15, 12, 12, 10]
```

<Modo ON #21>

Vais criar um sistema para uma loja. Os preços-base dos produtos vêm de uma base de dados guardada numa tupla. Para aplicares promoções e gerires a montra do dia, decides converter esses dados numa lista.

a) Cria uma tupla chamada **precos_tabela** com os seguintes valores (em euros): (450, 800, 150, 1200, 800, 300).

b) Cria uma lista chamada **stock_dia** a partir da tupla anteriormente criada.

c) Adiciona um novo produto de 600 euros ao final da lista.

d) Chegou um produto *premium* que deve aparecer logo no início da lista com o valor de 2500 euros.

e) Identifica e exibe o preço mais caro, o mais barato e o valor total do inventário em loja.

f) Quantos produtos de 800 euros existem em *stock*?

g) Qual é a posição do primeiro produto que custa 1200 euros?

h) Um cliente comprou o produto de 150 euros. Remove esse valor da lista.

i) Remove o último produto da lista porque foi transferido para outra loja e guarda o seu valor numa variável **produto_transferido**.

j) O item que está no índice 4 foi retirado por estar avariado. Elimina-o permanentemente.

k) Exibe os preços ordenados do mais barato para o mais caro para um folheto publicitário, mas mantém a lista original intacta.

l) Organiza a tua lista de *stock* permanentemente por ordem crescente e depois inverte a ordem para que os mais caros apareçam primeiro.

Aninhamento em listas

Sabias que...

As **listas aninhadas** são muito usadas para organizar dados de forma hierárquica ou irregular.

Listas de árvore

– Hierarquia
– Pastas dentro de pastas
– Categorias e subcategorias

Listas irregulares

– *Ragged lists*
– Comprimentos diferentes
– Quantidade variável de dados

Listas de tuplas

– Cada linha é imutável (fixa), mas a lista pode crescer

```
# Uma lista que guarda: [Categoria, [Subcategorias]]
menu = [
    "Ficheiro", ["Novo", "Abrir", "Guardar"],
    "Editar", ["Copiar", "Colar"]
]
```

```
# Participantes em diferentes workshops
workshops = [
    ["Ana", "Bruno"],                # Workshop A (2 pessoas)
    ["Carla", "Daniel", "Eduardo"],  # Workshop B (3 pessoas)
    ["Filipa"]                       # Workshop C (1 pessoa)
]
```

```
# Lista de registos de clientes: (ID, Nome, Pontos)
clientes = [
    (101, "Alice", 500),
    (102, "Tiago", 320),
    (103, "Sofia", 890)
]
# Para aceder ao nome do segundo cliente
print(clientes[1][1])          # Tiago
```

Análise do exemplo **listas de árvore**

```
1   # Uma lista que guarda: [Categoria, [Subcategorias]]
2   menu = [
3       "Ficheiro", ["Novo", "Abrir", "Guardar"],
4       "Editar", ["Copiar", "Colar"]
5   ]
```

Menu			
Categorias	**Subcategorias**		
Ficheiro	**Novo**	**Abrir**	**Guardar**
Índice 0	**Índice 1**		
	Índice 0	**Índice 1**	**Índice 2**
Editar	**Copiar**	**Colar**	
Índice 2	**Índice 3**		
	Índice 0	**Índice 1**	

– **menu [0]** → Ficheiro;

– **menu [1]** → Novo, Abrir, Guardar;

– **menu [1][1]** → Abrir;

– **menu [3][0]** → Copiar;

– **menu [0][3]** → h.

<Modo ON #22>

Partindo do exemplo acima apresentado, indica o que é devolvido pelo Python:

a) menu [2];

b) menu [3][1];

c) menu [1][0];

d) menu [2][1].

Análise do exemplo **listas irregulares**

```
# Participantes em diferentes workshops
workshops = [
    ["Ana", "Bruno"],                  # Workshop A (2 pessoas)
    ["Carla", "Daniel", "Eduardo"],    # Workshop B (3 pessoas)
    ["Filipa"]                         # Workshop C (1 pessoa)
]
```

Neste exemplo, não existem categorias. A lista principal *workshops* contém apenas outras listas.

Workshops			
Categorias	**Subcategorias**		
	Ana	Bruno	
	Índice 0		
	Índice 0	Índice 1	
	Carla	Daniel	Eduardo
	Índice 1		
	Índice 0	Índice 1	Índice 2
	Filipa		
	Índice 2		
	Índice 0		

– **workshops [0]** → Ana, Bruno;

– **workshops [0][0]** → Ana;

– **workshops [1][1]** → Daniel;

– **workshops [3][0]** → Erro!;

– **workshops [2][1]** → Filipa.

<Modo ON #23>

Partindo do exemplo acima apresentado, indica o que é devolvido pelo Python:

a) workshops [1][3];

b) workshops [1];

c) workshops [0][2].

Análise do exemplo **listas de tuplas**

```
1   # Lista de registos de clientes: (ID, Nome, Pontos)
2   clientes = [
3       (101, "Alice", 500),
4       (102, "Tiago", 320),
5       (103, "Sofia", 890)
6   ]
7   # Para aceder ao nome do segundo cliente
8   print(clientes[1][1])           # Tiago
```

As tuplas são imutáveis. Isto significa que podes ler os dados do Tiago, mas não podes alterá-los individualmente como fazias com as listas. Não existem categorias.

Clientes			
Categorias	**Subcategorias**		
	101	Alice	500
	Índice 0		
	Índice 0	Índice 1	Índice 2
	102	Tiago	320
	Índice 1		
	Índice 0	Índice 1	Índice 2
	103	Sofia	890
	Índice 2		
	Índice 0	Índice 1	Índice 2

– **clientes [0]** → 101, Alice, 500;

– **clientes [0][0]** → 101;

– **clientes [2][1]** → Sofia.

<Modo ON #24>

Partindo do exemplo acima apresentado, indica o que é devolvido pelo Python:

a) clientes [1][3];

b) clientes [2][2];

c) clientes [3].

Testa os teus conhecimentos

1 **Para cada uma das questões seguintes, assinala a opção correta.**

1.1. Qual é a sintaxe correta para criar uma lista e uma tupla, respetivamente?

a) L = (1, 2) e T = [3, 4]

b) L = [1, 2] e T = (3, 4)

c) L = {1, 2} e T = [3, 4]

d) L = [1, 2] e T = {3, 4}

1.2. O que acontece se tentares executar **minha_tupla[0] = 10** numa tupla já existente?

a) O primeiro elemento é alterado para 10 com sucesso.

b) O valor 10 é adicionado ao final da tupla.

c) O Python gera um erro de tipo (*TypeError*) porque as tuplas são imutáveis.

d) A tupla é convertida automaticamente numa lista.

1.3. Qual método de lista que removia o valor "Azul" de cores = ["Verde", "Azul", "Vermelho"]?

a) cores.pop("Azul")

b) cores.delete("Azul")

c) cores.remove("Azul")

d) cores.discard("Azul")

2 **Completa as frases com os termos ou métodos corretos.**

2.1. Para adicionar um elemento ao final de uma lista, utilizamos o método _____.

2.2. A escolha entre lista e tupla depende da natureza dos dados. Se precisa de uma estrutura inalterável para garantir maior segurança e menor consumo de _____, deve optar pela tupla.

2.3. O operador _____ permite verificar se um determinado item existe dentro de uma lista ou tupla (por exemplo, **if** "maçã" _____ frutas:).

3 **Classifica em verdadeira (V) ou falsa (F) cada uma das afirmações seguintes.**

3.1. As listas em Python são ordenadas, o que significa que os itens têm uma ordem definida que não muda a menos que seja solicitado.

3.2. As tuplas ocupam, geralmente, mais espaço na memória do que listas equivalentes.

Testa os teus conhecimentos

3.3. É possível ter uma lista dentro de outra lista (aninhamento).

3.4. O **método .sort()** pode ser aplicado tanto a listas como a tuplas para ordenar os seus elementos.

4 **Associa a operação à esquerda com o resultado/descrição à direita, considerando x = [10, 20, 30, 40].**

Operação	Descrição/Resultado
(A) x[1]	**1.** Retorna o valor 20
(B) len(x)	**2.** Retorna o valor 40
(C) x[-1]	**3.** Retorna o tamanho da lista (4)

[Tarefa integradora 1]

Tema – Gestão avançada de lista de compras

Objetivo:

Compreender e aplicar o uso de **tuplas** e **listas**, distinguindo estruturas de dados imutáveis e mutáveis através da criação e gestão de uma lista de compras organizada por categorias.

Notas:

- As categorias devem ser armazenadas numa tupla, pois representam dados fixos.
- Os produtos devem ser armazenados em listas, uma vez que podem ser adicionados ou removidos ao longo da execução do programa.

Contexto:

Um utilizador pretende organizar as suas compras de forma simples e eficiente, separando os produtos por categorias. Para evitar alterações indevidas na estrutura principal da lista, as categorias são definidas previamente e não devem ser modificadas. No entanto, os produtos podem variar conforme as necessidades.

O programa deve receber:

- uma tupla com as categorias de compras (frutas, legumes, bebidas e higiene);

- uma lista de listas, em que cada sublista corresponde aos produtos de uma categoria;
- o nome de um produto a adicionar ou a remover;
- a categoria à qual o produto pertence.

Com base nessas informações, o programa deve aplicar:

- a associação entre cada categoria da tupla e a respetiva lista de produtos;
- a exibição da lista de compras organizada por categorias;
- a adição de um novo produto a uma categoria escolhida;
- a remoção de um produto existente;
- a contagem do número de produtos em cada categoria;
- o cálculo do número total de produtos da lista de compras;
- uma tentativa de alteração da tupla de categorias, permitindo observar o erro gerado e confirmar a sua imutabilidade.

OUTPUT PRETENDIDO

```
LISTA DE COMPRAS
-------------------------
Frutas:
 - Maçã
 - Banana
 - Laranja

Legumes:
 - Cenoura
 - Tomate
 - Alface

Bebidas:
 - Água
 - Sumo

Higiene:
 - Sabonete
 - Pasta de dentes

LISTA DE COMPRAS ATUALIZADA
-------------------------
Frutas:
 - Maçã
 - Banana
 - Laranja
 - Pera
```

```
LISTA DE COMPRAS ATUALIZADA
-------------------------
Frutas:
 - Maçã
 - Banana
 - Laranja
 - Pera

Legumes:
 - Cenoura
 - Alface

Bebidas:
 - Água
 - Sumo
 - Leite

Higiene:
 - Sabonete
 - Pasta de dentes

Número de produtos em Frutas : 4
Número de produtos em Legumes : 2
Número de produtos em Bebidas : 3
Número de produtos em Higiene : 2
Total de produtos na lista de compras: 11
```

Testa os teus conhecimentos

[Tarefa integradora 2]

Tema – Gestão de atividades

Objetivo:

Compreender e aplicar o uso de **tuplas** e **listas** através da criação de um programa que organiza os alunos e as suas participações em várias atividades.

Nota:

As atividades do clube serão armazenadas numa **tupla**, pois são fixas e não podem ser alteradas.

Os alunos inscritos em cada atividade e as suas pontuações serão armazenados em **listas**, pois podem ser atualizados com novas participações.

Contexto:

Um monitor quer acompanhar a participação dos alunos em várias atividades, como futebol, xadrez, teatro e música. Para cada atividade, será registado o nome dos alunos e a pontuação que cada um obteve. As atividades são fixas (tupla) e as listas permitem adicionar ou atualizar alunos e pontos.

O programa deve receber/permitir:

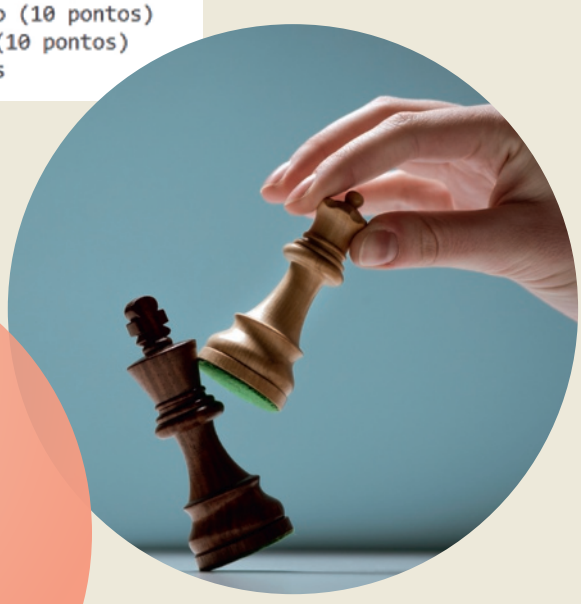
- uma tupla com o nome das atividades do clube;
- uma lista de listas, em que cada sublista corresponde a uma atividade e guarda os nomes dos alunos inscritos;
- uma lista paralela com a pontuação de cada aluno por atividade;
- adicionar alunos a uma atividade e registar a sua pontuação.

Com base nessas informações, o programa deve:

- associar cada atividade (tupla) à sua lista de alunos e pontuações;
- adicionar novos alunos ou atualizar pontuações;
- exibir todos os alunos e as suas pontuações por atividade;
- calcular a média de pontuação por atividade;
- determinar o aluno com a maior pontuação em cada atividade;
- mostrar a pontuação média geral do clube;
- demonstrar que a tupla de atividades é imutável, tentando alterar um elemento.

OUTPUT PRETENDIDO

```
RELATÓRIO DE ATIVIDADES
------------------------------
Futebol:
 - Ana: 8 pontos
 - Bruno: 9 pontos

Xadrez:
 - Carla: 7 pontos
 - Diogo: 10 pontos

Teatro:
 - Ana: 9 pontos
 - Carla: 8 pontos
 - Diogo: 10 pontos

Música:
 - Bruno: 7 pontos
 - Ana: 10 pontos

Média de Futebol: 8.5 pontos
Média de Xadrez: 8.67 pontos
Média de Teatro: 9.0 pontos
Média de Música: 8.5 pontos
Maior pontuação em Futebol: Bruno (9 pontos)
Maior pontuação em Xadrez: Diogo (10 pontos)
Maior pontuação em Teatro: Diogo (10 pontos)
Maior pontuação em Música: Ana (10 pontos)
Média geral do clube: 8.7 pontos
```



3.2. Conjuntos e dicionários

Os **conjuntos** e os **dicionários** são estruturas de dados não sequenciais, ou seja, não dependem de índices numéricos.

Conjuntos (*sets*)

Um **conjunto** é uma coleção de elementos únicos. Isto significa que **não podem existir valores repetidos** dentro de um **conjunto**.
Os **conjuntos** são criados com chavetas { } e são muito úteis para eliminar duplicações automaticamente.

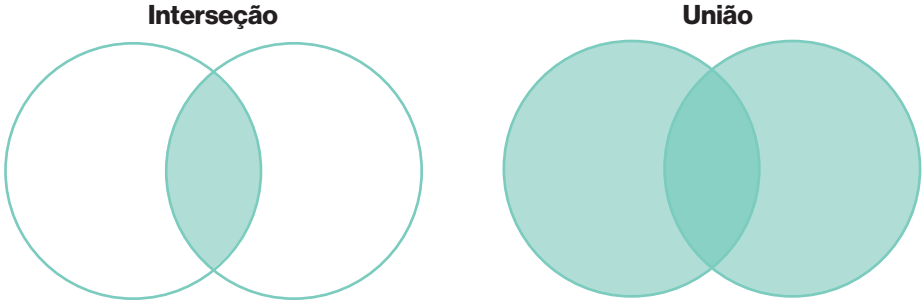
valores = {1, 2, 3, 3, 4}

Mesmo que o número 3 seja repetido, o conjunto armazenará apenas uma ocorrência.

Características principais

Sabias que...

Os **conjuntos** permitem operações matemáticas como **união (operador |)**, **interseção (operador &)** e **diferença**, sendo úteis em problemas de lógica e análise de dados.



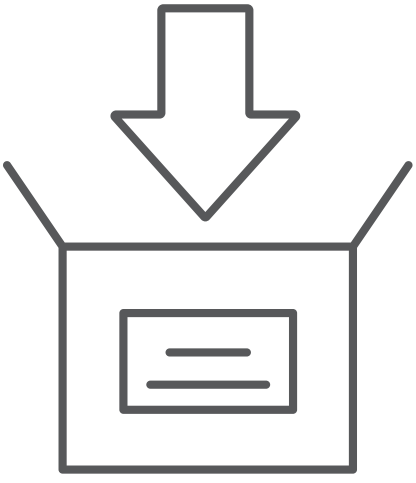
Para adicionar elementos num conjunto...

add()

Usa-se em **conjuntos**. Como os conjuntos não têm ordem, o elemento é colocado onde for mais eficiente para o computador.

append()

Usa-se em **listas**. Adiciona sempre ao final da lista.



Exemplo

```
1   # conjuntos de alunos com números de cartão
2   autorizados = {101, 102, 105, 110, 120, 135}
3   entraram_hoje = {102, 110, 150, 135}
4
5   # quem estava autorizado mas não compareceu
6   faltas = autorizados - entraram_hoje
7
8   # quem entrou sem autorização
9   intrusos = entraram_hoje - autorizados
10
11  # Total de pessoas que passaram pelo portão ou estão autorizadas
12  registo_total = autorizados | entraram_hoje
13
14  # Adicionar um novo aluno autorizado
15  autorizados.add(140)
16
17  # Resultados
18  print(f"Alunos autorizados: {autorizados}")
19  print(f"Alunos que faltaram: {faltas}")
20  print(f"Intrusos detetados: {intrusos}")
21  print(f"Lista completa de cartões: {registo_total}")
```

autorizados	**101**	**102**	**105**	**110**	**120**	**135**
entraram_hoje	**102**	**110**	**150**	**135**		
faltas (não entraram e estavam autorizados)	**101**		**105**		**120**	
intrusos (entraram e não estavam autorizados)			**150**			
registo_total (os que estavam autorizados com os que entraram)	**101** **102**	**110**	**105** **150**	**135**	**120**	

OUTPUT

```
Alunos autorizados: {101, 102, 135, 120, 105, 140, 110}
Alunos que faltaram: {120, 105, 101}
Intrusos detetados: {150}
Lista completa de cartões: {101, 102, 135, 105, 110, 150, 120}
```

<Modo ON #25>

Imagina que és o responsável técnico de uma empresa de *software*. Precisas de analisar as competências de duas equipas diferentes para decidir como alocar um novo projeto de Inteligência Artificial.

Cria um programa para realizar as seguintes tarefas:

a) Cria um conjunto chamado **equipa_front** com as competências: "HTML", "CSS", "JavaScript", "React".

b) Cria um conjunto chamado **equipa_back** com as competências: "Python", "Java", "SQL", "JavaScript", "C#".

c) Cria um novo conjunto chamado **tecnologias_totais** que contenha todas as competências da empresa (sem dados repetidos).

d) Descobre qual é a competência comum a ambas as equipas.

e) Identifica quais as competências que a **equipa_back** tem que a **equipa_front** não possui.

f) Adiciona a competência "Cloud" ao conjunto **tecnologias_totais** utilizando o método adequado.

g) Exibe no ecrã o resultado de cada operação.

Dicionários (*dict*)

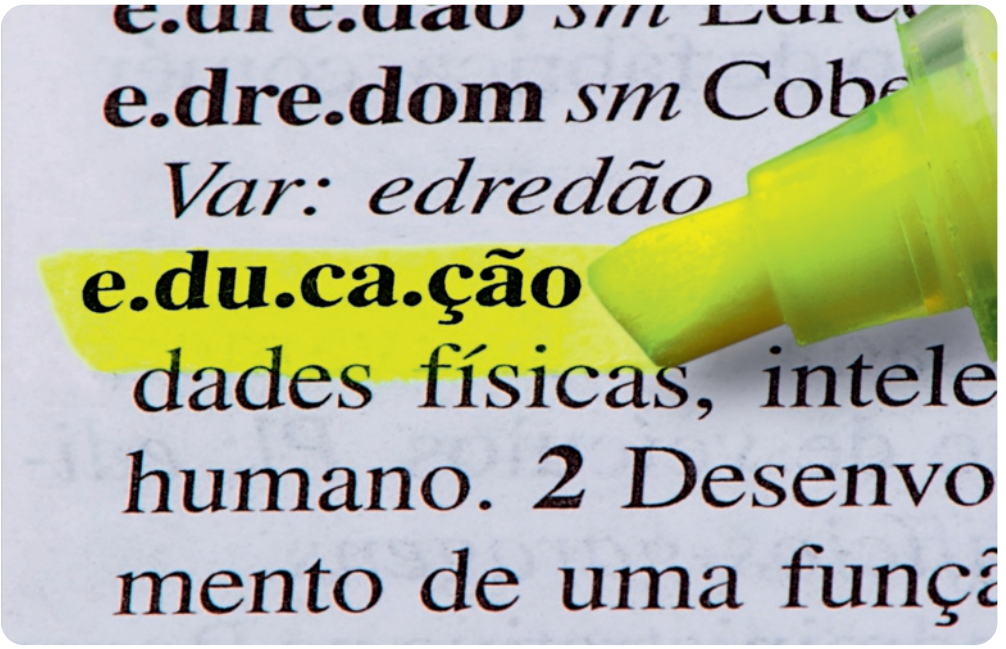
Os **dicionários** são, possivelmente, a estrutura de dados mais poderosa e versátil do Python.
São estruturas que armazenam informação em pares chave-valor. Uma das grandes vantagens dos dicionários é o acesso rápido aos valores através da chave.
Ao contrário das listas (que usam números para encontrar itens), os **dicionários** funcionam como um **mapeamento** de chaves para valores.

Sabias que...

Os **dicionários** são amplamente utilizados para representar entidades do mundo real, como alunos, professores, produtos ou registos administrativos.

O conceito fundamental: chave e valor

Pensa num dicionário real:
– procuras uma **palavra (chave)** para obter a sua **definição (valor)**.



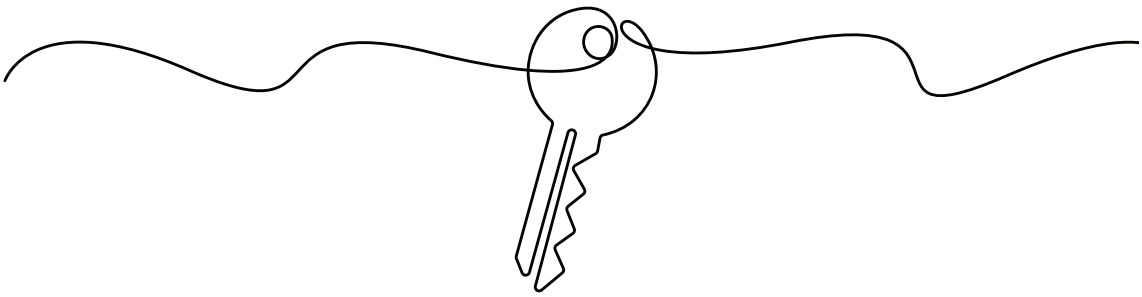
Sintaxe

dicionário = {chave: valor }

Exemplo

```
aluno = {
    "nome": "Ricardo",
    "idade": 22,
    "curso": "Engenharia"
}
```

O dicionário **aluno** funciona como uma ficha de identificação organizada através de pares de **chave** e **valor**.



Neste caso, as **chaves são as etiquetas descritivas** – **"nome"**, **"idade"** e **"curso"** – que nos permitem localizar informações específicas sem conhecermos a posição numérica delas, como aconteceria numa lista.

A cada uma destas chaves está **associado um valor concreto**:

– à chave **"nome"** corresponde o texto **"Ricardo"**;

– à **"idade"** o número inteiro **22**;

– e ao **"curso"** a **string "Engenharia"**.

<Modo ON #26>

Cria um dicionário para armazenar os teus dados: nome, idade, peso, altura e número de irmãos.

Regras de ouro das chaves

Devem ser únicas

Se repetires uma chave, o último valor escrito irá substituir o anterior.

Devem ser imutáveis

Podes usar **strings**, números ou tuplas como chaves.

Nunca podes usar listas, pois as listas podem mudar, o que quebraria o índice do dicionário.

Manipulação dos dicionários

A **manipulação de dicionários** em Python é um dos processos mais flexíveis da linguagem, pois estas estruturas são mutáveis.

Isto significa que, ao contrário das **tuplas**, um **dicionário** pode crescer, encolher ou mudar de conteúdo ao longo da execução do programa, adaptando-se às necessidades do momento.

```
del dict["chave"]
dict.pop("chave")
```
Apagar

```
dict["nova_chave"] = "valor"
```
Criar/Adicionar

```
dict["chave"] = "novo_valor"
```
Atualizar

```
print(dict["chave"])
```
Ler

Exemplo

```python
# CRIAR
stock = {
    "PS5": 10,
    "Xbox": 8,
    "Switch": 15
}
print(f"Stock inicial: {stock}")

# ADICIONAR
stock["PC"] = 5
print(f"Após adicionar PC: {stock}")

# LER
quantidade_ps5 = stock["PS5"]
print(f"Existem {quantidade_ps5} unidades de PS5 em armazém.")

# ATUALIZAR
stock["Xbox"] = 5
print(f"Após venda de Xbox: {stock}")

# APAGAR
# apaga diretamente
del stock["Switch"]
# apaga mas "devolve" o valor para uma variável
removido = stock.pop("PC")
print(f"Retirámos o PC do catálogo. O último stock era: {removido}")

print(f"Dicionário final: {stock}")
```

OUTPUT

```
Stock inicial: {'PS5': 10, 'Xbox': 8, 'Switch': 15}
Após adicionar PC: {'PS5': 10, 'Xbox': 8, 'Switch': 15, 'PC': 5}
Existem 10 unidades de PS5 em armazém.
Após venda de Xbox: {'PS5': 10, 'Xbox': 5, 'Switch': 15, 'PC': 5}
Retirámos o PC do catálogo. O último stock era: 5
Dicionário final: {'PS5': 10, 'Xbox': 5}
```

Métodos essenciais nos dicionários

.keys()
Devolve todas as chaves (ex: "nome", "idade").

.values()
Devolve todos os valores (ex: "Ricardo", 22).

.items()
Devolve os pares em formato de tuplas.
É o mais usado em ciclos **for**.

.get()
A forma segura de aceder a um valor.
Se a chave não existir, ele devolve **None** em vez de *crashar* o programa.


```
1  aluno = {
2      "nome": "Ricardo",
3      "idade": 22,
4      "curso": "Engenharia"
5  }
6  print("Chaves",aluno.keys())
7  print("Valores",aluno.values())
8  print("Chave-Valor",aluno.items())
9  print("Acesso",aluno.get("nome"))
10 print("Acesso",aluno.get(2))
```

OUTPUT

```
Chaves dict_keys(['nome', 'idade', 'curso'])
Valores dict_values(['Ricardo', 22, 'Engenharia'])
Chave-Valor dict_items([('nome', 'Ricardo'), ('idade', 22), ('curso', 'Engenharia')])
Acesso Ricardo
Acesso None
```

A importância do método .get()

Sintaxe

dicionario.get(chave, valor_padrao)
O **.get()** aceita dois argumentos (o segundo é opcional).

☑ Not@ que:

Enquanto o acesso direto (com **[ ]**) exige que a chave exista, o **.get()** lida com a incerteza de forma elegante. Portanto, o **get** dá estabilidade ao programa.

Exemplo

```
1    stock = {"maçãs": 10, "bananas": 5}
```

10	5

```
1    stock = {"maçãs": 10, "bananas": 5}
2
3    print(stock.get("maçãs"))      # devolve 10, a chave existe
4    print(stock.get("peras"))      # devolve None, a chave não existe. O programa não crasha
5    print(stock.get("peras", 0))   # devolve 0 valor definido, a chave não existe.
```

Característica	dict["chave"]	dict.get("chave")
Se a chave existe	Devolve o valor.	Devolve o valor.
Se não existe	Lança **KeyError**. (erro de execução, o programa para).	Devolve **None** (o código continua).
Uso ideal	Quando tens a certeza de que a chave existe.	Quando a chave é opcional ou incerta.

Aninhamento em dicionários (*Nested*)

☑ Not@ que:

Um dicionário pode conter outro dicionário lá dentro. Funciona como uma base de dados organizada em camadas.

<Modo ON #27>

Desenvolve o *software* de entrada de um ginásio. Precisas de guardar o nome e o tipo de plano (bronze, prata, ouro ou *platinum*) de cada sócio.

a) Cria um dicionário chamado sócios em que a chave é o número ID e o valor é outro dicionário com os dados do sócio.

b) Adiciona os seguintes sócios:

– ID 101: {"nome": "Tiago", "plano": "Prata"};

– ID 102: {"nome": "Paulo", "plano": "Ouro"};

– ID 103: {"nome": "Ana", "plano": "Bronze"}.

c) Um novo sócio acabou de se inscrever. Adiciona-o ao dicionário.

– ID 104: {"nome": "Carla", "plano": "Ouro"}.

d) O sócio de ID 101 passou o cartão na entrada. Faz a leitura desses dados.

e) A Carla (ID 104) decidiu fazer uma atualização ao seu plano. Altera o plano dela de "Ouro" para "*Platinum*".

f) O Tiago (ID 101) cancelou a subscrição. Remove-o completamente do sistema.

g) Percorre o dicionário e imprime uma frase amigável para cada sócio existente, por exemplo: "O sócio [Nome] tem o plano [Plano]".

Exemplo

```
1  utilizadores = {
2      "user_1": {"nome": "Ana", "email": "helena@email.com"},
3      "user_2": {"nome": "Bruno", "email": "bruno@email.com"}
4  }
```

No primeiro nível, temos a variável **utilizadores**, em que cada chave (como **"user_1"** ou **"user_2"**) atua como um identificador único para um perfil específico.

Ao olharmos para o interior de **"user_1"**, encontramos uma ficha detalhada com as suas próprias chaves e valores: o nome **"Ana"** e o *email* **"helena@email.com"**.

Para acederes a um dado específico, como o *email* do Bruno, deves navegar pelas camadas usando dois pares de **[ ]**.

```
6  print(utilizadores["user_2"]["email"])   # devolve bruno@email.com
```

Sabias que...

Antigamente, os dicionários não guardavam a ordem pela qual inserias os itens. No entanto, desde o **Python 3.7+**, os dicionários são oficialmente **ordenados**. Se inserires "A" e depois "B", ao imprimir, eles aparecerão sempre nessa ordem.

Ordenar um dicionário

Para **ordenar um dicionário**, usamos a **função sorted()**.

Imagina que temos os preços dos produtos e queremos visualizar do mais caro para o mais barato.

Exemplo

```
precos = {
    "Monitor": 150.00,
    "Teclado": 45.00,
    "Rato": 25.00,
    "Portátil": 850.00
}
# Ordenar pelos VALORES (preços) em ordem decrescente
produtos_ordenados = sorted(precos, key=precos.get, reverse=True)

print("Produtos ordenados por preço (Decrescente):")
for p in produtos_ordenados:
    print(f"{p}: {precos[p]}€")
```

OUTPUT

```
Produtos ordenados por preço (Decrescente):
Portátil: 850.0€
Monitor: 150.0€
Teclado: 45.0€
Rato: 25.0€
```




O dicionário **precos** armazena os pares dos produtos e os seus valores, mas a ordenação padrão do Python atua sobre os nomes (ordem alfabética).

Para contornar isso, a instrução **key=precos.get** direciona o programa para o conteúdo numérico (o preço) de cada chave durante a comparação.

Ao adicionar o parâmetro **reverse=True**, a lista resultante, **produtos_ordenados**, passa a listar os itens do mais caro para o mais barato, começando pelo **"Portátil"** e a acabar no **"Rato"**.

Percorrer um dicionário

Exemplo

```
1   precos = {"Maçã": 1.50, "Banana": 0.80, "Cereja": 3.00}
2
3   for produto, valor in precos.items():
4       print(f"O item {produto} custa {valor}€")
```

OUTPUT

```
O item Maçã custa 1.5€
O item Banana custa 0.8€
O item Cereja custa 3.0€
```

Este código demonstra a forma mais eficiente de percorrer um dicionário em Python utilizando o **método .items()** para extrair simultaneamente as duas partes de cada entrada.

O processo começa com a definição de um dicionário chamado **precos**, em que cada nome de fruta (a chave) está ligado ao seu respetivo valor numérico.

O ponto central aqui é o ciclo **for**, que utiliza o "desempacotamento" (ou *unpacking*), em cada volta do ciclo, o programa retira um par do dicionário e entrega a chave à variável **produto** e o valor à **variável valor**.

O resultado é uma listagem clara e automática de todo o inventário, na qual o programa imprime, linha a linha, o custo de cada item.

<Modo ON #28>

Considera um dicionário em que as chaves são os nomes dos alunos e os valores são as suas notas finais (de 0 a 20).

Pauta				
Tiago	Ana	Hugo	Maria	Sofia
15	19	12	18	10

a) Extrai e imprime apenas os nomes de todos os alunos que estão na pauta, para que o professor saiba quem fez o exame.

b) Extrai todas as notas e calcula a média da turma.

c) Percorre o dicionário e imprime uma lista personalizada. Para cada aluno, deve aparecer: "Aluno: [Nome] | Nota: [Valor]".

d) Identifica quem teve nota igual ou superior a 18 e imprime: "[Nome] recebeu uma Menção Honrosa!".

e) O professor quer verificar a nota de um aluno específico chamado "Ricardo". Como o Ricardo pode não estar na pauta, usa o **.get()** para que, se ele não for encontrado, o programa imprima: "Nota não registada".

f) Faz o mesmo para a "Ana", garantindo que a nota dela (19) é exibida corretamente.

Conversão automática: de uma lista de tuplas para um dicionário

Se tiveres dados em pares (ou tuplas), o Python consegue criar um dicionário rapidamente usando a **função dict()**.

Exemplo

```
1   dados_brutos = [("ID_1", "Ativo"), ("ID_2", "Inativo"), ("ID_3", "Ativo")]
2   dicionario_estados = dict(dados_brutos)  # conversão
3   print(dicionario_estados["ID_1"])        # devolve "Ativo"
```

Resumo das estruturas de dados

	Símbolo	Estrutura única?	Estrutura ordenada?	Exemplo de uso
Lista	[ ]	Não	Sim	Fila de espera, histórico.
Tupla	( )	Não	Sim	Coordenadas GPS, registos fixos.
Conjunto	{ }	Sim	Não	Filtrar duplicados, convidados únicos.
Dicionário	{k:v}	Chaves Sim	Sim (Python 3.7+)	Perfis de utilizador, inventários.

<Modo ON #29>

1 **Considera o seguinte dicionário com nomes e telefones:**

contactos = {"Ana": "912 323 113", "Pedro": "965 512 324", "Maria": "934 534 744"}

a) Adiciona um novo contacto: "Tiago" com o número "920 023 900".

b) Verifica se a "Joana" existe no dicionário. Se não existir, escreve "Contacto não encontrado".

2 **Uma loja recebeu uma lista de códigos de barras, mas muitos estão repetidos devido a um erro no *scanner*:**

codigos_sujos = [101, 202, 101, 303, 202, 404, 505, 101]

a) Cria uma nova variável chamada codigos_unicos que contenha cada código apenas uma vez.

b) Escreve quantos códigos únicos existem no total.

3 **Considera um dicionário com os nomes dos alunos e as suas notas finais:**

notas = {"Tiago": 15, "Vasco": 19, "Carla": 12, "Bruno": 18}

a) Escreve o programa para encontrar o nome do aluno com a nota mais alta.

b) Escreve o nome do aluno com a nota mais baixa.

4 **Dada a string: "banana", cria um dicionário que conte quantas vezes cada letra aparece na palavra.**

5 **Considera dois grupos de alunos inscritos em diferentes clubes extracurriculares:**

futebol = {"Tiago", "Ana", "Hugo", "Maria"}

teatro = {"Maria", "Hugo", "Ricardo", "Sofia"}

a) Descobre quais são os alunos que estão inscritos em ambos os clubes (Interseção).

b) Descobre quais são os alunos que estão apenas no clube de futebol (Diferença).

c) Cria uma lista com todos os alunos inscritos sem repetições (União).

Testa os teus conhecimentos

1 **Para cada uma das questões seguintes, assinala a opção correta.**

1.1. Se quiseres alterar o ano para 2023, qual é o comando correto? – carro = {"marca": "Ford", "modelo": "Mustang", "ano": 1964}.

(A) carro[2] = 2023

(B) carro["ano"] = 2023

(C) carro.update("ano" = 2023)

1.2. Observa este código:

```
1   stock = {"maçãs": 50, "bananas": 20}
2   item = "peras"
3
4   if item in stock:
5       print(stock[item])
6   else:
7       print("Não disponível")
```

Qual será o resultado final?

(A) *KeyError*

(B) *None*

(C) Não disponível

1.3. Tens a lista:

cores = ["azul", "verde", "azul", "vermelho", "verde"].

Qual é a forma mais rápida de obter apenas as cores únicas, sem repetições?

(A) list(set(cores))

(B) cores.unique()

(C) cores.remove_duplicates()

2 **Completa as frases com os termos ou métodos corretos.**

2.1. Para adicionar um item a um conjunto, usamos o método _____.

2.2. Para obter apenas as palavras-chave de um dicionário, usamos o método _____().

2.3. Para apagar todos os dados de um dicionário e deixá-lo vazio, usamos o método _____().

Testa os teus conhecimentos

2.4. Para evitar que o programa pare com um erro ao procurar uma chave que pode não existir, em vez de meu_dict["chave"], devemos usar o método meu_dict._____("chave"), que devolve *None* se a chave não for encontrada.

3 **Classifica em verdadeira (V) ou falsa (F) cada uma das afirmações seguintes.**

(A) Um conjunto permite guardar três vezes o número 7.

(B) Os conjuntos são úteis para remover duplicados de uma lista.

(C) O comando meu_set[0] devolve o primeiro item do conjunto.

(D) Posso colocar uma lista dentro de um conjunto (por exemplo, {1, 2, [3, 4]}).

(E) Posso colocar uma tupla dentro de um conjunto (por exemplo, {1, 2, (3, 4)}).

(F) Se eu transformar a lista [1, 2, 2, 3] num conjunto, o tamanho (len) do conjunto será 3.

4 **Associa a operação à esquerda com a descrição/resultado à direita, considerando dois conjuntos: A = {1, 2, 3} e B = {3, 4, 5}.**

Operação	Descrição/Resultado
(A) A & B (Interseção)	**1.** {1, 2, 3, 4, 5}
(B) A | B (União)	**2.** {1, 2}
(C) A - B (Diferença)	**3.** {3}

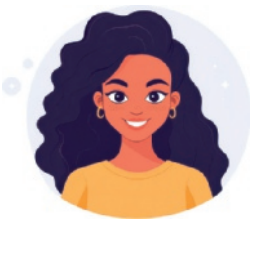
5 **Imagina a seguinte estrutura de dados:**

```
1  utilizadores = {
2      "admin": {"id": 1, "email": "admin@site.com"},
3      "user1": {"id": 2, "email": "tiago@mail.pt"}
4  }
```

Associa o comando ao que ele devolve.

Operação	Descrição/Resultado
(A) utilizadores["admin"]["email"]	**1.** {"id": 2, "email": "tiago@mail.pt"}
(B) utilizadores["user1"]	**2.** admin@site.com
(C) utilizadores["user1"]["id"]	**3.** 2

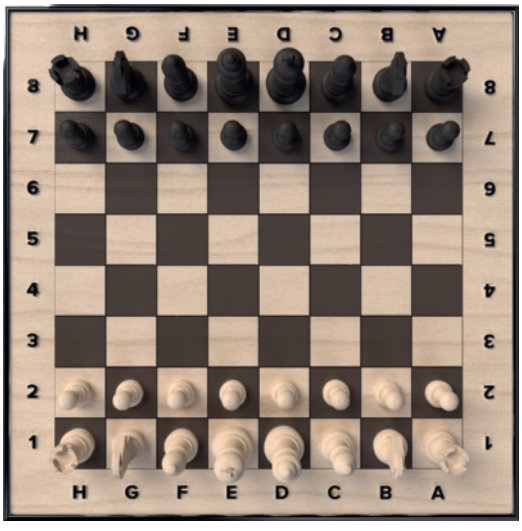
3.3. Matrizes

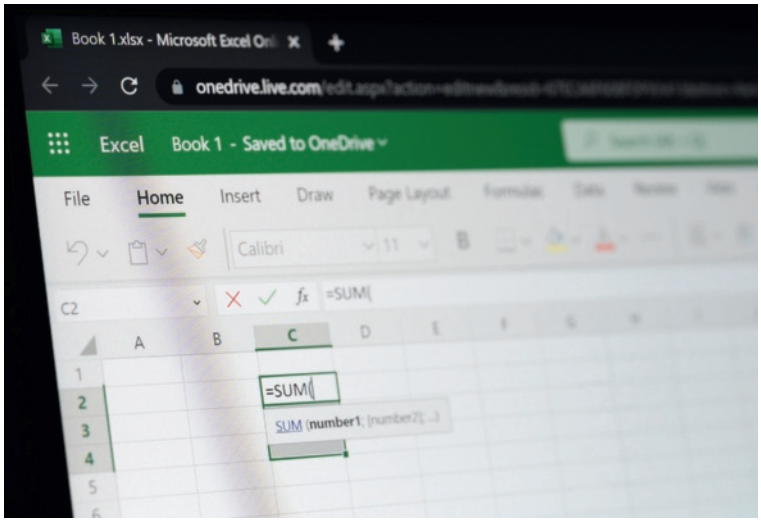


As **matrizes** (*array* bidimensional) são estruturas de dados bidimensionais muito utilizadas em matemática, física e informática.
Em Python, uma **matriz** é representada por uma lista de listas, em que cada lista interna representa uma linha.

Uma das formas mais simples de visualizar uma **matriz** é pensar num tabuleiro de xadrez que funciona como uma grelha de 8 × 8 em que cada peça ocupa uma posição definida pelo cruzamento de uma linha com uma coluna.

Da mesma forma, uma folha de cálculo de Excel® é uma representação prática e direta deste conceito; os dados são organizados em linhas numeradas (1, 2, 3...) e colunas identificadas por letras (A, B, C, etc.), permitindo que qualquer valor seja localizado através de uma coordenada específica, tal como acontece com os índices na programação.





Sabias que...

As **matrizes** são utilizadas em situações como tabelas de dados, tabuleiro de jogos, resultados de avaliações ou representações gráficas.

Exemplo

```
# matriz 2x3 (2 linhas, 3 colunas)
matriz = [
    [1, 2, 3],
    [4, 5, 6]
]

# aceder ao elemento "5"
print(matriz[1][1]) #(Linha 1, Coluna 1)
```

	Índice 0	Índice 1	Índice 2
Índice 0	1	2	3
Índice 1	4	5	6

Para aceder a um elemento da **matriz**, utilizam-se dois índices: o primeiro para a linha e o segundo para a coluna, tendo presente que os índices iniciam em 0.

Matrizes dinâmicas (*List comprehension*)

Em Python, podes criar matrizes de forma muito mais rápida usando **matrizes dinâmicas** (*List comprehension*). Este modo é útil quando precisas de inicializar uma grelha (por exemplo, um tabuleiro de jogo vazio).

Sabias que...

O uso do **range()** permite definir o tamanho da matriz dinamicamente sem teres de escrever todos os elementos à mão.

Exemplo

```
tabuleiro = [[10 for _ in range(4)] for _ in range(4)]

print(tabuleiro)
```

OUTPUT

```
[[10, 10, 10, 10], [10, 10, 10, 10], [10, 10, 10, 10], [10, 10, 10, 10]]
```

<Modo ON #30>

Pretendes configurar uma rede de sensores térmicos espalhados por uma sala de servidores, a sala está organizada numa grelha de cinco linhas por cinco colunas.

a) Cria uma matriz chamada **grelha_sensores** de tamanho cinco por cinco em que todos os valores iniciais sejam 0.0. Utiliza uma **List comprehension** para esta tarefa.

b) Simula o aquecimento de uma zona específica da sala. Altera os valores da terceira linha (índice 2) para que todos os sensores dessa linha marquem 25.5 graus.

c) Imprime no ecrã apenas os valores da quarta coluna (índice 3) de todas as linhas para verificar a ventilação lateral.

d) Exibe a matriz final de forma organizada (linha a linha) para que o utilizador consiga visualizar a grelha da sala.

Matrizes quadradas e retangulares

Muitas vezes pensamos em matrizes como 3×3 ou 4×4 (matriz quadrada), mas na informática elas são frequentemente retangulares.

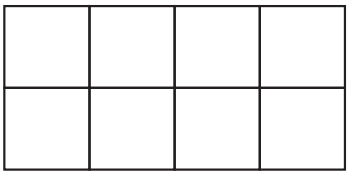
Por exemplo:

– **matriz de imagem**: uma fotografia *HD* é uma matriz de 1920 por 1080 píxeis;

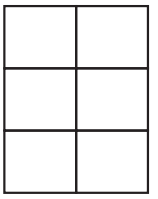
– **base de dados**: uma tabela com 1000 clientes (linhas) e cinco características (colunas) é uma matriz 1000 por cinco.

Uma **matriz** é considerada **retangular** quando o número de linhas (m) é diferente do número de colunas (n).
A matriz retangular é horizontal quando $n > m$.
A matriz retangular é vertical quanto $m > n$.
Uma matriz é **quadrada** quando $n = m$. Denomina-se matriz de ordem n.

Matriz retangular horizontal ($n > m$)
(*array* bidimensional)



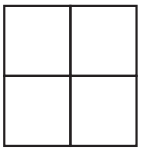
Matriz retangular vertical ($m > n$)
(*array* bidimensional)



Vetor linha ($1 \times n$)
(*array* unidimensional)

Vetor coluna ($n \times 1$)
(*array* unidimensional)

Matriz quadrada ($n = m$)
(*array* bidimensional)



☑ Not@ que:

Se consegues dizer que um elemento está na 'Linha X, Coluna Y', estás perante uma matriz. Se ele está apenas na 'Posição X', é um *array* simples (vetor).

<Modo ON #31>

Observa as duas variáveis definidas abaixo:
estrutura_1 = [[10, 20], [30, 40]]
estrutura_2 = [[10, 20, 30], [40, 50, 60]]
Analisa as dimensões destas estruturas e indica a afirmação correta.

(A) A estrutura_1 representa uma matriz retangular porque tem um total de quatro elementos.

(B) A estrutura_2 representa uma matriz quadrada, pois todas as suas sublistas têm o mesmo tamanho.

(C) A estrutura_1 é uma matriz quadrada (2 × 2), pois o número de sublistas (linhas) é igual ao número de elementos em cada sublista (colunas).

(D) A estrutura_2 é uma matriz quadrada (3 × 2) porque contém três colunas e duas linhas.

A transposta de uma matriz

Uma operação clássica é "rodar" a matriz, transformando linhas em colunas. Em ciência de dados, isto é fundamental.

Se tens uma matriz $\boldsymbol{A}$ de ordem $\boldsymbol{m} \times \boldsymbol{n}$, a sua transposta, escrita como $\boldsymbol{A^T}$ (ou $\boldsymbol{A'}$), terá a ordem inversa $\boldsymbol{n} \times \boldsymbol{m}$.

Exemplo

$\boldsymbol{A}$

10	20	30
40	50	60

$\boldsymbol{A^T}$

10	40
20	50
30	60

```
1   tabuleiro = [
2       [10, 20, 30],
3       [40, 50, 60]
4   ]
5   print(tabuleiro)
6
7   # Inverter linhas por colunas
8   transposta = [[tabuleiro[j][i] for j in range(len(tabuleiro))] for i in range(len(tabuleiro[0]))]
9   print (transposta)
```

OUTPUT

```
[[10, 20, 30], [40, 50, 60]]
[[10, 40], [20, 50], [30, 60]]
```

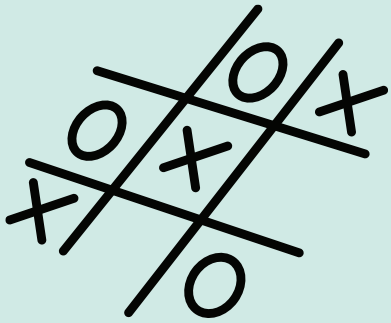
A

A^T

Esse código (em dois formatos possíveis) transforma a **matriz tabuleiro**, na qual os dados estão nas linhas, numa matriz nova (**matriz transposta**), na qual os dados passam para as colunas.

<Modo ON #32>

1. Executa no Python o código do exemplo anterior.
2. Cria uma matriz para um tabuleiro que simule o jogo do galo, utilizando os símbolos 'X' e '0'. As posições devem ser preenchidas de acordo com as coordenadas indicadas pelo jogador.



Numerical Python – a biblioteca NumPy

NumPy é uma biblioteca que adiciona suporte para **vetores e matrizes multidimensionais**, acompanhada por uma vasta coleção de funções matemáticas de alto nível.

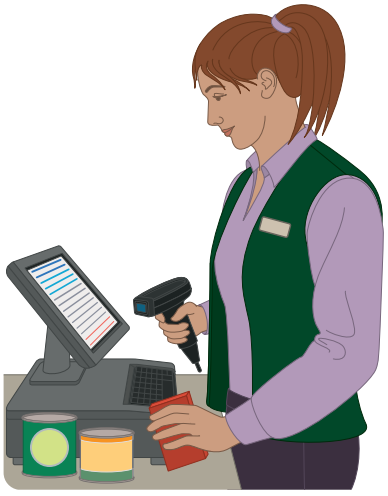
Intalação do NumPy através do Terminal

No **Visual Studio Code**, acede ao menu **Terminal**, opção **New Terminal** e digita o comando:

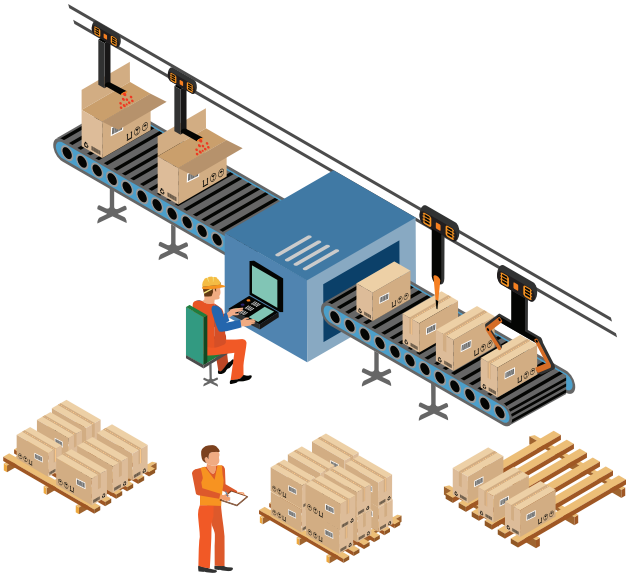
```
python -m pip install --user numpy
```

Vai surgir, no painel **Terminal**, informações do *download* e instalação do **NumPy**.

```
Collecting numpy
  Downloading numpy-2.4.2-cp312-cp312-win_amd64.whl.metadata (6.6
Downloading numpy-2.4.2-cp312-cp312-win_amd64.whl (12.3 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 12.3/12.3 MB 12.6 MB/
Installing collected packages: numpy
  WARNING: The scripts f2py.exe and numpy-config.exe are installe
\Scripts' which is not on PATH.
  Consider adding this directory to PATH or, if you prefer to sup
Successfully installed numpy-2.4.2

[notice] A new release of pip is available: 24.2 -> 26.0.1
[notice] To update, run: python.exe -m pip install --upgrade pip
```



Lista Python
É como um caixa de supermercado a passar um produto de cada vez.



NumPy
É como uma passadeira rolante automática que processa vários produtos num segundo.

Exemplo

```
1  import numpy as np
2  tabuleiro = np.array([
3      [1, 2, 3],
4      [4, 5, 6],
5      [7, 8, 9]
6  ])
7
8  print(tabuleiro)
9  print(tabuleiro.shape)
```

OUTPUT

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
(3, 3)
```

O programa começa com a importação da biblioteca através do comando **import numpy as np**, em que o apelido **np** é utilizado por convenção para tornar o código mais curto e legível.

De seguida, é criada uma variável chamada **tabuleiro**, que armazena uma matriz quadrada de ordem 3 (ou seja, três linhas e três colunas).

Esta matriz é definida a partir de uma lista de listas, em que a primeira linha contém os números 1, 2 e 3, a segunda os números 4, 5 e 6 e a terceira os números 7, 8 e 9.

Após a criação, o código utiliza a função **print(tabuleiro)** para exibir a matriz formatada no terminal, permitindo visualizar a disposição dos números em grelha.

Finalmente, a instrução **print(tabuleiro.shape)** é executada para extrair e mostrar as dimensões da matriz. Neste caso, o ***output*** será (3, 3), confirmando que a estrutura possui exatamente 3 linhas e 3 colunas, o que a caracteriza matematicamente como uma matriz quadrada.

<Modo ON #33>

1 **Cria uma matriz 3 × 3 com zeros usando NumPy.**

2 **Na matriz A = [[1, 2], [3, 4]], qual a instrução para obter o número 4?**

3 **Considera que um aluno está a desenvolver um programa para processar as notas de uma turma. Criou a seguinte matriz utilizando o NumPy:**

```
import numpy as np
notas = np.array([
    [12, 15, 18, 14],
    [10, 11, 13, 15],
    [17, 16, 14, 12]
])
```

Ao executar o comando print(notas.shape), qual será o resultado exibido no terminal e o que é que cada número representa?

a) (4, 3) – Representa 4 linhas (alunos) e 3 colunas (disciplinas).

b) (12) – Representa o total de 12 elementos presentes na matriz.

c) (3, 4) – Representa 3 linhas (horizontais) e 4 colunas (verticais).

d) (3, 4) – Representa 3 colunas e 4 linhas.

Operações com matrizes

Adicionar elementos a uma matriz

Exemplo

```
1   matriz = [[1, 2], [3, 4]]
2
3   # Adicionar uma nova linha
4   matriz.append([5, 6])
5   print(matriz)
6
7   # Adicionar uma nova coluna
8   nova_coluna = [10, 20, 30]
9   for i in range(len(matriz)):
10      matriz[i].append(nova_coluna[i])
11  print(matriz)
```

OUTPUT

```
[[1, 2], [3, 4], [5, 6]]
[[1, 2, 10], [3, 4, 20], [5, 6, 30]]
```

... utilizando o NumPy

```
1   import numpy as np
2
3   m = np.array([[1, 2], [3, 4]])
4
5   # Adicionar uma linha
6   nova_linha = np.array([[5, 6]])
7   m = np.append(m, nova_linha, axis=0)
8
9   # Adicionar uma coluna
10  nova_coluna = np.array([[10], [20], [30]])
11  m = np.append(m, nova_coluna, axis=1)
```

☑ Not@ que:

Adicionar elementos a uma matriz é como organizar uma estante de livros. Se quisermos adicionar uma "linha", estamos a colocar uma prateleira nova no fundo. Se quisermos adicionar uma "coluna", estamos a esticar a estante para o lado, acrescentando um espaço novo em cada prateleira já existente. No Python-padrão, fazemos isto linha a linha, enquanto no **NumPy** usamos o conceito de eixos (*axis*), em que o **Eixo 0** representa a vertical e o **Eixo 1** a horizontal.

<Modo ON #34>

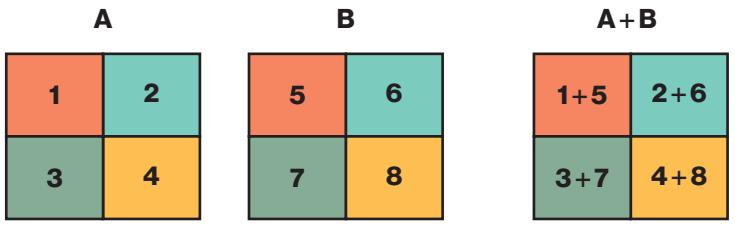
Escreve um código que peça ao utilizador quatro números e os coloque numa matriz 2 × 2.

Soma de matrizes

Para somar duas matrizes (sem usar bibliotecas externas), precisamos de seguir a regra matemática: as matrizes têm de ter o mesmo tamanho, e somamos os elementos que ocupam a mesma posição.

Atenta no exemplo de como somar a **matriz_A** com a **matriz_B** para gerar uma **matriz_soma sem utilização do NumPy**:

Exemplo




```python
matriz_A = [
    [1, 2],
    [3, 4]
]

matriz_B = [
    [5, 6],
    [7, 8]
]

# Matriz vazia com o mesmo tamanho
matriz_soma = [[0, 0], [0, 0]]

# Percorrer as linhas
for i in range(len(matriz_A)):

# Percorrer as colunas
    for j in range(len(matriz_A[0])):
        matriz_soma[i][j] = matriz_A[i][j] + matriz_B[i][j]

# Resultado
for linha in matriz_soma:
    print(linha)
```

OUTPUT

```
[6, 8]
[10, 12]
```

Soma direta com NumPy

Exemplo

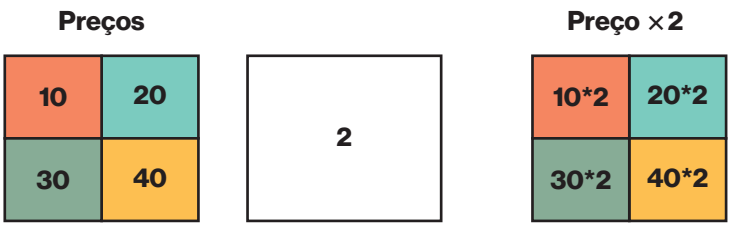
```
1  import numpy as np
2
3  m1 = np.array([[1, 2], [3, 4]])
4  m2 = np.array([[5, 6], [7, 8]])
5
6  # Soma direta!
7  resultado = m1 + m2
8
9  print(resultado)
```

OUTPUT

```
[[ 6  8]
 [10 12]]
```

Multiplicação com matrizes

Exemplo




```
1  import numpy as np
2
3  # Matriz original (ex: preços de 2 produtos em 2 lojas)
4  precos = np.array([
5      [10, 20],
6      [30, 40]
7  ])
8
9  # Multiplicar a matriz por 2 (dobrar os valores)
10 resultado = precos * 2
11
12 print(resultado)
```

O que acontece matematicamente:

$$2 \times \begin{pmatrix} 10 & 20 \\ 30 & 40 \end{pmatrix} = \begin{pmatrix} 2 \times 10 & 2 \times 20 \\ 2 \times 30 & 2 \times 40 \end{pmatrix} = \begin{pmatrix} 20 & 40 \\ 60 & 80 \end{pmatrix}$$

Característica	Listas de Python	Matrizes (*arrays*) do NumPy
Conteúdo	Podem misturar tipos (texto, números, etc.)	Tipo único (apenas números, por exemplo)
Velocidade	Lentas para cálculos grandes	Ultrarrápidas
Memória	Consumo elevado	Muito eficiente e compacta
Operações	***list* * 2** duplica a lista	***array* * 2** multiplica cada número por 2

<Modo ON #35>

Uma loja de informática tem dois armazéns (Armazém A e Armazém B). Cada armazém guarda o *stock* de ratos e teclados em duas prateleiras diferentes.
As matrizes de *stock* atual são:
Armazém A: [[10, 20], [30, 40]]
Armazém B: [[5, 5], [10, 10]]

a) Cria uma matriz chamada **stock_total** que represente a soma do *stock* dos dois armazéns.

b) A loja decidiu fazer uma promoção e encomendou material para triplicar o **stock_total**. Cria uma matriz chamada **stock_final** que seja o triplo da soma anterior.

c) Após calculares a matriz **stock_final**, indica qual é o valor que se encontra na posição [1][0] (Linha 1, Coluna 0).

Testa os teus conhecimentos

1 **Em Python "puro" (sem bibliotecas externas), qual é a forma correta de representar uma matriz de duas linhas e três colunas?**

a) matriz = [1, 2, 3, 4, 5, 6]

b) matriz = [[1, 2, 3], [4, 5, 6]]

c) matriz = [1, 2], [3, 4], [5, 6]

2 **Dada a matriz A = [[10, 20, 30], [40, 50, 60]], qual é o valor que se encontra na posição A[1][0]?**

a) 10 **b)** 20 **c)** 40 **d)** 60

3 **Se tivermos uma matriz M = np.array([[1, 1], [1, 1]]) e executarmos o comando M * 5, qual será o resultado?**

a) Uma matriz em que apenas o primeiro elemento é 5.

b) Uma matriz 2×2 em que todos os elementos são 5.

c) Uma matriz com 5 linhas e 5 colunas.

4 **Se uma matriz original tem o shape de (4, 2) (4 linhas e 2 colunas), qual será o shape da sua matriz transposta?**

a) (4, 2) **b)** (2, 4) **c)** (4, 4) **d)** (2, 2)

5 **Considera que tens uma lista de listas que representa uma matriz A:**
A = [[1, 2, 3], [4, 5, 6]]
Para criar a matriz transposta (AT), qual das seguintes descrições define corretamente a nova estrutura a obter?

a) Uma nova lista em que a ordem dos elementos de cada linha é invertida, passando a ser [[3, 2, 1], [6, 5, 4]].

b) Uma estrutura com 3 linhas e 2 colunas, em que os elementos que estavam na primeira coluna de A passam a ser a primeira linha de AT.

c) Uma estrutura com 3 linhas e 2 colunas, em que os elementos da primeira linha de A (1, 2, 3) passam a ser a primeira coluna de AT.

6 **Classifica em verdadeira (V) ou falsa (F) cada uma das afirmações seguintes.**

(A) Uma matriz é sempre um *array* de duas dimensões (linhas e colunas).

(B) A estrutura [10, 20, 30] pode ser classificada como uma matriz quadrada.

(C) Em Python "puro", representamos uma matriz através de uma lista que contém outras listas.

(D) Se uma estrutura tem 3 linhas e 5 colunas, dizemos que é um *array* bidimensional ou uma matriz retangular.

(E) Um *array* unidimensional e uma matriz são exatamente a mesma coisa.

3.4. Funções e módulos

Funções nativas e funções definidas pelo programador

No ecossistema da programação em Python, as funções são blocos de código reutilizáveis que executam tarefas específicas. Estas podem ser classificadas quanto à sua origem em duas categorias principais: as nativas da linguagem e as definidas pelo programador.

Funções internas

- Integradas no intepretador do Python.
- Permanentemente disponíveis.
- Que dispensam declaração prévia.
- Que dispensam importação de ficheiros externos.
- Que fornecem operações fundamentais de manipulação de dados e interação com o sistema.

Built-in Functions

print()
len()
input() ...

Funções criadas pelo programador

- Que revelam capacidade de expansão da linguagem.
- Que utilizam a palavra-chave **def**.
- Que possibilitam a divisão de problemas complexos em subproblemas mais simples.
- Que evitam a repetição de código.

User-defined Functions

def nome_da_função()

Analogia...

Usar funções é como uma receita de cozinha. A receita está escrita no livro (definição da função), mas o bolo só existe (resultado a obter) quando decides seguir as instruções (chamada da função).

definição

chamada da função

resultado

User-defined Functions

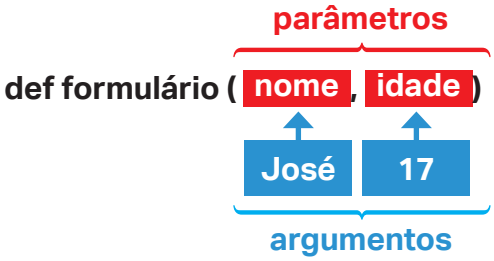
Sintaxe

```
def nome_da_funcao (parametro1, parametro2):
    # Corpo da função
    ...
    return resultado # Envia o valor de volta para quem chamou
```

– Parâmetros (opcionais): atuam como "espaços reservados" para os dados que a função espera receber.

– Argumentos: são os valores reais passados à função no momento da sua invocação (chamada à função).

– **return**: é fundamental para a modularidade, pois permite que a função exporte um valor processado para o fluxo principal do programa, encerrando a sua execução.



☑ Not@ que:

O corpo da função deve estar obrigatoriamente indentado. O fim da indentação sinaliza o fim da função.

Exemplo

```
1  def calcular_area(largura, comprimento):
2      return largura * comprimento
3
4  resultado = calcular_area(20, 50)
5
6  print(resultado)
```

Definição da função
– largura e comprimento são parâmetros;
– são variáveis locais.

Chamada à função
– 20, 50 são argumentos;
– resultado é variável global.

– O argumento 20 é dado ao parâmetro largura.

– O argumento 50 é dado ao parâmetro comprimento.

– A função **calcular_area** processa e devolve o resultado através do **return**.

☑ Not@ que:

O número de argumentos passados deve coincidir com o número de parâmetros definidos (exceto quando existem valores por omissão). A discrepância entre estes dois elementos resultará num erro de execução designado por **TypeError**.

Variável local	Variável global
• É estritamente restrita ao interior dessa função. O programa principal "não sabe" que essa variável existe. • A variável é criada quando a função é chamada e é eliminada da memória assim que a função termina a sua execução (após o **return**).	• Pode ser lida a partir de qualquer lugar do ficheiro, inclusive dentro de funções. • Permanece na memória desde o momento da sua criação até ao encerramento do programa.

<Modo ON #36>

O código seguinte faz parte de um sistema de gestão para uma alfândega que calcula o valor de desalfandegamento de mercadorias.

```
1   taxa_fixa = 500  # Linha A
2
3   def calcular_taxa_alfandega(valor_mercadoria, percentagem=0.10): # Linha B
4       custo_servico = 150 # Linha C
5       total = (valor_mercadoria * percentagem) + custo_servico + taxa_fixa
6       return total
7
8   valor_final = calcular_taxa_alfandega(5000, 0.15) # Linha D
9   print(valor_final)
```

a) Associa os elementos do código acima (linhas A, B, C, D) às respetivas definições técnicas.

1. Parâmetros: _____
2. Argumentos: _____
3. Variável de âmbito global: _____
4. Variável de âmbito local: _____

b) No momento da execução na Linha D, quais são os valores atribuídos aos parâmetros **valor_mercadoria** e **percentagem**, respetivamente?

c) Se tentarmos executar o comando **print(custo_servico)** na última linha do programa (fora da função), o que acontecerá? Justifica a tua resposta com base no conceito de ciclo de vida das variáveis.

d) Caso a chamada da função fosse alterada para **calcular_taxa_alfandega(5000)**, o programa continuaria a funcionar? Se sim, qual seria o valor utilizado para o cálculo da taxa?

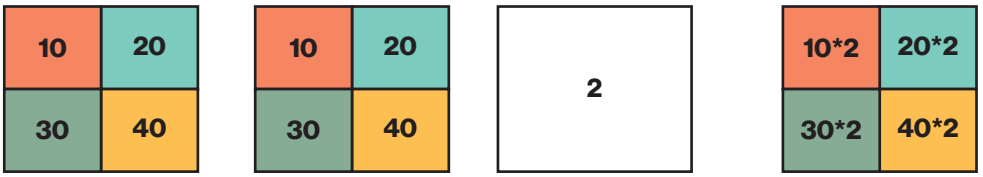
Funções aplicadas a matrizes

Exemplo

```
1   import numpy as np
2
3   def duplicar_stock(matriz):
4       resultado = matriz * 2
5       return resultado
6
7   # Usando a função
8   armazem = np.array([[10, 20], [30, 40]])
9   novo_stock = duplicar_stock(armazem)
```

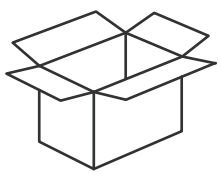
O programa inicia-se com a importação da biblioteca NumPy. De seguida, é definida uma função denominada **duplicar_stock**, que aceita um parâmetro chamado matriz. No corpo desta função, ocorre uma operação aritmética fundamental do NumPy: a multiplicação. Ao multiplicar a matriz por 2, a biblioteca

não duplica o objeto em si, mas sim cada um dos valores individuais contidos nos seus elementos, devolvendo o novo resultado através da instrução **return**.



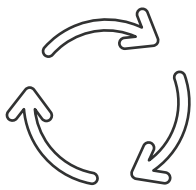
No programa principal, é criada uma variável chamada **armazem**, que armazena uma matriz criada pelo comando **np.array**. Esta matriz representa o inventário atual, contendo os valores **10**, **20**, **30** e **40**. Posteriormente, a função **duplicar_stock** é chamada, recebendo a matriz **armazém** como argumento. O valor processado e devolvido pela função é então atribuído à variável **novo_stock**.

Organização



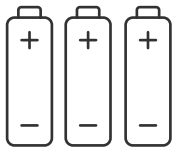
O código fica mais limpo e fácil de ler.

Reutilização



Escrever a lógica de algo uma vez e usar em dez programas diferentes.

Fácil manutenção



Se houver um erro no cálculo, só é necessário corrigir num sítio (dentro da função).

Módulos

Os **módulos** são ficheiros Python que contêm funções, variáveis ou constantes.
A utilização de **módulos** evita a repetição de código e promove boas práticas de programação.
A modularização é essencial em programas maiores e no trabalho colaborativo.

Embora ambos sirvam para reutilizar código e evitar a repetição, a **diferença**

entre função e módulo reside na escala e no local de armazenamento.

Característica	Função	Módulo
Definição	Bloco de código **def**	Ficheiro **.py** completo
Localização	Dentro de um ficheiro	Ficheiro externo ao programa
Acesso	Basta chamar o nome	Requer a instrução **import**
Analogia	Uma ferramenta (por exemplo, martelo)	Uma caixa de ferramentas inteira
		

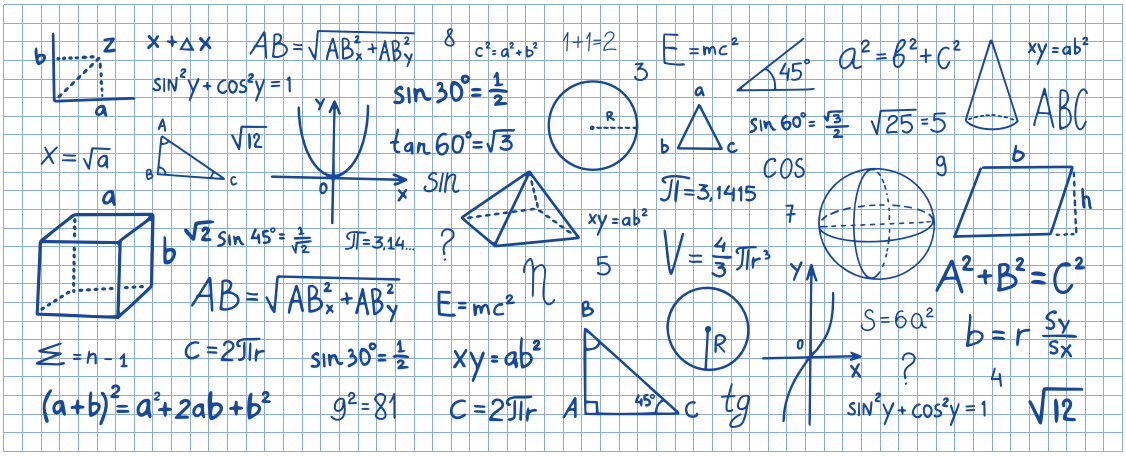
Exemplo

Imagina que estás a desenvolver o programa para a delegação de turismo:

- função: escreves uma **função saudar()** no início do programa para dizer **"Olá, bem-vindo a Cabo Verde"**;
- módulo: essa saudação agrada-te tanto que a decides guardar num ficheiro chamado **mensagens.py**. Agora, em qualquer outro programa, só será necessário fazer **import mensagens** para usar essa função.

Sabias que...

O Python já vem com uma "biblioteca" de módulos prontos a usar. No *site* **docs.python.org**, os módulos estão agrupados por categorias (texto, matemática, ficheiros, redes, etc.), facilitando a escolha da ferramenta certa para cada problema. Por exemplo, o **módulo math** facilita cálculos matemáticos.



Importação de módulos

Existem três formas principais de importar módulos em Python no código de programação:

Importação total

Importa o módulo completo; para usar as funções, precisas de usar o prefixo do nome do módulo.

import math

Importação com pseudónimo (as)

Muito útil para módulos com nomes longos.

import math as mt

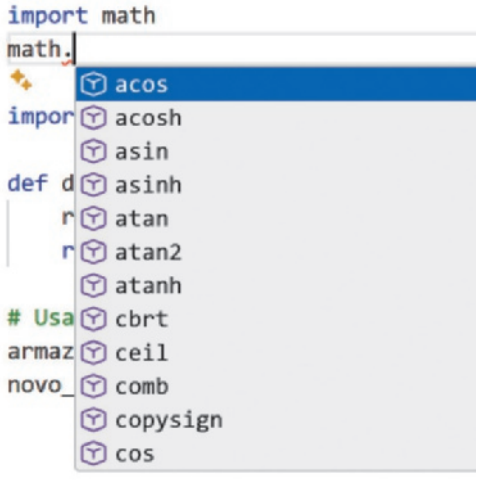
Importação específica (from)

Importa apenas uma função ou constante específica; não precisas de usar o prefixo.

from math import pi

Sabias que...

Uma das grandes vantagens do VS Code é o **IntelliSense**. Quando escreves **import math** e depois digitas **math.**, o VS Code abre automaticamente uma lista com todas as funções disponíveis nesse módulo.



Exemplo

Para prazos de projetos, o **módulo datetime** é o mais comum.

```
1   import datetime
2
3   # Usar uma função do módulo
4   hoje = datetime.date.today()
5   print(f"Hoje é dia {hoje}")
```

OUTPUT

```
Hoje é dia 2026-02-18
```

– **import datetime** permite carregar o módulo para a gestão de tempo e datas.

– **hoje = datetime.date.today()** vamos aceder à "gaveta" **datetime** e depois entramos na secção de **datas .date** e executamos a ação que é o **método .today()**. Este método comunica com o relógio interno do seu sistema operativo para obter a data atual.

Módulo math

O módulo **math** é uma das ferramentas mais úteis da biblioteca nativa do Python. Disponibiliza funções para operações matemáticas mais complexas que não podem ser realizadas apenas com os símbolos básicos (+, -, *, /).

Arredondamentos inteligentes

Muitas vezes, em pautas de notas ou gestão de orçamentos, precisamos de controlar as casas decimais de forma rigorosa:

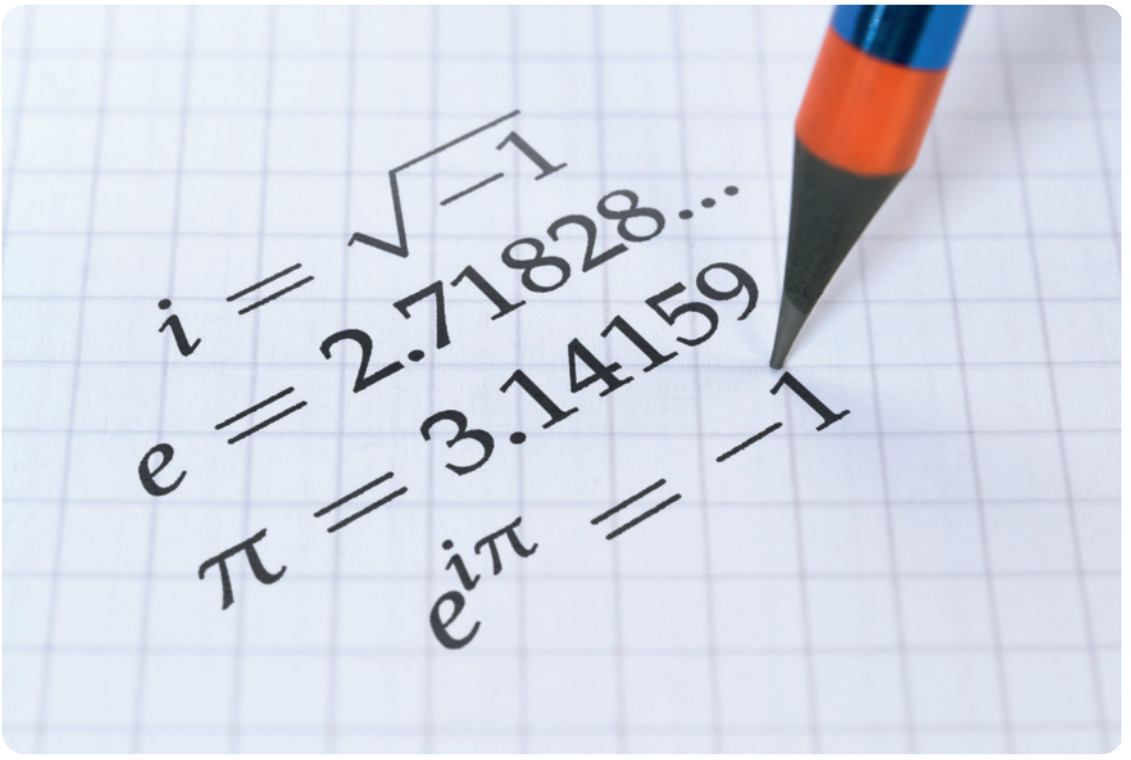
– **math.ceil(x):** arredonda sempre para cima;

– **math.floor(x):** arredonda sempre para baixo;

– **math.trunc(x):** remove a parte decimal e mantém apenas o número inteiro, sem arredondar.

Constantes matemáticas

Não precisas de decorar o valor de **Pi** ou de **e** (número de Euler).



O **módulo math** já os tem com precisão máxima:

```
1   import math
2
3   print(math.pi)  # 3.141592653589793
4   print(math.e)   # 2.718281828459045
```

Potências e raízes

Embora o Python use ****** para potências, o **math** oferece funções mais específicas:

– **math.sqrt(x):** calcula a raiz quadrada de *x*;

– **math.pow(x, y):** eleva *x* à potência de *y* (*xy*). Retorna sempre um número **decimal/float**.

Exemplo

```
1   import math
2
3   # Imagina que tem 21 passageiros e cada carrinha de transporte leva 5 pessoas.
4   carrinhas_necessarias = math.ceil(21 / 5) # 5 carrinhas
5   #(Arredonda para cima para garantir que ninguém fica sem transporte)
6
7   # Tem 50€ e cada livro custa 15.90€.
8   livros_que_posso_comprar = math.floor(50 / 15.90) # 3 livros
9   #(Arredonda para baixo, pois não há dinheiro para o 4.º livro)
10
11  # Precisa de calcular o lado de um terreno quadrado que tem 144m2.
12  lado_terreno = math.sqrt(144) # lado de 12.0 metros
```

math.cell

math.floor

math.sqrt

<Modo ON #37>

Cria dois ficheiros:

a) No ficheiro **geometria.py**, cria uma função que calcule o perímetro de um círculo (perímetro= 2 * Pi * raio). Utiliza o **math.pi** para o efeito.

b) No ficheiro **principal.py**, importa o módulo geometria e utiliza a função para calcular o perímetro de uma roda com 50 cm de raio.

Outras funções úteis

– **math.factorial(x):** calcula o fatorial de um número inteiro positivo. Multiplica um número inteiro por todos os seus antecessores até chegar a 1;

– **math.gcd(x, y):** encontra o máximo divisor comum entre dois números. Esta função encontra o maior número inteiro que consegue dividir dois números ao mesmo tempo sem deixar resto.

Exemplo

De quantas formas diferentes podes organizar cinco tarefas.

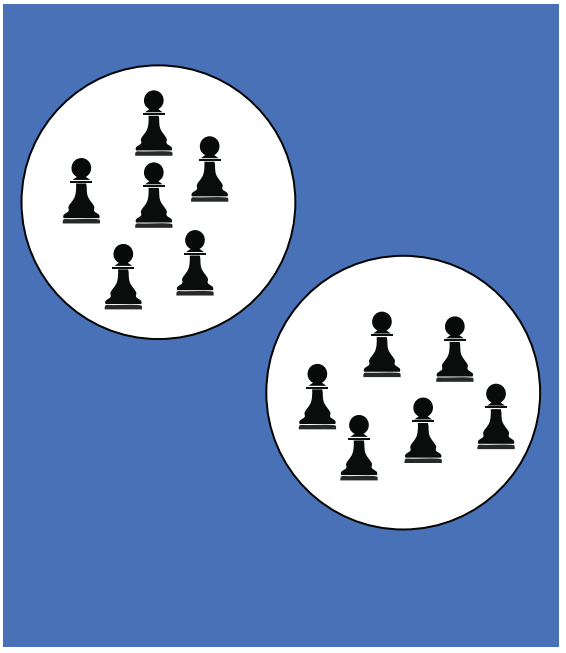
```
1   import math
2
3   organizacoes = math.factorial(5)
4   print(organizacoes) # Resultado: 120 (5 * 4 * 3 * 2 * 1)
```

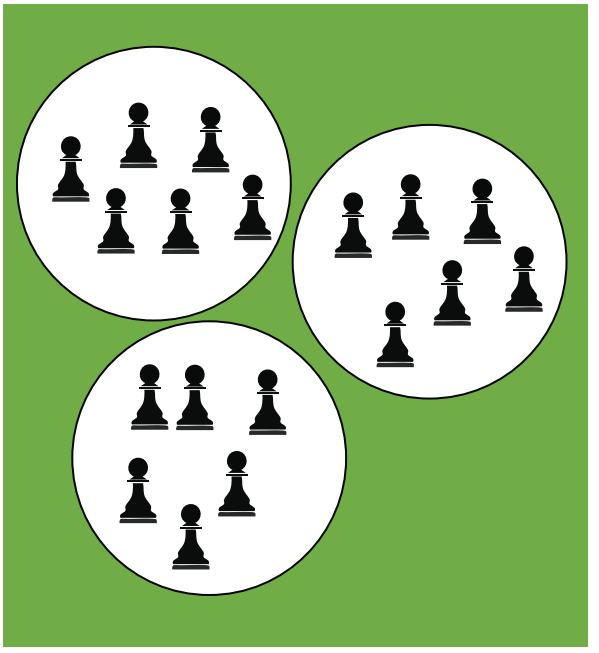
Exemplo

Imagina que tens dois grupos de amigos, um com 12 pessoas e outro com 18, e queres dividi-los em equipas com o mesmo número, sem sobrar ninguém.

```
1   import math
2
3   tamanho_equipa = math.gcd(12, 18)
4   print(tamanho_equipa) # Resultado: 6
```

Isto significa que pode fazer duas equipas de seis no primeiro grupo e três equipas de seis no segundo.





<Modo ON #38>

Abaixo tens um programa que combina as duas funções gcd e factorial num cenário real: organizar um evento para o projeto *Skills*. O teu desafio é explicar o que cada parte faz comentando cada instrução.

```
1   import math
2
3   n_alunos = 24
4   n_brindes = 36
5   n_tarefas = 4
6
7   tamanho_grupo = math.gcd(n_alunos, n_brindes)
8   combinacoes_tarefas = math.factorial(n_tarefas)
9
10  print(f"Cada grupo terá {tamanho_grupo} elementos para não sobrar material.")
11  print(f"Existem {combinacoes_tarefas} formas de ordenar as tarefas do projeto.")
```

Módulo random (números aleatórios)

O módulo **random** é a ferramenta nativa do Python para lidar com o acaso. É essencial quando precisas de imprevisibilidade no teu programa, seja para sortear um amigo para uma festa, baralhar uma lista de cartas ou gerar números aleatórios.

Gerar números aleatórios

Existem três formas principais de pedir um número ao Python:

- **random.random()**: devolve um número decimal (**float**) entre 0.0 e 1.0;
- **random.randint(a, b)**: devolve um número inteiro entre a e b (inclusive). É o mais usado para simular dados ou sorteios;
- **random.uniform(a, b)**: devolve um número decimal entre a e b. Útil para valores como preços ou medidas.

Random em listas

– **random.choice(lista)**: escolhe um elemento aleatório da lista;

– **random.sample(lista, k)**: escolhe *k* elementos diferentes da lista (sem repetição). Ótimo para formar grupos;

– **random.shuffle(lista)**: baralha a lista original. Esta função altera a lista diretamente, não cria uma nova.

Exemplo

Imagina que queres automatizar algumas decisões para a tua festa de aniversário (quem inicia o jogo, pontuação e grupo de amigos).

```
1   import random
2
3   amigos = ["Ana", "Bruno", "Carlos", "Duarte", "Elena"]
4
5   # Sortear quem começa a jogar
6   jogador = random.choice(amigos)
7
8   # Gerar uma pontuação de 0 a 100
9   pontos = random.randint(0, 100)
10
11  # Criar um grupo de 2 amigos para o jogo
12  grupo_especial = random.sample(amigos, 2)
13
14  print(f"O primeiro jogador será: {jogador}")
15  print(f"A pontuação obtida foi de: {pontos}")
16  print(f"Grupo sorteado: {grupo_especial}")
```

OUTPUT

```
O primeiro jogador será: Carlos
A pontuação obtida foi de: 69
Grupo sorteado: ['Carlos', 'Bruno']
```

<Modo ON #39>

Precisas de criar um programa que simule o lançamento de diferentes tipos de dados. Escreve um programa em Python que:

a) pergunte ao utilizador se ele quer lançar um dado de seis faces (comum) ou um dado de 20 faces.;

b) gira um número aleatório correspondente à escolha;

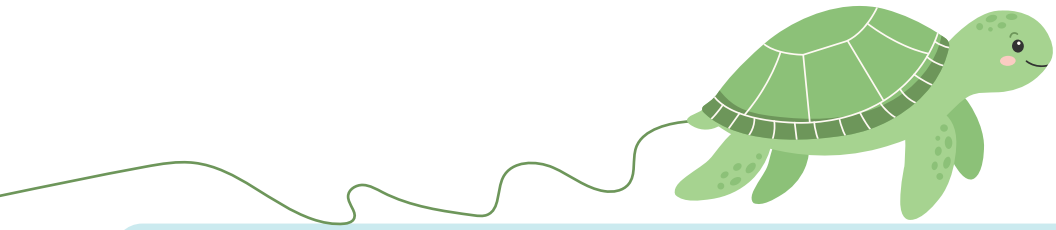
c) caso o utilizador tire o valor máximo do dado (6 ou 20), exiba uma mensagem especial: "Sucesso!";

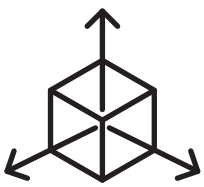
d) caso o utilizador tire o valor 1, exiba: "Estamos a iniciar...";

e) permita que o utilizador continue a lançar dados até que decida sair do programa.

Módulo turtle

O **turtle** é uma biblioteca gráfica pré-instalada no Python. Imagina uma tartaruga real num plano cartesiano e que segura uma caneta na cauda. À medida que dás ordens de movimento, ela desenha o rasto.





Estado

A tartaruga tem uma posição (x, y) e uma direção (ângulo).

Comandos de movimento

forward()
→ para a frente
backward()
→ para trás
right()
→ para a direita
left()
→ para a esquerda

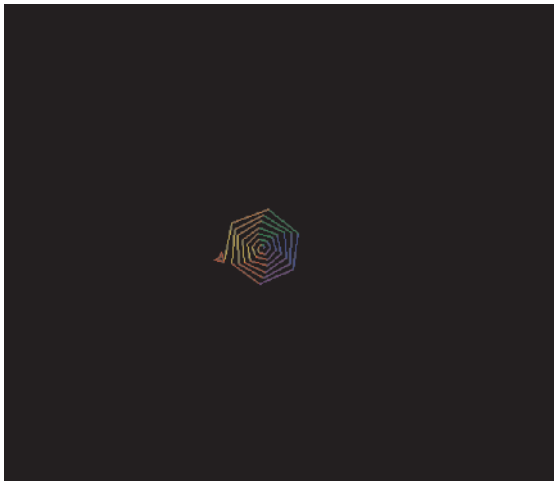
Controlo da caneta

penup()
→ levanta
pendown()
→ baixa
pencolor()
→ muda a cor

Exemplo

```
1   import turtle                          # importa o módulo
2
3   def desenho_colorido():                # definição da função
4       janela = turtle.Screen()           # janela gráfica
5       janela.bgcolor("black")            # fundo preto da janela
6       t = turtle.Turtle()                # crie a tartaruga t
7       t.speed(0)                         # velocidade máxima
8
9       cores = ["red", "purple", "blue", "green", "orange", "yellow"] # paleta de cores
10
11      for x in range(200):               # 200 passos para caminhar
12          t.pencolor(cores[x % 6])       # defina a cor da caneta
13          t.width(x/100 + 1)             # aumenta a espessura do traço
14          t.forward(x)                   # aumenta o comprimento, avança
15          t.left(59)                     # ângulo que gera o padrão
16
17      janela.exitonclick()               # janela fecha com click
18
19  desenho_colorido()                     # chamada à função
```

OUTPUT



☑ Not@ que:

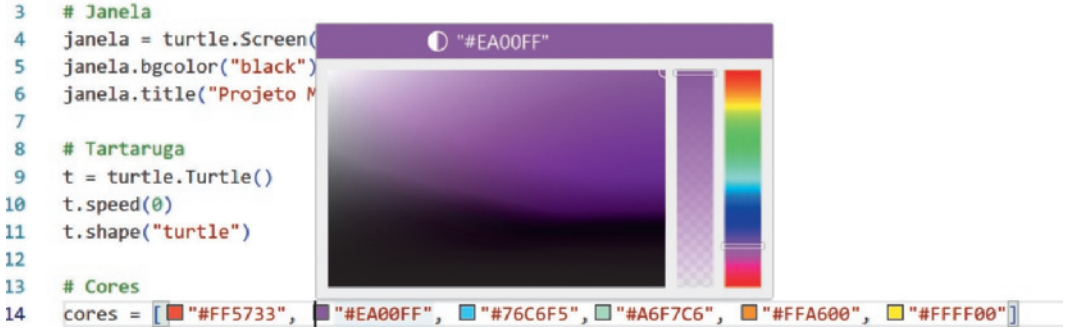
Ao contrário do que se possa pensar, o 0 em t.speed(0) não significa "parado". No módulo **turtle**, as velocidades funcionam numa escala de 1 a 10:
– 1: o mais lento (bom para depurar erros e ver o movimento);
– 6: velocidade normal;
– 10: rápido;
– 0: a velocidade máxima. Desativa a animação do movimento e "salta" logo para o final do traço.

Analisando a instrução **t.pencolor(cores[x % 6])**:

Valor de x	Cálculo (x % 6)	Resultado (Índice)	Cor selecionada
0	0 \div 6 = 0, sobra 0	cores[0]	"red"
1	1 \div 6 = 0, sobra 1	cores[1]	"purple"
5	5 \div 6 = 0, sobra 5	cores[5]	"yellow"
6	6 \div 6 = 1, sobra 0	cores[0]	"red" (Recomeça!)
7	7 \div 6 = 1, sobra 1	cores[1]	"purple"

Sabias que...

A maioria dos editores (como o *VS Code* ou PyCharm) reconhece o formato hexadecimal das cores e coloca um pequeno quadrado de cor ao lado do código, o que torna a programação muito mais intuitiva e visual. Ao clicar no quadrado, terás acesso à paleta das cores.



<Modo ON #40>

Cria um programa que desenhe uma estrutura geométrica complexa de quatro lados seguindo as regras abaixo:

a) Define uma lista com, no mínimo, cinco cores à tua escolha.

b) Cria um ciclo que se repita 200 vezes.

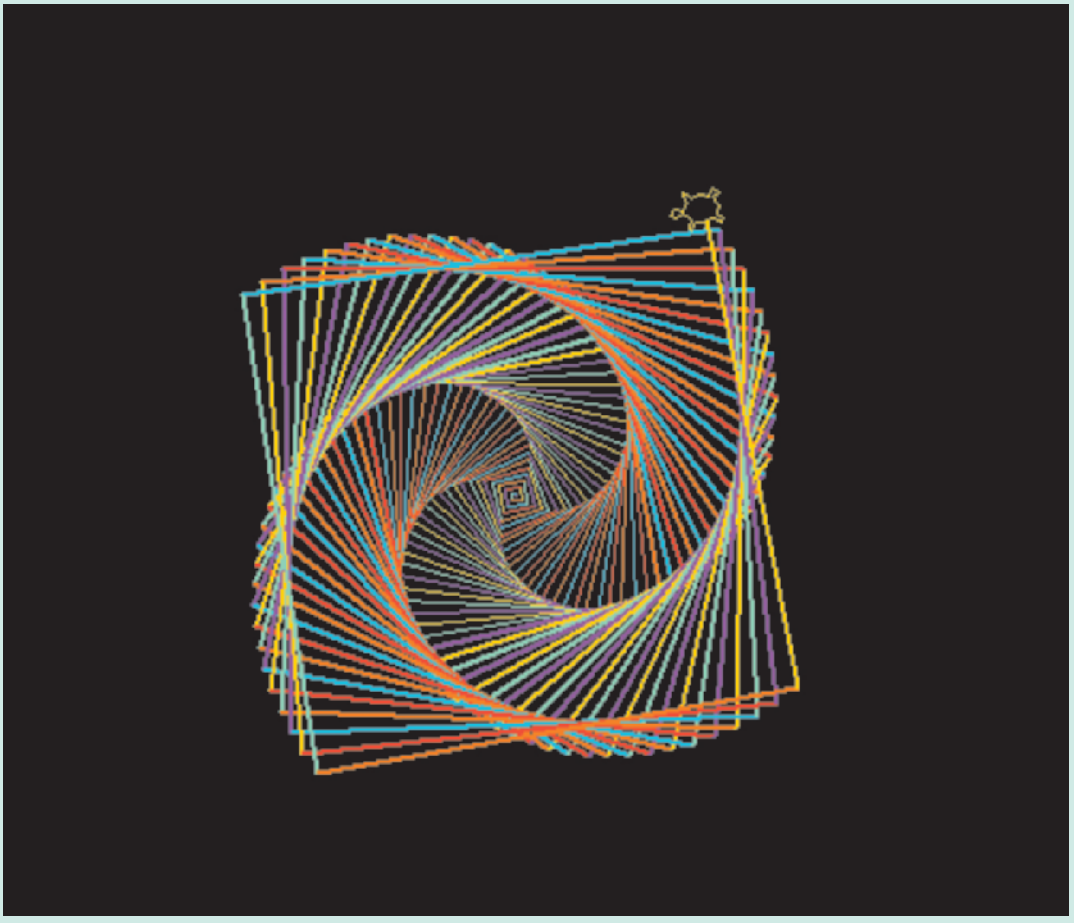
c) A cada passo, a tartaruga deve avançar uma distância baseada na variável do ciclo (por exemplo, x * 1.5).

d) A tartaruga deve rodar um ângulo fixo (sugestão: entre 59° e 91° para efeitos simétricos ou assimétricos).

e) A espessura da linha deve aumentar gradualmente à medida que o desenho se expande.

f) O desenho deve ser centrado e a janela só deve fechar quando o utilizador clicar nela.

OUTPUT PRETENDIDO



Testa os teus conhecimentos

1. Testa o seguinte código no Python e comenta cada instrução.

```
def saudar_aluno(nome):
    return f"Olá, {nome}! Bem-vindo ao manual de matrizes."

print(saudar_aluno("Helena"))
```

2. Analisa o comportamento do interpretador Python no seguinte cenário, no que concerne ao âmbito local ou global das variáveis.

```
ilha = "Sal"

def detalhar_viagem():
    hospede = "António"
    print(f"O Sr. {hospede} está na ilha do {ilha}.")

detalhar_viagem()

# Teste de visibilidade:
print(ilha)
print(hospede)
```

3. Cria uma função chamada **analisar_matriz** que receba uma matriz e imprima o seu **shape** e o seu valor máximo.

4. Qual é a principal diferença entre o resultado da função **math.sqrt(16)** e o valor 4 escrito manualmente?

5. Se tiveres um valor x = 4.2, qual é a diferença de **output** entre **math.ceil(x)** e **math.floor(x)**? Explica o conceito matemático subjacente a cada um.

6. Por que razão devemos usar **math.pi** em vez de digitar 3.14 nos nossos cálculos?

7. No comando **random.randint(1, 10)**, os números 1 e 10 são incluídos no sorteio? Como é que isto difere da função **range(1, 10)**?

8. Tens uma lista com 50 nomes de amigos. Qual é a diferença prática entre usar **random.choice(lista)** e **random.shuffle(lista)**?

9. Qual é a função pedagógica do comando **janela.exitonclick()**?


4

Projeto integrador

4.1. Integração dos conteúdos estudados

4 Projeto integrador

No final deste capítulo, deverás ser capaz de:

- Planear, desenvolver e implementar um projeto completo utilizando os conceitos aprendidos.
- Documentar as soluções e apresentar o projeto de forma clara e estruturada.
- Desenvolver autonomia e criatividade.
- Trabalhar colaborativamente.

4.1. Integração dos conteúdos estudados

Revisão rápida e panorâmica

Ao longo deste ano letivo, percorremos um caminho denso e enriquecedor na disciplina de Programação. Explorámos desde a lógica estruturada até ao desenvolvimento de soluções mais complexas, consolidando competências que agora servem de alicerce ao nosso **Projeto integrador**.

Este projeto não é apenas uma avaliação das competências adquiridas, mas a síntese prática de todas as temáticas abordadas, valorizando as seguintes dimensões: rigor da sintaxe, eficiência dos algoritmos usados e capacidade de resolução de problemas.

É o momento de cruzar o domínio das estruturas de dados com a criatividade na arquitetura de *software*, garantindo que cada linha de código contribui para uma solução robusta, funcional e profissional.

Mapa de conteúdos abordados

Processamento de dados	`soma = a + b  # processamento`
Entrada de dados	`nome = input("Como te chamas? ")`
Saída de dados	`print("Resultado:", soma)  # saída de dados`
Variáveis do tipo de dados simples	**int** **float** **str** **bool**
Operadores aritméticos, lógicos, relacionais e de atribuição	**+ - * / // % **** **AND OR NOT** **> < >= <= != ==** **= += -= *= /= //= %= **=**
Conversão após **input**	`idade=int(input())`
Formatação de saída	`print(f"O/a aluno/a {nome} tem {idade} anos.")`
Estruturas condicionais	**if** **else** **elif** **match/case**
Estruturas de repetição	**for** **while**
Controlos de fluxo	**break** **continue**
Variáveis do tipo de dados compostos	**list** **tuple** **set** **dict**
Matrizes	**Retangulares** **Quadradas** **Dinâmicas** **Transposta** **Operações com matrizes**
Funções nativas	**print ( )** **len ( )**
Funções definidas pelo programador	**def nome_da_função( )**
Módulos	**math** **random** **turtle**

Planear, desenvolver e implementar um projeto completo

Programar não é apenas escrever código, mas seguir etapas. A programação (o código em si) é apenas uma parte de um processo muito maior.

O **ciclo de vida de desenvolvimento de *software*** é o "mapa" que os profissionais seguem para transformar uma ideia abstrata num programa funcional, estável e fácil de manter.

1. Criação da ideia (*Idea generation*)
É o ponto de partida. Aqui, define-se o problema que o *software* vai resolver.

2. Análise de requisitos (*Requirement analysis*)
Nesta fase, define-se o que o programa deve fazer exatamente.

3. *Design* do produto (*Product design*)
Antes de tocar no teclado, planeia-se a arquitetura.

4. Desenvolvimento do produto (*Product development*)
Esta é a fase da codificação.

5. Integração e testes (*Integration & testing*)
O código é testado para encontrar erros (*bugs*).

6. Implementação (*Deployment*)
É o momento de colocar o programa a funcionar no "mundo real".

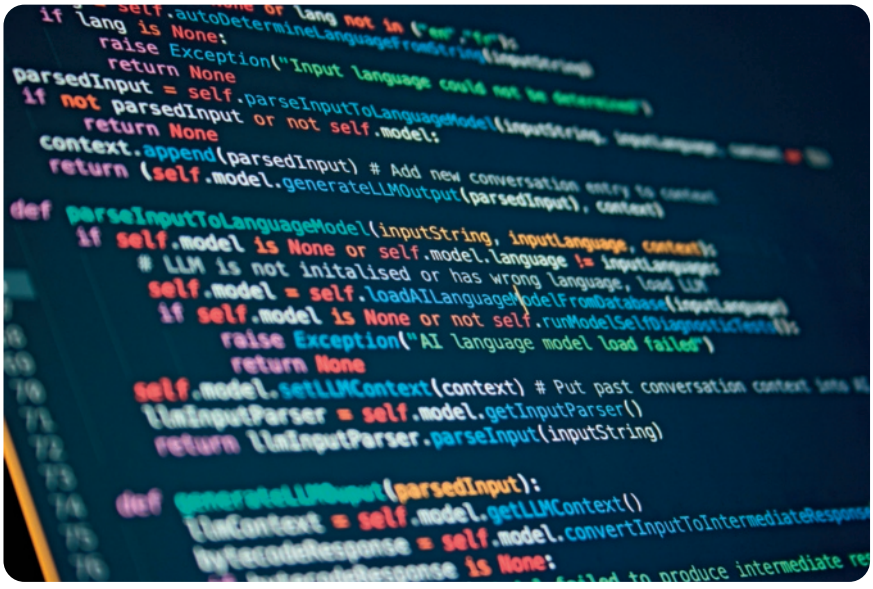
7. Manutenção (*Maintenance*)
Envolve corrigir erros de código ou adicionar novas funcionalidades sugeridas pelos utilizadores.

Documentar e apresentar soluções

A documentação é muitas vezes negligenciada pelos programadores, mas é crucial para o sucesso de um produto.

Sabias que...

O código é lido mais vezes do que é escrito.



As **Docstrings – *Documentation Strings*** funcionam como o manual de instruções embutido numa função ou classe. Em Python, utiliza-se a convenção das triplas aspas """ """.

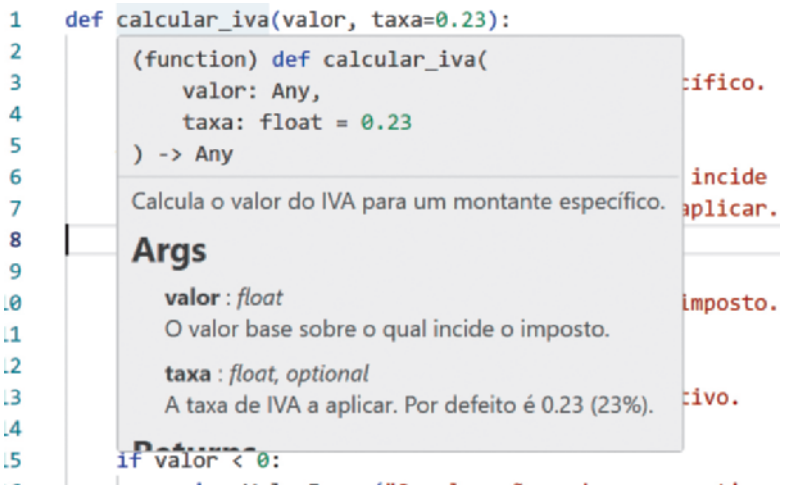
Exemplo

```python
def calcular_iva(valor, taxa=0.23):
    """
    Calcula o valor do IVA para um montante específico.

    Args:
        valor (float): O valor base sobre o qual incide o imposto.
        taxa (float, optional): A taxa de IVA a aplicar. Por defeito é 0.23 (23%).

    Returns:
        float: O valor correspondente apenas ao imposto.

    Raises:
        ValueError: Se o valor inserido for negativo.
    """
    if valor < 0:
        raise ValueError("O valor não pode ser negativo.")
    return valor * taxa
```

Este exemplo foca-se numa função utilitária básica de cálculo do IVA. Repara como descrevemos na **docstring** o que entra, o que sai e o que pode correr mal.

As **docstrings** permitem que alguém utilize o código sem ter de ler a implementação lógica. Basta saber o que a função espera receber e o que ela promete entregar.

Em ambientes de desenvolvimento (como o VS Code ou PyCharm), ao passar o rato sobre uma função externa, a **docstring** aparece num *pop-up*. Sem ela, o programador teria de abrir o ficheiro de origem para perceber como a função funciona.



Sabias que...

É possível a criação de documentação externa de forma automática através de ferramentas como o **Sphinx** ou **Doxygen**.

Trabalhar colaborativamente

O Python é a linguagem perfeita para introduzir ferramentas de colaboração que se usam na indústria. Após o registo em plataformas partilhadas, os programadores podem colaborar em projetos privados e/ou *open source* de qualquer origem e assim criar um repositório de programas.

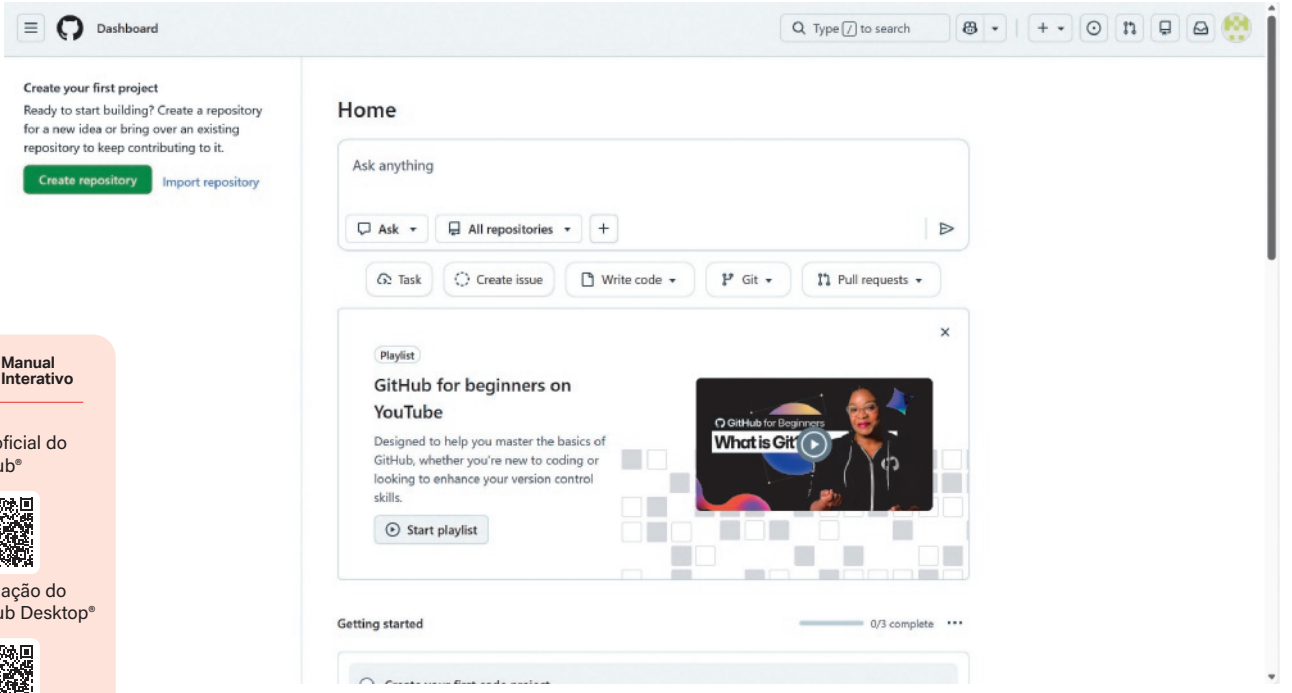
O **Git** é um sistema de controlo de versões que regista o histórico de todas as alterações no código feitas ao longo do trabalho colaborativo. É a linha de tempo do projeto: quem fez/o que fez. Possibilita voltar para uma versão anterior do projeto.
O **GitHub** é a plataforma na nuvem que armazena os repositórios originados no **Git**.

O **Git** funciona através de repositórios, pastas que armazenam todas as versões do código de programação. Cada mudança no código é registada com um **COMMIT**, o que adiciona um novo ponto na linha do tempo das versões.

Sabias que...

Para criar conta no **GitHub**, deves aceder ao *site* oficial da plataforma através do *link* https://github.com/

Como trabalhar em equipa sem "atropelar" ninguém no GitHub?

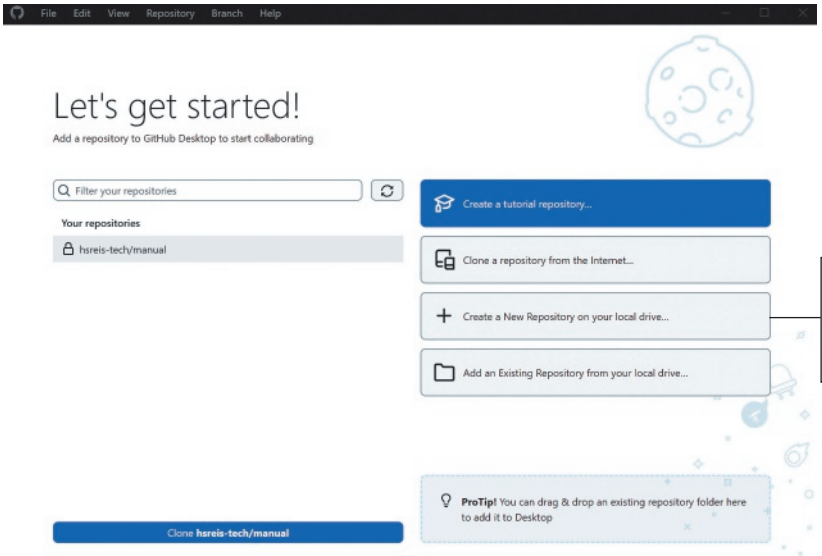


Manual Interativo

Link
Site oficial do GitHub®

Instalação do GitHub Desktop®

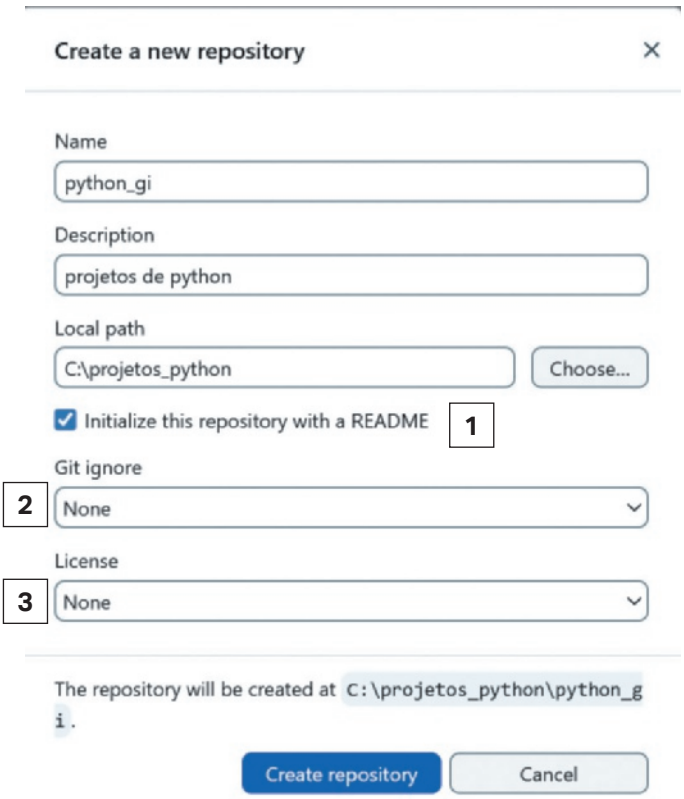
Instala o **GitHub Desktop** (disponível em https://desktop.github.com/download/) no computador em que estás a trabalhar e sincroniza com a tua conta **GitHub** *online*. Essa aplicação **Desktop** irá facilitar o desenvolvimento do código. Vamos ao passo a passo!



No **GitHub Desktop**, cria um novo repositório **python_git**

Preenche os campos da janela **Create a new repository** atribuindo um nome ao repositório e criando uma pasta local para serem guardados os ficheiros.

A aplicação oferece a possibilidade de configurar automaticamente três ficheiros fundamentais para a organização e legalidade do código: **README**, **Ignore**, **License**.



1 **README** – É o "rosto" do projeto. Funciona como um manual de introdução que explica o propósito do *software*, como instalá-lo e como utilizá-lo.

2 **Ignore** – Este ficheiro é essencial para manter o repositório limpo. Define uma lista de regras que dizem ao **Git** quais os ficheiros ou pastas que não devem ser enviados para a nuvem (como ficheiros temporários, pastas de bibliotecas pesadas ou ficheiros de configuração pessoal que contenham *passwords*). **Escolhe Python na lista de opções**.

3 **License** – Define os termos legais de utilização do teu código. Indica a terceiros o que podem ou não fazer com o *software* (por exemplo, se podem usá-lo para fins comerciais ou se são obrigados a dar crédito ao autor original).

É necessário publicar o repositório para que fique visível na nuvem do **GitHub** (na plataforma *online*).

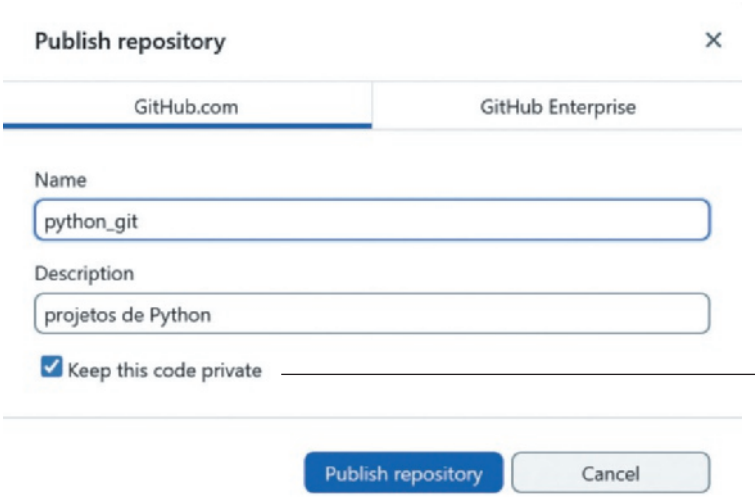
Quando crias um repositório no **GitHub**, a primeira grande decisão que tens de tomar é entre um repositório **público** ou **privado**.

Onde somente tu e os colaboradores do projeto poderão ver os códigos-fonte e alterá-los.

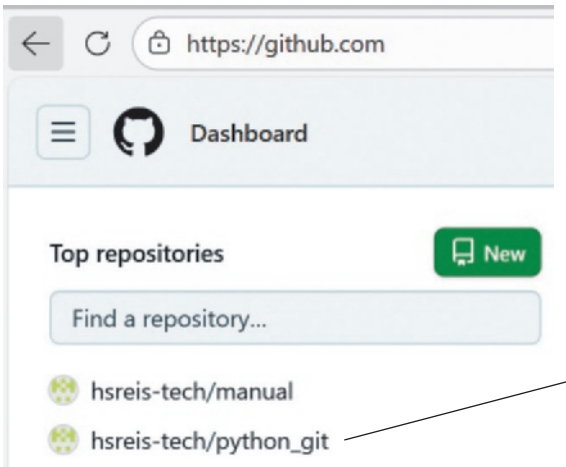
Repositório privado

Onde qualquer pessoa poderá ver e alterar o código-fonte.

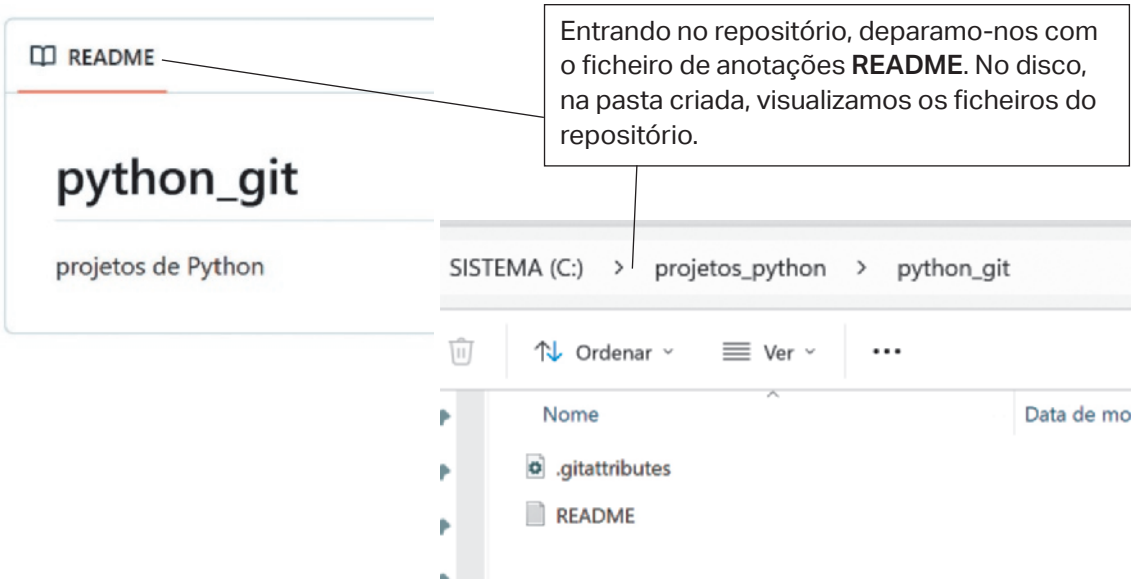
Repositório público



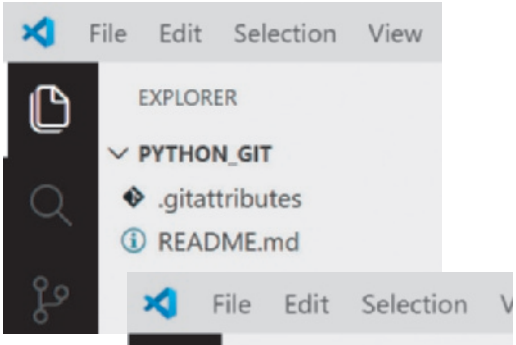
Permite definir se o repositório é **público** ☐ ou **privado** ☑.

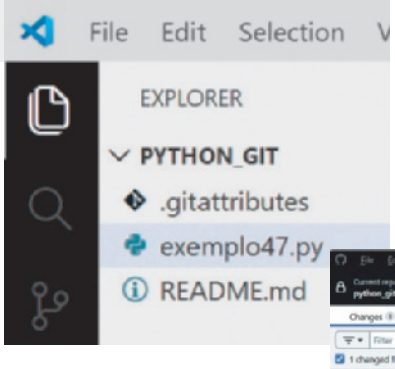


O repositório **python_git** criado em **Desktop** fica visível na plataforma *online* do **GitHub**.



Entrando no repositório, deparamo-nos com o ficheiro de anotações **README**. No disco, na pasta criada, visualizamos os ficheiros do repositório.

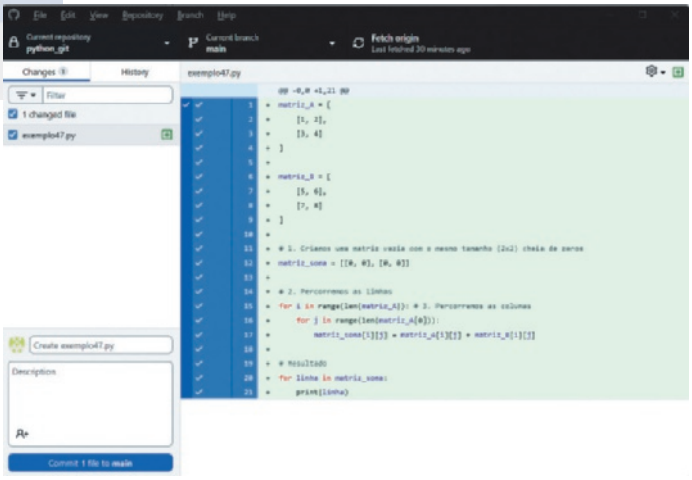




No interpretador **Visual Studio Code**, onde programamos em Python, abre a pasta do repositório **PYTHON_GIT** que está em disco.

File > Open Folder

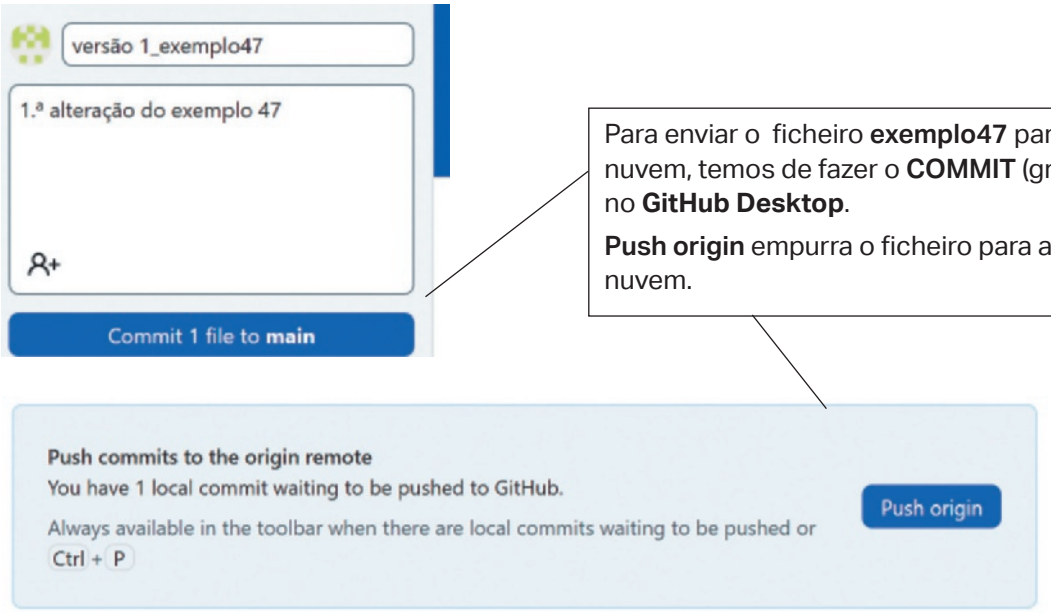
Cria um ficheiro Python (**exemplo47**).
No **GitHub Desktop**, irá surgir o ficheiro **exemplo47.py**



O exemplo foi criado em **main**, o que significa que está na primeira ramificação da árvore.

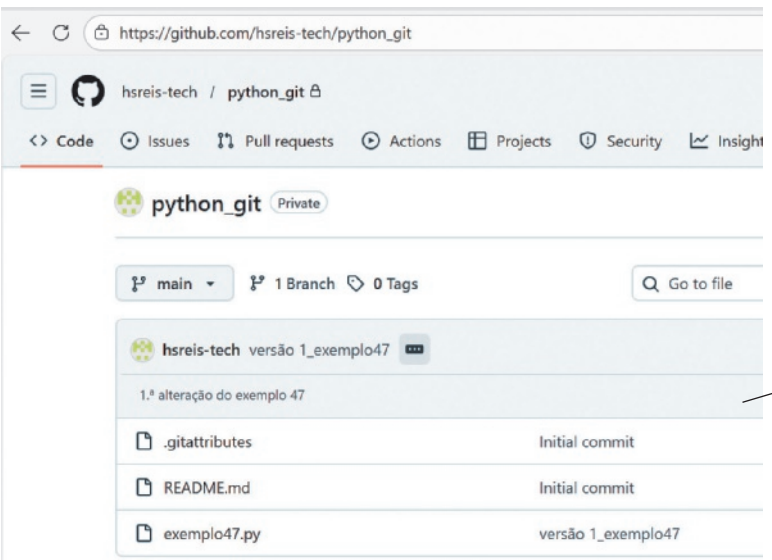
Sabias que...

Por predefinição, quando um repositório é criado, possui apenas um ramo principal, normalmente designado por **main**.
Contudo, a flexibilidade do **GitHub** permite a criação de tantos ramos (*branches*) quantos forem necessários.



Para enviar o ficheiro **exemplo47** para a nuvem, temos de fazer o **COMMIT** (gravar) no **GitHub Desktop**.

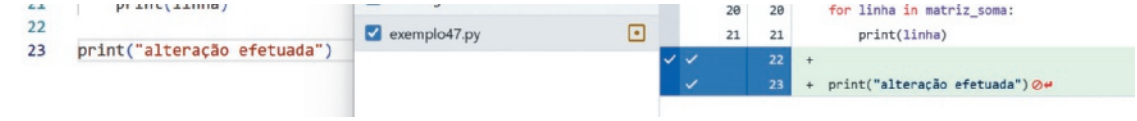
Push origin empurra o ficheiro para a nuvem.



O **exemplo47** já se encontra na nuvem.

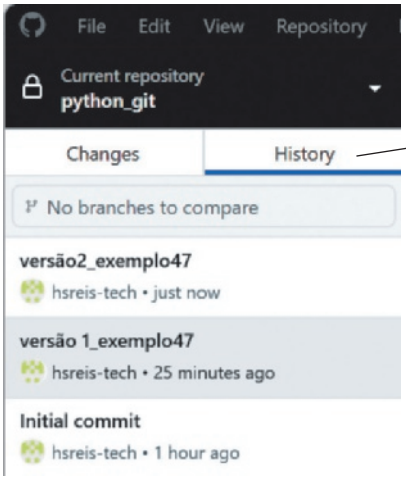
Na versão **Desktop**, o ficheiro **exemplo47** já não está listado.

As alterações no código podem ser efetuadas no **Visual Studio Code** sem o **GitHub Desktop** estar aberto.



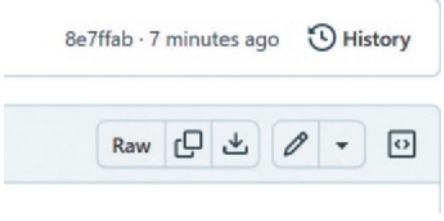
Qualquer alteração ao código de programação efetuada no **Visual Studio Code** é refletida no **GitHub Desktop**, mesmo estando este fechado.
A alteração é assinalada a fundo verde.

A seta indica que falta fazer **COMMIT** para enviar para a nuvem.



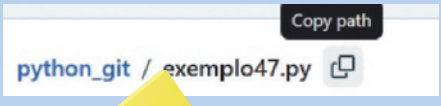
Apenas a última versão do código fica guardada na nuvem.

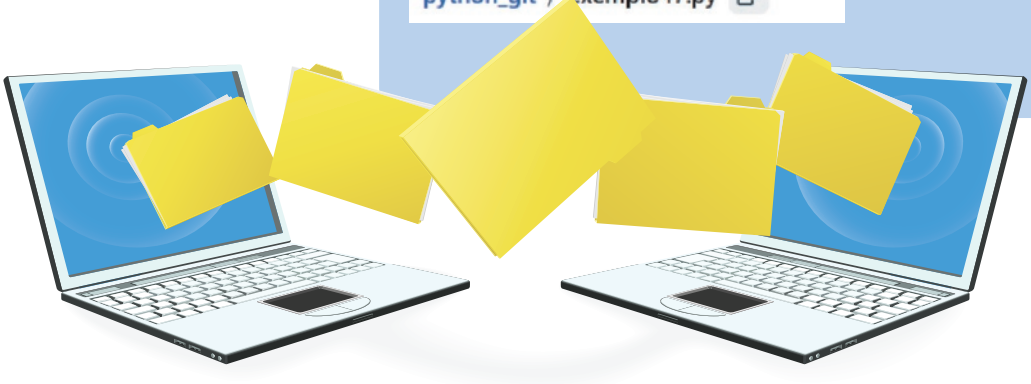
A visualização das diversas versões é possível no **GitHub Desktop** e na nuvem em **History**.



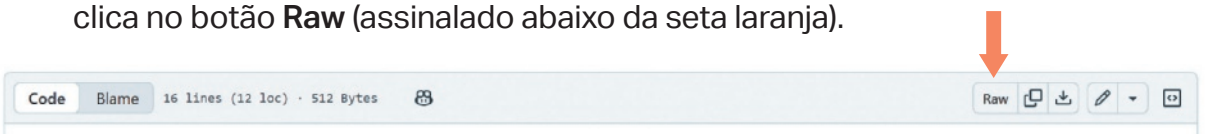
Para compartilhar um repositório, basta fornecer ao interessado o seu URL.

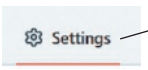
Na nuvem, o URL pode ser obtido clicando em **Copy path**.



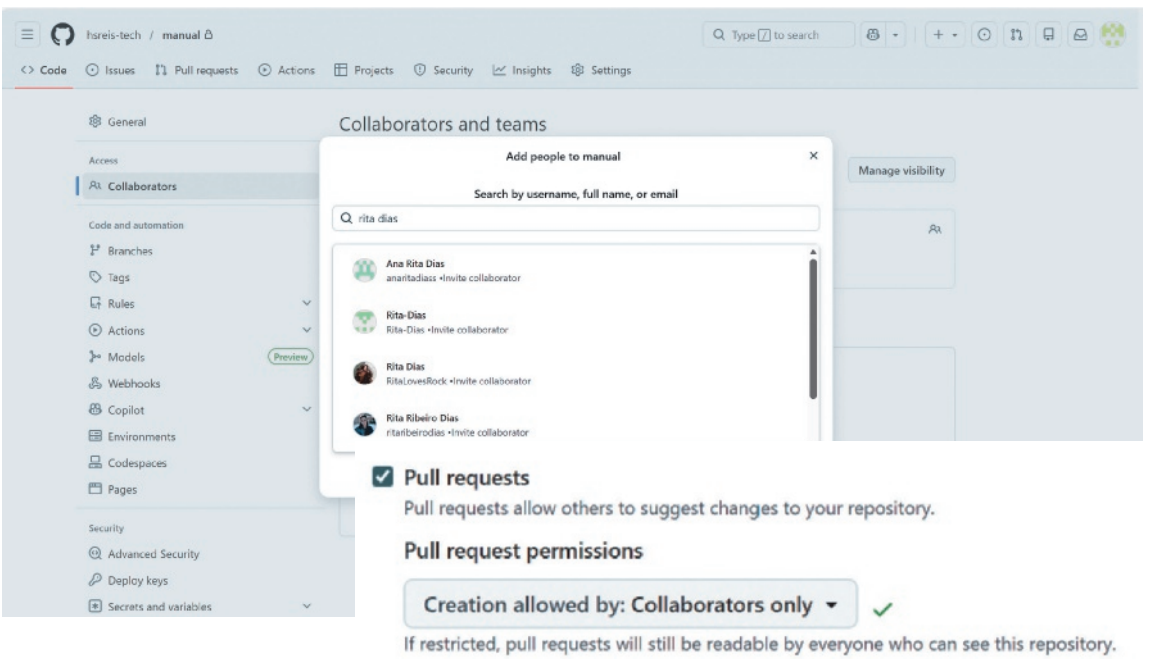


Se quiseres partilhar apenas o texto do código (útil para *scripts* de instalação), clica no botão **Raw** (assinalado abaixo da seta laranja).

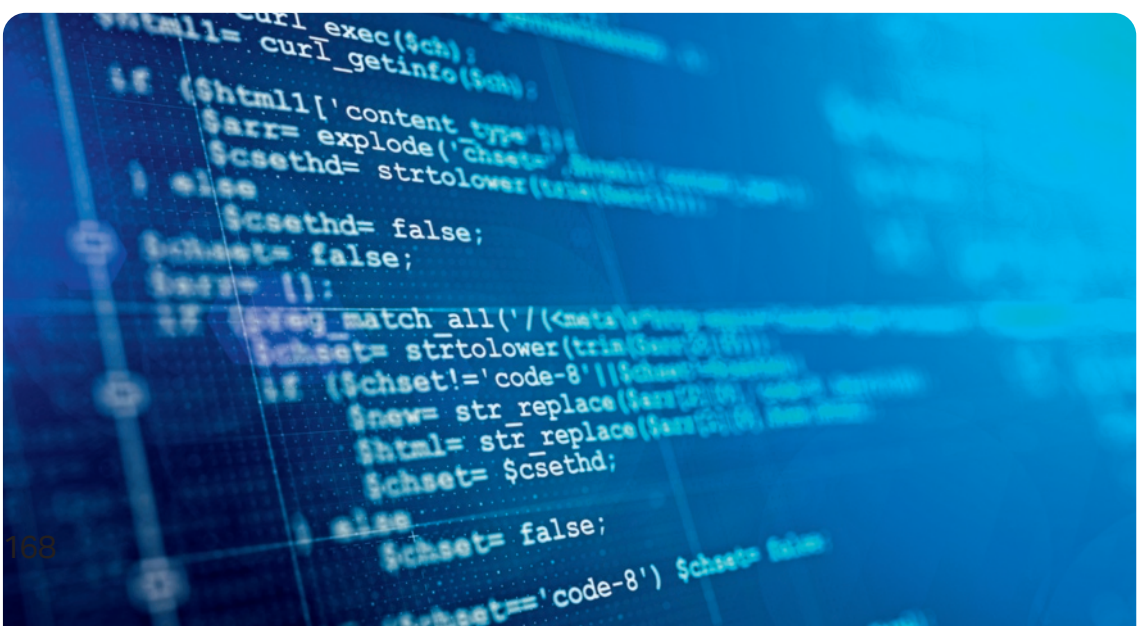




Outra possibilidade de partilha será convidares utilizador a utilizador como colaboradores. Nas **configurações (Settings)** do repositório *online*, no painel lateral escolhe **Colaboradores** e adiciona o *email* da pessoa. Poderás visualizar, na tua conta **GitHub** *online*, as atualizações de cada programador.



Vamos ao projeto final!



Testa os teus conhecimentos

O objetivo deste projeto é aplicar os conhecimentos adquiridos em Python para resolver um problema real ou criar uma experiência interativa. O trabalho deve ser desenvolvido em equipa (trabalho de pares) e gerido através das plataformas colaborativas. Dos temas disponíveis, devem escolher apenas um.

Tema A: Sistema de gestão de biblioteca

Desenvolvimento de uma plataforma para registar livros, gerir empréstimos e devoluções e consultar o catálogo atualizado.

Tema B: Jogo de Trivial (*Quiz*)

Criação de um jogo de perguntas e respostas de múltipla escolha, com sistema de pontuação e *feedback* imediato para o utilizador.

Tema C: Aplicação de controlo financeiro

Um sistema para registo de fluxos de caixa (receitas e despesas), permitindo visualizar o saldo atual e gerar um relatório resumido.

Testa os teus conhecimentos

Manual Interativo

Link
Site oficial do MindMeister®

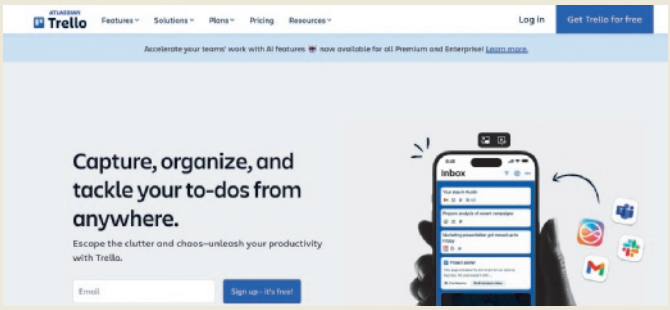
Site oficial do Trello®

Fase 1: Análise e requisitos (o que o programa deve fazer?)

Nesta fase inicial, a equipa deve definir o âmbito do projeto.

a) Desenvolvam um mapa mental que interligue as ideias principais do tema selecionado. Este processo deve consolidar o vosso *brainstorming* inicial. Para a criação deste diagrama, sugerimos a utilização de plataformas *online* colaborativas como o **MindMeister** (https://www.mindmeister.com/) ou a app **Mindly**. Partilhem o vosso mapa mental com o professor.

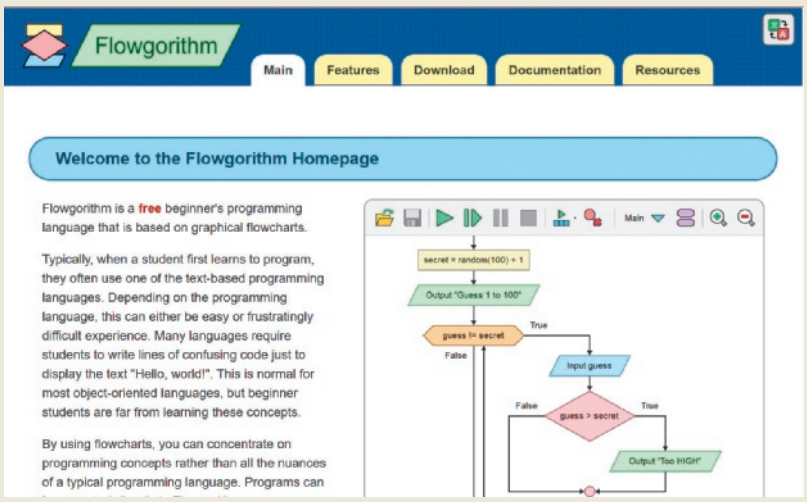
b) Listem detalhadamente todas as **ações que o programa deve executar**. Devem incluir os requisitos técnicos do programa (por exemplo, "o sistema deve importar um módulo externo", "o catálogo da biblioteca deve conter um universo de 30 livros" ou "cada registo de despesa deve incluir data, categoria e valor"). Para a gestão e listagem destas tarefas, sugerimos a utilização de uma plataforma colaborativa como o **Trello** (trello.com) ou similar.



c) Ainda utilizando o **Trello**, ou outra plataforma similar, identifiquem as limitações e regras do programa. Devem listar as condições que o *software* deve garantir (por exemplo, "o sistema não deve permitir saldos negativos" ou "a entrada de dados nas opções está restrita a números inteiros").

Fase 2: Estrutura (como será o programa?)

d) Utilizando a plataforma **Flowgorithm** (lecionada no 10.º ano) ou um *software* equivalente, desenvolvam um fluxograma que represente a estrutura global e a lógica de funcionamento do programa.



Fase 3: Implementação e colaboração

e) Desenvolvam o código em Python, garantindo a total conformidade com o plano previamente delineado. O desenvolvimento deve observar os seguintes requisitos técnicos:

- é obrigatória a inclusão de **docstrings** em todas as funções, bem como a utilização de **comentários** pertinentes que facilitem a compreensão da lógica do programa;
- utilizem a plataforma **GitHub** (ou outro recurso similar) para o trabalho colaborativo.

Testa os teus conhecimentos

Fase 4: Testes e integração

f) Verifiquem a robustez do programa desenvolvido. As diferentes partes do código funcionam bem juntas? O programa lida com erros (por exemplo, o utilizador pode introduzir letras onde se pedem números)?

CRITÉRIOS DE AVALIAÇÃO

☑ **Qualidade do código**

Avaliação da organização lógica, da escolha de nomes de variáveis intuitivos e da modularização eficiente através de funções.

☑ **Documentação**

Verificação da presença obrigatória de **docstrings** em todas as funções e de comentários que clarifiquem instruções complexas.

☑ **Colaboração**

Avaliação da colaboração efetiva entre o par em todas as fases do projeto.

☑ **Robustez**

Ausência de erros fatais (*crashes*) durante a execução do programa.

Programação 11.º ano

Criação intelectual
Dina Tavares
Helena Reis
Rita Cadima

Design
Porto Editora

Créditos fotográficos
© Stock.Adobe.com
Depositphotos.com

Edição
2026

Este manual segue
o programa experimental
da disciplina, publicado pelo
Ministério da Educação.

Cabo Verde

Brasão

Bandeira

Hino Nacional

Cântico da Liberdade

Canta, irmão
Canta, meu irmão
Que a liberdade é hino
E o homem a certeza.

Com dignidade, enterra a semente
No pó da ilha nua;
No despenhadeiro da vida
A esperança é do tamanho do mar
Que nos abraça,
Sentinela de mares e ventos
Perseverantes
Entre estrelas e o Atlântico
Entoa o cântico da liberdade.

Canta, irmão
Canta, meu irmão
Que a liberdade é hino
E o homem a certeza!

Este livro foi aprovado pelo Ministério da Educação para utilização obrigatória nas escolas