<CodeCamp/>
<@FEUP/>

14 fevereiro 2026

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

FEUP
ESCOLAS
Programa de captação
e desenvolvimento de talento

acm
Student Chapter

ni

Porquê aprender a programar?

Atualmente a programação está presente em muitos aspetos do nosso dia a dia, mesmo sem nos aperceber, num mercado em constante crescimento e evolução:

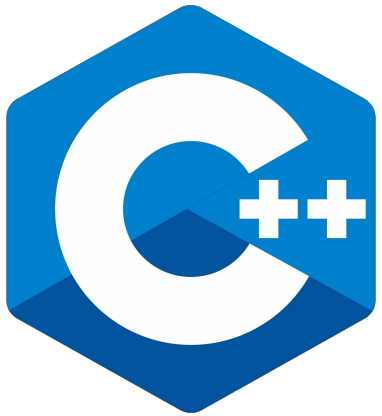
- Sistemas de Processamento de Transações (Bancos, Logísticas de Entregas, etc)
- Dispositivos Eletrónicos (smartphones, computadores)
- Análises de Dados (em áreas das Ciências e Contabilidade, por exemplo)
- ...

Para além disso, permite-nos automatizar tarefas repetitivas e desenvolve o Pensamento Computacional, uma boa capacidade para aplicar em problemas do mundo real.

Linguagens de Programação

Com o uso de linguagens de programação, nós conseguimos instruir um computador a fazer algo que nós queiramos através de um conjunto de instruções.

Claro que existem várias linguagens, cada uma com as suas vantagens. Para os objetivos de hoje, aprenderemos e usaremos Python.



Porquê Python?

Python é linguagem funcional imperativa, isto é, temos que dar instruções ao computador de **COMO** fazer algo (em contraste com linguagens declarativas onde as instruções ditam o que o computador precisa de fazer, como o faz não nos interessa).

Para além disso, por ser de alto nível e com uma sintaxe mais simples que outras linguagens, é recomendada como a linguagem introdutória para pessoas novas ao mundo da programação.

<CodeCamp/>
<@FEUP/>

14
fevereiro
2026

Variáveis e Tipos de Dados

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

FEUP
ESCOLAS
Programa de captação
e desenvolvimento de talento.

Apoio

acm
Student Chapter

ni

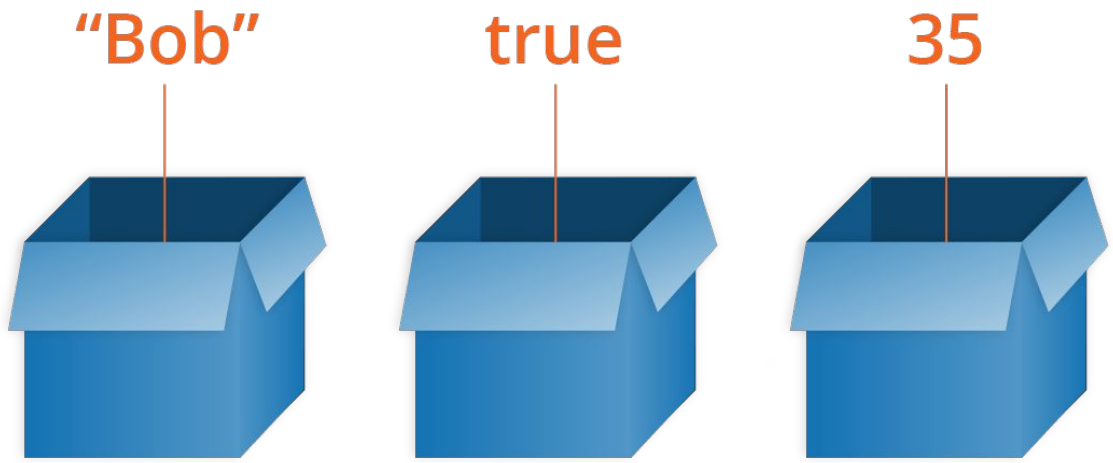
Variáveis

Em Python, uma variável é um objeto capaz de reter e representar um valor ou expressão. Elas só existem durante a execução do programa, pelo que as variáveis estão associadas a "nomes" a elas atribuídas.

Por outras palavras, variáveis são objetos que são guardados em caixas com nomes específicos para depois serem utilizados ao longo do código, sempre que forem precisos.

Para se definir variáveis em Python, usa-se o igual = :

```
hello = "World"
```



Tipos de Dados

Ao longo de um código, é preciso identificar de que tipo são os dados introduzidos, para que o Python saiba como utilizar a informação que lhe é dada.

Sendo assim existem quatro tipos de dados principais, eles são:

String (str)
são um conjunto de caracteres

(“Hello”, “Bom dia”, …)

Integer (int)
são números inteiros

(-10, 0, 8, …)

Float (float)
são números reais

(-20.8, 0.111…, 3.3, …)

Bool (bool)
são valores booleanos

(True, False)

Funções Básicas

Existem algumas funções essenciais em Python, que são usados em praticamente em qualquer código:

`print()`

Imprime texto no terminal

```
>> a = “Hello World”
>> print(a)
Hello World
```

`type()`

Devolve o tipo de dado

```
>> a = “Hello World”
>> print(type(a))
<class 'str'>
```

`len()`

Devolve o comprimento de uma string

```
>> a = “CodeCamp@FEUP”
>> print(len(a))
13
```

Funções Básicas

Existem algumas funções essenciais em Python, que são usados em praticamente em qualquer código:

input()

Recebe informação do utilizador através do input

```
>> a = input("Escreve aqui: ")
Escreve aqui: FEUP
>> print(a)
FEUP
```

round()

Arredonda um float

```
>> a = 3.14
>> print(round(a))
3
```

Operações Aritméticas

Tal como na matemática, na programação, uma operação aritmética é a execução de um cálculo matemático sobre um ou mais valores numéricos (float ou integer) utilizando símbolos definidos pela linguagem (operadores), resultando num novo valor numérico.

Numa expressão com vários operadores, os parênteses influenciam a ordem das operações.

```
>> print(2*(5+1))
12
```

OPERAÇÃO	OPERADOR	EXEMPLO	RESULTADO
Adição	+	`c = 1 + 2`	`c = 3`
Subtração	-	`c = 10 - 5`	`c = 5`
Multiplicação	*	`c = 2 * 3`	`c = 6`
Exponenciação	**	`c = 2 ** 4`	`c = 16`
Divisão	/	`c = 20 / 4`	`c = 5`
Divisão Inteira	//	`c = 23 // 3`	`c = 7`
Módulo	%	`c = 23 % 3`	`c = 2`

Operações Aritméticas

Para além destas operações comuns, o Python define umas operações especiais simples mas intuitivas em strings.

```
>> print("Code"+"Camp"+"@FEUP")
CodeCamp@FEUP
```

```
>> print(3 * "ab")
ababab
```

Condicionais

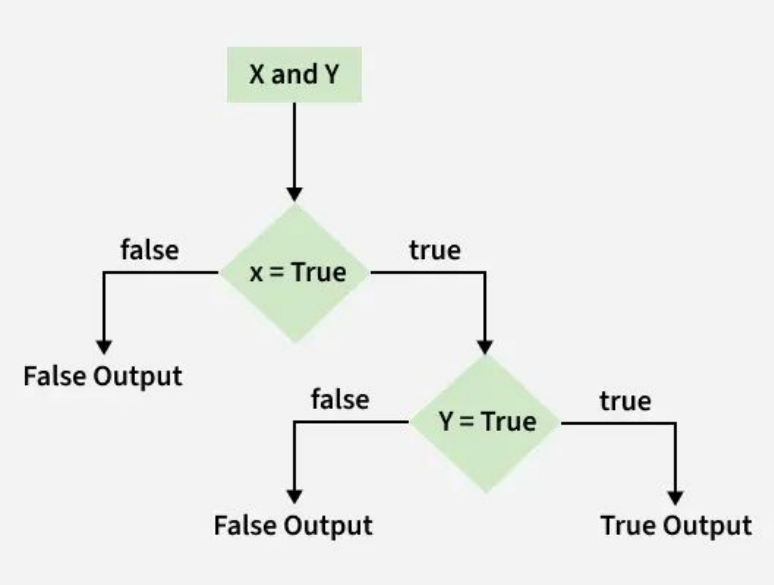
Operadores Relacionais

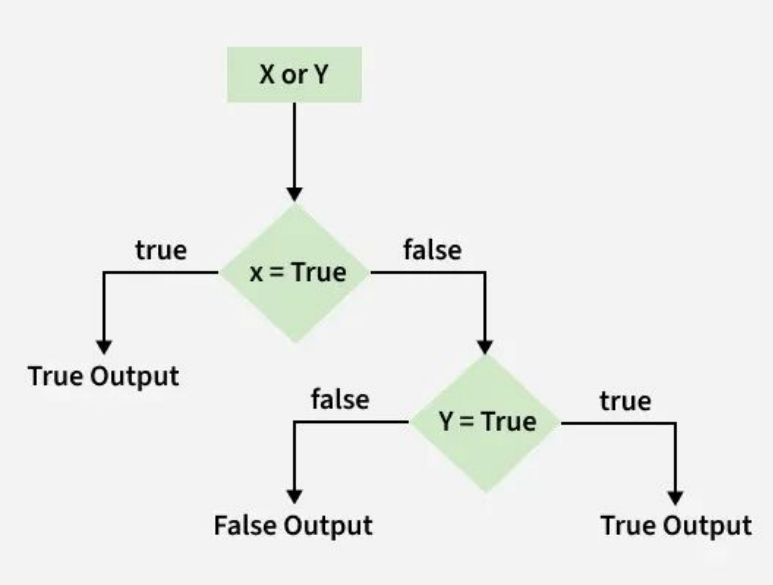
Permitem comparar dois valores e retornar um booleano como resultado.

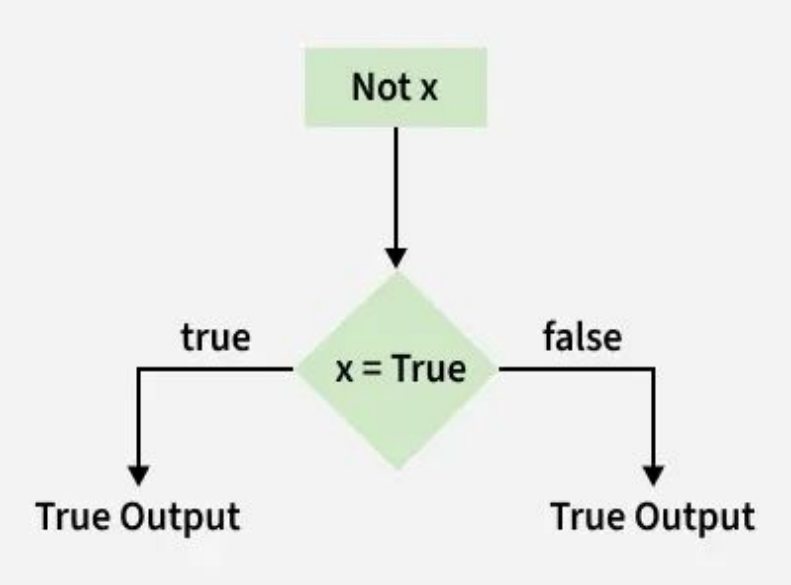
- `>` (“maior que”)
- `<` (“menor que”)
- `==` (“igual a”)
- `!=` (“diferente de”)
- `>=` (“maior ou igual que”)
- `<=` (“menor ou igual que”)

```
2 > 2     retorna False
2 != 3    retorna True
2 == 2    retorna True
4 <= 4    retorna True
```

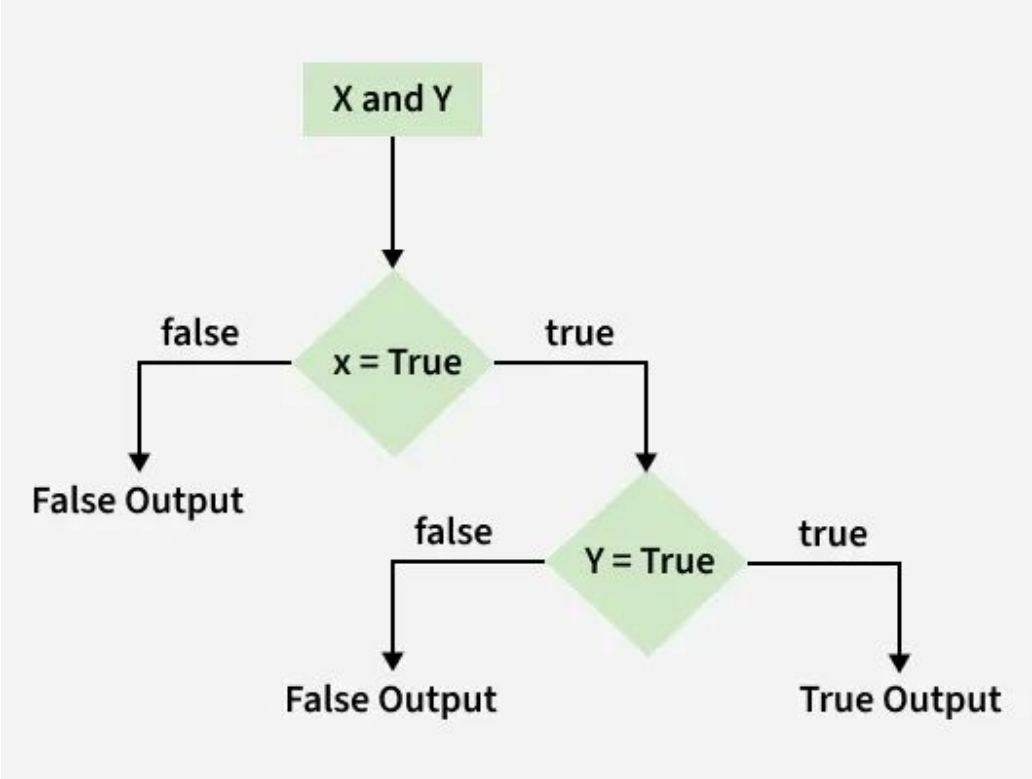
Operadores Lógicos







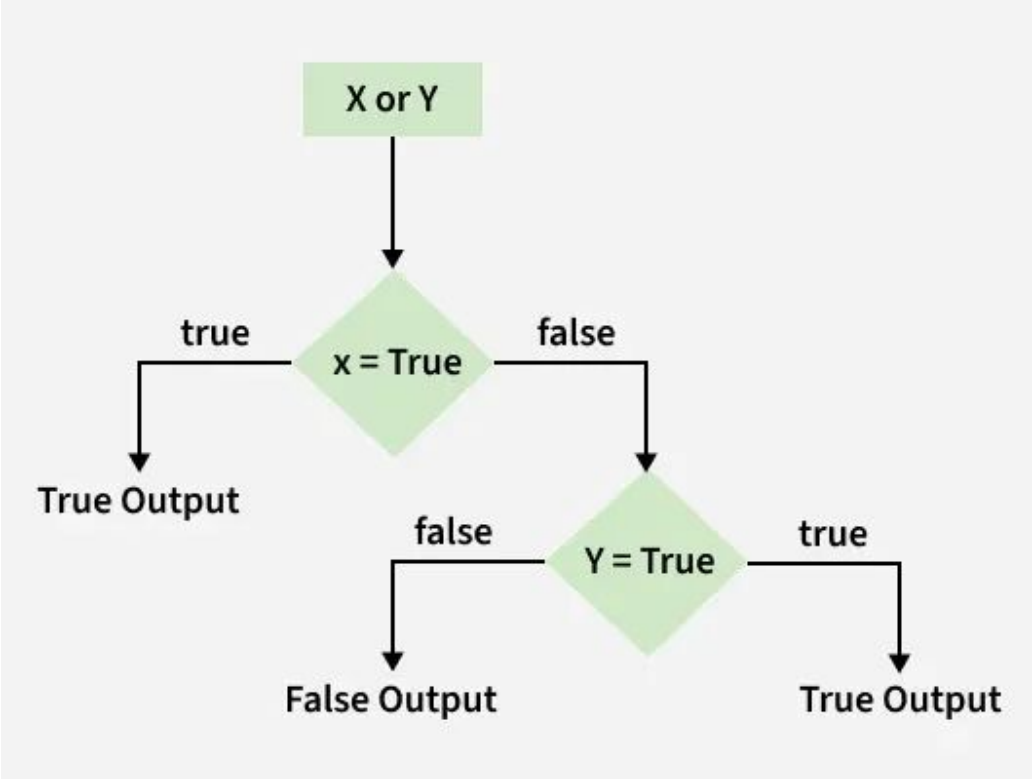
Operadores Lógicos - and



Verifica se ambos os lados da expressão são verdade

`False`	and	`False`	retorna `False`
`True`	and	`False`	retorna `False`
`False`	and	`True`	retorna `False`
`True`	and	`True`	retorna `True`

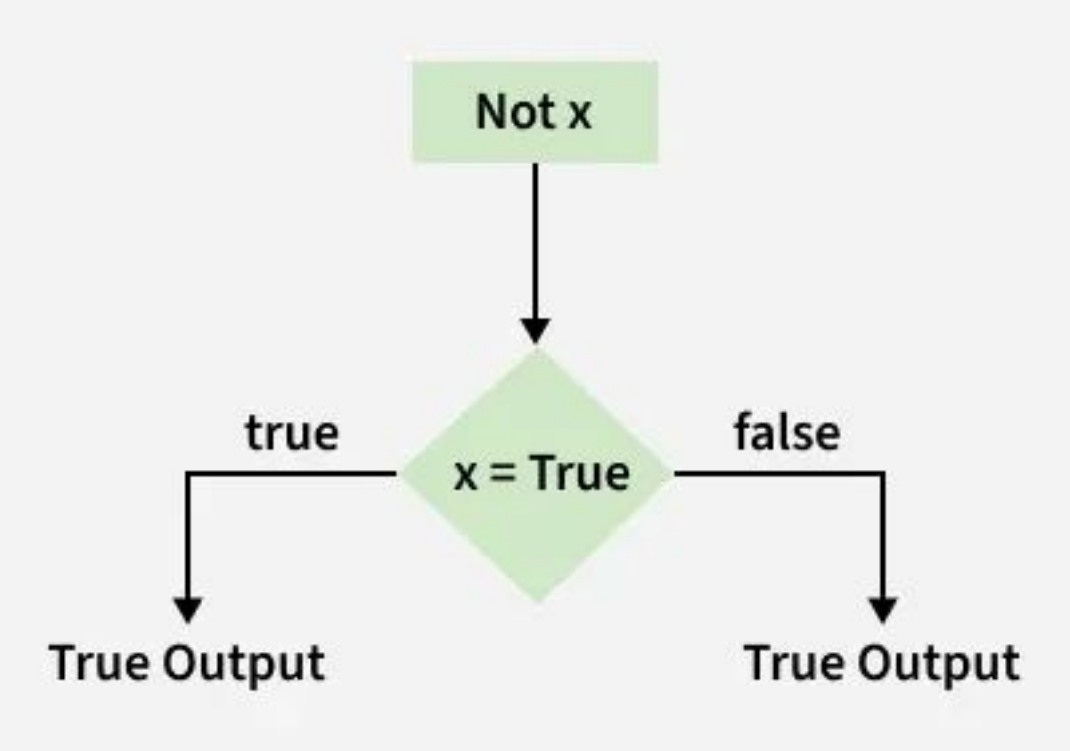
Operadores Lógicos - `or`



Verifica se pelo menos um lado da expressão é verdade.

```
False or  False retorna False
True  or  False retorna True
False or  True  retorna True
True  or  True  retorna True
```

Operadores Lógicos - `not`



Nega o valor de verdade da expressão.

```
not False    retorna True
not True     retorna False
```

Encadeamento de operadores lógicos

Em Python, os operadores podem ser combinados para formar expressões mais complexas. A avaliação sempre segue a ordem **`not`** >> **`and`** >> **`or`**.

`True and False or not False` retorna ...

`False or True and not True` retorna ...

Encadeamento de operadores lógicos

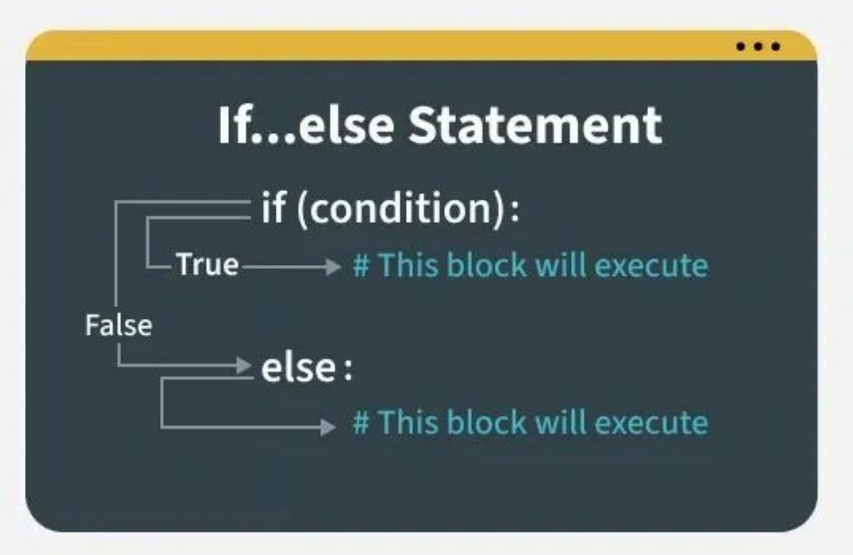
Em Python, os operadores podem ser combinados para formar expressões mais complexas. A avaliação sempre segue a ordem `not` >> `and` >> `or`.

`True and False or not False` retorna `True`

`False or True and not True` retorna `False`

Estruturas Condicionais

A principal utilidade dos operadores lógicos está em permitir a criação de **estruturas condicionais,** que permitem executar certas secções de código apenas se certas condições se verificarem, através do uso de `if` statements.



`if`: Se a condição for verdadeira, executa o código ("body" do `if`).

`elif`: Significa "else if"; se a condição do `if` for falsa, testa a condição de `elif` e executa o código se esta for verdadeira.

`else`: Se nenhuma das condições anteriores se verificar (é verdadeira), executa este código.

Exemplos de Estruturas Condicionais

```
idade = 17

if (idade < 18):
    print(“Menor de idade”)
else:
    print(“Maior de idade”)
```

IMPORTANTE: Ao contrário de outras linguagens, a indentação é muito importante em Python; falhas de indentação levam a erros no programa.

Exemplos de Estruturas Condicionais

```
player_level = 48
boss_level = 50
great_weapon = True

if (player_level > boss_level and great_weapon):
    print("Vitória!")
elif (player_level > boss_level and (not great_weapon)):
    print("Sobreviveste...por um triz")
else:
    print("Derrota")
```

IMPORTANTE: Ao contrário de outras linguagens, a indentação é muito importante em Python; falhas de indentação levam a erros no programa.

<CodeCamp/>
<@FEUP/>

14
fevereiro
2026

Loops e Tipos de Dados Complexos

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

FEUP
ESCOLAS
Programa de captação
e desenvolvimento de talento.

Apoio

acm
Student Chapter

ni

Tipo de Dados Complexos

Até agora aprendemos o que são variáveis: uma caixa com um nome que armazena **UM** objeto. E se pudéssemos guardar **VÁRIOS** objetos na caixa?

Aí é que entram os Tipos de Dados Complexos!

Listas

As Listas são estruturas que armazenam dados e possuem estas 3 propriedades:

1. **Ordenação**: Os dados presentes na lista estão ordenados pelas suas ordens de inserção, inicialmente.
2. **Mutabilidade**: É possível alterar os conteúdos dos dados na lista.
3. **Dinamismo**: As listas podem crescer e encolher, isto é, é possível inserir novos dados ou remover dados existentes.

```
["Code","Camp","@FEUP", "2026"]
```

Listas em Python

Em Python, as listas são inicializadas com parênteses retos []:

```
>> lista = []
```

Os dados são separados por vírgulas.

```
>> lista = [1, 2, 3]
```

Uma das características de Python é que as listas podem ter dados de diferentes tipos:

```
>> lista = [1, “Hello World”, True]
```

(Em outras linguagens de programação, isto geraria um erro.)

Operações Básicas em Listas

Ótimo, agora temos vários dados numa só variável! Mas como é que os usamos? Através da indexação!

```
>> lista = [1, 2, 3]
>> print(lista[0])
```

1

Notem que a indexação tem como **base 0**: o primeiro dado tem índice 0, o segundo tem índice 1 e assim por diante. Para além disso, em Python, é possível usar **indexação negativa**: o índice -1 aponta para o último dado da lista, o -2 para o penúltimo...

Operações Básicas em Listas

Em relação à mutabilidade das listas:

```
>> lista = [1, 2, 3]
>> print(lista)
[1, 2, 3]
>> lista[1] = 3
>> print(lista)
[1, 3, 3]
```

Em relação ao dinamismo das listas:

```
>> lista = [4, 5, 6]
>> print(lista)
[4, 5, 6]
>> lista.append(7)
>> print(lista)
[4, 5, 6, 7]
>> lista.remove(4)
>> print(lista)
[5, 6, 7]
>> lista.pop(0)
>> print(lista)
[6, 7]
```

Resumo de Operações Básicas em Listas

- Aceder a dados: `lista[0]`
- Adicionar um novo dado: `lista.append()`
- Remover um dado existente: `lista.remove()`
- Remover um dado pelo seu índice: `lista.pop()`
- Obter o tamanho da lista: `len(lista)`
- Ordenar uma lista: `lista.sort(reverse= True | False)`
- Imprimir todos os dados da lista no terminal: `print(lista)`
- Copiar uma lista: `lista.copy()`

Nota importante sobre Listas

Imaginando que temos a nossa `lista = [1, 2, 3]` e queremos trabalhar nela sem mantendo uma cópia da lista original.

O mais intuitivo seria fazer `copy = lista`. Mas isto é **errado!**

`[1, 2, 3]` é um objeto em memória do programa enquanto que `lista` podemos pensar nela como um post-it com um nome. Ao fazermos `copy = lista`, apenas estamos a colocar outro post-it ao lado do post-it original. O que isto significa? Qualquer mudança à `copy` é refletida na `lista`.

Para criar realmente uma cópia, basta fazer `copy = lista.copy()`.

Ciclos `while`

Imaginemos agora que queremos fazer operações numa lista onde não saberemos qual será o seu conteúdo nem o seu tamanho. Supondo que queríamos saber quantos dados são menor que 10 e imprimi-los, seria inviável fazer:

```
contador = 0
if (lista[0] < 10):
    print(lista[0])
    contador += 1
if (lista[1] < 10):
    print(lista[1])
    contador += 1
if (lista[2] < 10):
    …
```

Ciclos `while`

E aqui entram os Ciclos While!

Fundamentalmente, ciclos While são uma rotina que é executada enquanto uma determinada expressão é avaliada como verdadeira.

Para o exemplo anterior...

```
i = 0
contador = 0
while (i < len(lista)):
    if (lista[i] < 10):
        print(lista[i])
        contador += 1
    i += 1
```

A expressão dentro de parênteses precisa de retornar True ou False.

Ciclos `for`

Ciclos For são uma versão mais "chique" do ciclo While, isto porque, tudo o que fazemos com um ciclo For, podemos fazer com um ciclo While. Mas um ciclo For permite-nos abstrair certos detalhes.

Em seguida, estão duas versões do exemplo anterior escritas com um ciclo For.

```
contador = 0
for n in lista:
    if (n < 10):
        print(n)
        contador += 1
```

```
contador = 0
for i in range(len(lista)):
    if (lista[i] < 10):
        print(lista[i])
        contador += 1
```

Um ciclo For permite-nos iterar por uma lista, do seu início até ao fim, atribuindo o valor do dado à variável declarada depois da *keyword for*.

Ciclos `for`

Nesta primeira solução, iteramos diretamente pela lista, atribuindo o valor dos dados à variável n.

Nota: a variável `n` não aponta para o objeto na `lista`, logo qualquer modificação a `n` não será refletida na `lista`.

```
contador = 0
for n in lista:
    if (n < 10):
        print(n)
        contador += 1
```

Ciclos `for`

```
contador = 0
for i in range(len(lista)):
    if (lista[i] < 10):
        print(lista[i])
        contador += 1
```

Deste modo, já seria possível modificar os dados da lista através da indexação.

```
>> print(list(range(5)))
[0, 1, 2, 3, 4]
>> print(list(range(1, 10, 2)))
[1, 3, 5, 7, 9]
```

Nesta versão, usamos uma função auxiliar de python:

`range(start, stop, step)`

Esta função cria uma lista com números começando em `start` (o padrão é 0) e acabando em `stop` (não incluído), pulando `step` (o padrão é 1) números a cada número.

Nota: a função range não cria uma lista mas sim um objeto do tipo range. Para obter a lista, precisamos de converter o objeto range numa lista através da função list.

break e continue

```
lista1 = [23, 86, “FEUP”, 451, 8236, 931, None, 1276, 312, False]
lista2 = []
for obj in lista:
    lista2.append(obj)
```

Imaginemos agora que queremos filtrar todos os números menores que 1000 de uma lista para colocar noutra até encontrarmos um **None**.

Por enquanto, esta função copiaria todos os dados da `lista1`, o que não é o nosso objetivo.

break e continue

```
lista1 = [23, 86, “FEUP”, 451, 8236, 931, None, 1276, 312, False]
lista2 = []
for obj in lista:
    if type(obj) != int:
        continue
    lista2.append(obj)
```

A *keyword* **continue** ignora todas as linhas seguintes do ciclo e passa para a próxima iteração na lista.
Neste caso, se o `obj` não for um `int`, passamos para o próximo dado, sem executar o `lista2.append(obj)`.
Ainda assim, não teremos o resultado desejado já que todos os números seriam copiados.

break e continue

```
lista1 = [23, 86, “FEUP”, 451, 8236, 931, None, 1276, 312, False]

lista2 = []

for obj in lista:
    if type(obj) != int:
        if obj == None:
            break
        continue
    lista2.append(obj)
```

A *keyword* **break** ignora todas as instruções seguintes do ciclo e termina-o, executando a próxima instrução fora do ciclo.
Neste caso, se o nosso obj for um **None**, terminamos de filtrar a lista e saímos do ciclo.

Funções

Funções

Uma função é um bloco de código reutilizável que executa uma ação específica. Elas ajudam a dividir programas complexos em partes menores e mais fáceis de gerir.

Para definir uma função, usa-se **def** seguido do nome e parênteses. Dentro dos parênteses, podemos colocar argumentos/ parâmetros.

```
def emitir_alerta():
    print("Atenção: O sistema detetou uma anomalia!")


emitir_alerta() # Chamada da função
```

O Comando `return`

O **`return`** é a saída da função. Ele envia um valor de volta para o ponto onde a função foi chamada, permitindo que esse valor seja guardado numa variável. Uma função sem um **`return`** retorna **`None`**, por padrão.

```
def somar(a, b):
    return a + b


resultado = somar(5, 3) # A variável 'resultado' agora guarda o valor 8
```

Nota: Assim que o Python encontra um **`return`**, ele encerra a execução da função imediatamente. Qualquer código abaixo dele dentro da mesma função será ignorado.

Parâmetros Opcionais

Permitem definir valores padrão para os argumentos, tornando a função mais flexível. Se quem chamar a função não passar esse argumento, o Python usará o valor pré-definido.

Os parâmetros opcionais devem sempre vir necessariamente **depois** dos parâmetros obrigatórios.

```python
def criar_mensagem(user, plataforma="Web"):
    return f"Utilizador {user} acedeu via {plataforma}."


print(criar_mensagem("Ricardo"))           # Usa "Web"
print(criar_mensagem("Ana", "Telemóvel"))  # Usa "Telemóvel"
```

Escopo em Funções (Local vs Global)

O escopo define a "visibilidade" de uma variável.

- **Escopo Local:** Variáveis criadas dentro de uma função. Elas "nascem" quando a função começa e "morrem" quando ela termina.
- **Escopo Global:** Variáveis criadas fora de qualquer função. São acessíveis em todo o ficheiro.

```python
moeda = "EUR" # Global

def mostrar_preco(valor):
    simbolo = "€" # Local
    print(f"Preço: {valor}{simbolo} ({moeda})")


mostrar_preco(50)

# print(simbolo) # ERRO: 'simbolo' não existe fora da função.
```

O Perigo das Variáveis Globais

Variáveis globais tornam o código imprevisível e difícil de corrigir, pois qualquer parte do programa pode alterar o seu valor silenciosamente sem deixar rasto.

Elas criam uma dependência forte que impede a reutilização das funções noutros projetos, pois estas deixam de ser unidades isoladas e autossuficientes.

Além disso, aumentam o risco de conflitos de nomes (poluição do namespace), onde dados importantes podem ser sobrescritos acidentalmente por outras partes do código.

Por fim, violam o princípio do encapsulamento, tornando o fluxo de dados confuso e muito mais difícil de testar.

Exemplo de uma Função

```
ICMS = 0.18

def calcular_preco_final(valor, desconto=0):
    valor_com_desconto = valor - (valor * desconto)
    preco_final = valor_com_desconto + (valor_com_desconto * ICMS)

    return preco_final
```

Exemplo de uma Função

```python
produto_a = calcular_preco_final(100)
produto_b = calcular_preco_final(100, 0.10)

print(f"Produto A: {produto_a:.2f}€") #Produto A: 118.00€
print(f"Produto B: {produto_b:.2f}€") #Produto B: 106.20€
```

Muito obrigado pela vossa atenção!


U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO
FEUP
ESCOLAS
Programa de captação
e desenvolvimento de talento.
acm
Student Chapter
ni