

# Introdução ao Python

## Python Journey

Pedro Gasparinho

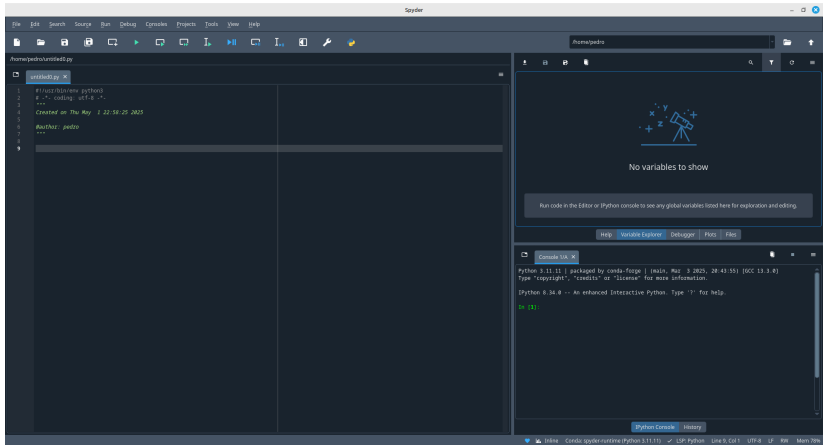
May 2, 2025

Antes de escrevermos o nosso primeiro programa em Python, é necessário saber onde escrevê-lo.

A nossa recomendação é o ambiente de desenvolvimento Spyder, que se adapta perfeitamente para quem não tem experiência de programação. Para além disso, será o ambiente utilizado nestes materiais.

Alunos com experiências prévias de programação poderão usar outras alternativas com as quais já estejam habituados.

# Ambiente de desenvolvimento Spyder



Ambiente de desenvolvimento Spyder

O Spyder, por defeito, é composto por 3 grupos de janelas:

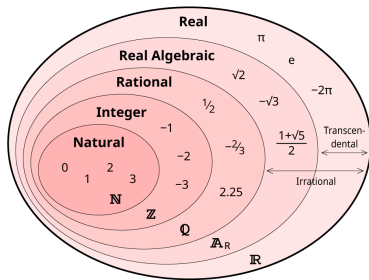
- À direita:
  - e em cima, há janelas auxiliares de ajuda, de ferramentas para ver o estado do programa, de gráficos produzidos ou para ver a pasta de trabalho.
  - e em baixo, encontra-se a consola/interpretador, onde se podem testar comandos Python de forma rápida, embora não seja prática para usos mais complexos.
- À esquerda, é possível visualizar um ou mais ficheiros Python. Útil para casos de uso mais extensos ou que envolvem mais do que um ficheiro.

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

*Python*

- *print()* - Função
- *'Hello World!'* - Valor do tipo string

# Conjuntos Numéricos



[https://thinkzone.wlonk.com/Numbers/RealSet\\_w1000.png](https://thinkzone.wlonk.com/Numbers/RealSet_w1000.png)

- Existe a necessidade de agrupar números com dadas características em conjuntos diferentes.
- Estes conjuntos são normalmente construídos de forma incremental.
- No entanto, quando necessário é possível restringir o conjunto ou subconjunto a usar.

Similarmente à Matemática, em Python, existem conjuntos de valores, denominados **tipos de dados**.

Os tipos de dados não se limitam apenas a números. Também é possível representar **texto**, **verdadeiros e falsos**, etc.

Um valor pode pertencer a vários tipos de dados, mas por norma, tem uma representação diferente por cada um, por exemplo:

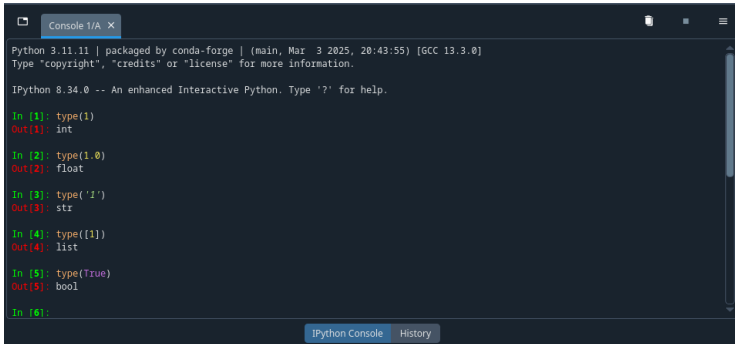
- **1** é considerado como um **inteiro**,
- **1.0** é considerado como um **número real** (float).

# Tipos de dados em Python

Tipo de dados	Python	Exemplos
Inteiros (Integers)	<i>int</i>	1, -4, 0
Reais (Floats)	<i>float</i>	1.0, -6.2, 3.14
Booleanos (Booleans)	<i>bool</i>	True, False
Texto (String)	<i>str</i>	'1', "Hello, world", """Olá"""
Listas	<i>list</i>	[1,2,3], [], ["Olá", [1], true]
Entre outros	...	...



# Função *Type()*



```
Python 3.11.11 | packaged by conda-forge | (main, Mar 3 2025, 20:43:55) [GCC 13.3.0]
Type "copyright", "credits" or "license" for more information.

IPython 8.34.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: type(1)
Out[1]: int

In [2]: type(1.0)
Out[2]: float

In [3]: type('1')
Out[3]: str

In [4]: type([1])
Out[4]: list

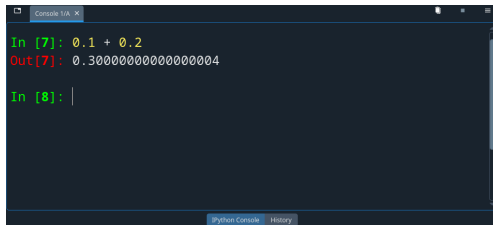
In [5]: type(True)
Out[5]: bool

In [6]:
```

Interpretador de Python

A função *type()* identifica o tipo de dados de um valor.

# Representação de Números Reais

A screenshot of a Python console window. The window has a title bar that says "Console 1/A". Inside the console, the input "In [7]: 0.1 + 0.2" is shown in green. The output "Out[7]: 0.30000000000000004" is shown in red. Below the output, the input "In [8]: |" is shown in green, with a cursor at the end. At the bottom of the console, there are two buttons: "Python Console" and "History".

```
Console 1/A  
In [7]: 0.1 + 0.2  
Out[7]: 0.30000000000000004  
In [8]: |  
Python Console History
```

Interpretador de Python

Por vezes operações aritméticas com *floats* produzem valores estranhos.

Isto acontece devido a um fenómeno semelhante ao seguinte:

$$\frac{1}{3} + \frac{1}{3} + \frac{1}{3} = 1$$

$$0.33333... + 0.33333... + 0.33333... = 0.99999...$$

É impossível representar corretamente números com **dízimas infinitas**, pois o espaço de armazenamento dos computadores é **finito**.

## Observação

0.1 e 0.2 não são dízimas infinitas

A observação anterior é **parcialmente correta**, no sistema decimal (base 10, e que usamos no dia a dia) é verdade, mas o mesmo não se aplica no sistema binário (base 2).

Internamente os computadores utilizam o **sistema binário** para guardar toda a informação, desde números a texto, etc.

Existem técnicas para separar a informação e normas para representar os diferentes tipos de dados. Exemplos:

- IEEE 754 para *floats*,
- ASCII para caracteres.

**Nota:** nenhuma norma ou técnica faz parte do escopo destes materiais.

Para aprofundar mais o tema da representação de números reais, recomendam-se os seguintes links: [Porquê  \$0.1 + 0.2 \neq 0.3\$ ?](#) e [Conversor de \*Floats\*](#).

- São obrigatoriamente delimitadas ou por apóstrofes (`'...'`) ou por aspas (`"..."`).
- Escolher as aspas como delimitador permite utilizar os apóstrofes dentro do conteúdo da string e vice-versa.
- Para além disso, é possível usar o mesmo delimitador 3 vezes seguidas (ex: (`"""..."""`)) para permitir que a string tenha múltiplas linhas. O que geralmente é utilizado para comentários.

- As listas servem para guardar múltiplos valores de forma ordenada e que permite repetições.
- Em Python, ao contrário de muitas outras linguagens, as listas podem conter valores com diferentes tipos de dados.

## Expressões aritméticas

---

Um dos usos mais simples das linguagens de programação é como calculadora, através dos seguintes operadores aritméticos básicos:

$+$ ,  $-$ ,  $*$ ,  $/$

Existem também operadores ligeiramente mais avançados:

- `//` representa a divisão inteira. Ex:
  - $4//2 = 2$
  - $5//2 = 2$
- `%` representa o resto da divisão inteira. Ex:
  - $4\%2 = 0$
  - $5\%2 = 1$
- `**` representa a potência. Ex:
  - $2**4 = 16$
  - $9**0.5 = 3.0$



# Expressões aritméticas



```
Console 1/A X

In [8]: 2+2
Out[8]: 4

In [9]: 8-0.5
Out[9]: 7.5

In [10]: 4*0.1
Out[10]: 0.4

In [11]: 5/0
-----
ZeroDivisionError                                Traceback (most recent call last)
Cell In[11], line 1
----> 1 5/0

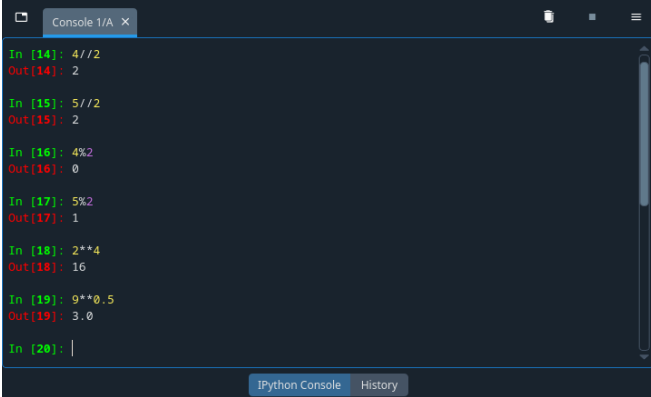
ZeroDivisionError: division by zero

In [12]: 4/2
Out[12]: 2.0

IPython Console History
```

Operadores aritméticos simples

# Expressões aritméticas



The screenshot shows an IPython console window with a dark background. The title bar at the top says "Console 1/A X". The console contains several lines of input and output. Each input line is preceded by "In [n]:" and each output line by "Out[n]:". The inputs are arithmetic expressions, and the outputs are the results. The expressions include integer division, modulo, and exponentiation. The console also has a scrollbar on the right and buttons for "IPython Console" and "History" at the bottom.

```
Console 1/A X

In [14]: 4//2
Out[14]: 2

In [15]: 5//2
Out[15]: 2

In [16]: 4%2
Out[16]: 0

In [17]: 5%2
Out[17]: 1

In [18]: 2**4
Out[18]: 16

In [19]: 9**0.5
Out[19]: 3.0

In [20]: |
```

IPython Console History

Operadores aritméticos mais avançados

## Erro: Divisão por 0

Tal como na Matemática, a divisão por 0 não está definida e produz um erro durante a execução. Também é aplicável à divisão inteira e ao resto.

Para a maioria dos operadores, se pelo menos um dos operandos for *float*, então o resultado também será *float*, caso contrário será *int*.

A divisão real é uma exceção já que produz sempre um *float*.

Surpreendentemente a divisão inteira e o resto podem produzir floats, mas devido às suas propriedades são números inteiros com 0 como casa decimal.

```
Console 1/A X

In [1]: 1/0
-----
ZeroDivisionError                                Traceback (most recent call last)
Cell In[1], line 1
----> 1 1/0

ZeroDivisionError: division by zero

In [2]: 1//0
-----
ZeroDivisionError                                Traceback (most recent call last)
Cell In[2], line 1
----> 1 1//0

ZeroDivisionError: integer division or modulo by zero

In [3]: 1%0
-----
ZeroDivisionError                                Traceback (most recent call last)
Cell In[3], line 1
----> 1 1%0

ZeroDivisionError: integer modulo by zero

In [4]:
```

Divisões por 0



The screenshot shows an IPython console window with a dark background. The title bar at the top says "Console 1/A" with a close button. The console contains the following input-output pairs:

```
In [9]: type(4/2)
Out[9]: float

In [10]: type(4//2)
Out[10]: int

In [11]: type(4.0//2)
Out[11]: float

In [12]: 4/2
Out[12]: 2.0

In [13]: 4//2
Out[13]: 2

In [14]: 4.0/2
Out[14]: 2.0

In [15]: |
```

At the bottom of the console, there are two tabs: "IPython Console" (which is active) and "History".

Divisões por 0

E será que o Python permite outras operações, como o **logaritmo**?

**Sim**, através de uma biblioteca matemática com mais operações e constantes. E pode ser vista com mais detalhe [aqui](#).

Para fazer uso desta biblioteca, primeiramente deve-se fazer o *import* para carregar os seus conteúdos, os quais podem ser usados através do prefixo **math**:

```
import math
```

*Python*

```
print(math.pi)
print(math.e)
print(math.log(2*math.e))
print(math.sqrt(16))
```

## Variáveis

---

Escreva um programa em Python que calcula o valor da seguinte expressão:

$$\sqrt{6^4 + 7 * 2^3 + 5^2 - 8} + \frac{1}{\ln(9^2 + 1)}$$

Uma solução possível é:

```
import math
```

*Python*

```
print(math.sqrt(6**4 + 7*2**3 + 5**2 - 8) + 1/(math.log(9**2 +
```

**Nota:** O exemplo anterior não cabe prepositadamente no slide.



## Boa Prática

As linhas de código devem ser relativamente curtas, pois facilitam a leitura e diminuem a chance de ter erros.

Em vez da solução anterior, devemos usar variáveis para guardar valores intermédios, de modo a seguir as boas práticas:

```
import math
```

*Python*

```
raiz = math.sqrt(6**4 + 7*2**3 + 5**2 - 8)  
fracao = 1/(math.log(9**2 + 1))
```

```
print(raiz + fracao)
```

## Notas:

- A escolha dos nomes anteriores é apenas simbólica.
- Era perfeitamente aceitável usar mais variáveis.

É um símbolo utilizado para descrever um valor não especificado.

Por vezes o seu valor ainda é desconhecido, mas eventualmente pode ser calculado:

$$3x - 4 = 8$$

$$3x = 12$$

$$x = 4$$

Outra opção, é quando a variável pode ser substituída por uma constante, por exemplo no contexto de uma função, tal como:

Seja  $f(x) = 2x + 5$ , então:

$$f(1) = 2 * 1 + 5 = 7$$

**Nota:** Este tipo de variáveis são também denominados de parâmetros ou argumentos.

Ao contrário da Matemática, é um **identificador** utilizado para **referir** a um **valor** guardado na memória do computador.

Por norma, os valores em memória podem ser alterados durante a execução do programa, mas cada linguagem tem a sua estratégia para lidar com a gestão da memória.

A gestão de memória em Python é feita de forma implícita, por simplicidade, e os valores das variáveis podem ser **alterados livremente**.

A **criação** e **atualização** de variáveis é feita da mesma forma, apenas depende se o identificador já foi definido anteriormente:

```
x = 10      # variavel x criada
y = 5      # variavel y criada

x = x + 1   # variavel x atualizada
print(x)    # imprime 11
print(y)    # imprime 5

x = "Ok"    # variavel x atualizada
print(x)    # imprime Ok
print(y)    # imprime 5
```

*Python*

**Note:** Ao contrário de outras linguagens, o Python permite que uma variável ao ser atualizada possa mudar para um tipo de dados completamente diferente.

Uma constante, por definição, não deve sofrer alterações no seu valor durante a execução. Caso contrário poderia levar a erros inesperados nos nossos programas.

Ao contrário de outras linguagens, em Python **não** existem suporte para constantes (pelo menos de forma simples e direta).

Em vez disso, cabe ao Programador usar variáveis e ser **cuidadoso** ao ponto de não atualizar o seu valor, de modo a simular constantes.

## Boa Prática

Todas as letras no nome de uma constante devem ser maiúsculas.

As seguintes regras e convenções de nomenclatura aplicam-se tanto a variáveis como constantes:

## Regras

- Apenas é permitido usar letras, números e underscore " \_"
- O primeiro caracter não pode ser um número
- Os nomes são sensíveis a letras maiúsculas e minúsculas
- Não podem ser uma das seguintes **palavras reservadas**

## Convenções e boas práticas

- Evitar nomes demasiads curtos (Exceção: fórmulas matemáticas com nomes de variáveis curtos, ex: 'x' ou 'y')
- O nome de uma variável deve ser descritivo do seu significado
- Evitar nomes demasiados gerais ou longos

# Funções

---

Escreva um programa em Python que calcula a distância euclidiana entre os pontos (3, 4) e (7, 1). A fórmula para dois pontos  $(x_1, y_1)$  e  $(x_2, y_2)$  é:

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Talvez a solução mais intuitiva seja:

```
import math
```

*Python*

```
print(math.sqrt((3 - 7)**2 + (4 - 1)**2))
```

E se nos pedirem para calcular a distância entre outros dois pontos? Simples, fazer copy paste e alterar os valores!



```
import math
```

*Python*

```
print(math.sqrt((3 - 7)**2 + (4 - 1)**2))
print(math.sqrt((0 - 0)**2 + (0 - 1)**2))
print(math.sqrt((1 - 4)**2 + (3 - 7)**2))
print(math.sqrt((-2 - 3)**2 + (5 + 1)**2))
print(math.sqrt((0 - 5)**2 + (0 - 5)**2))
print(math.sqrt((10 + 6)**2 + (-4 - 8)**2))
print(math.sqrt((100 - 150)**2 - (200 - 250)**2))
print(math.sqrt((5 - 2)**2 + (9 - 6)**2))
print(math.sqrt((8 - 3)**2 + (2 + 3)**2))
print(math.sqrt((4 - 0)**2 + (7 - 2)**2))
print(math.sqrt((-5 - 5)**2 + (1 + 4)**2))
print(math.sqrt((6 + 4)**2 + (-2 - 3)**2))
print(math.sqrt((0 - 8)**2 + (5 - 9)**2))
print(math.sqrt((7 - 2)**2 + (3 + 2)**2))
print(math.sqrt((-3 - 1)**2 + (-6 - 2)**2))
print(math.sqrt((9 - 4)**2 + (1 - 5)**2))
```

Como podemos ver o erro pode não ser óbvio, pelo formatação do exemplo com os pontos (100, 200) e (150, 250). Para além disso, a simplificação entre dos sinais também pode resultar em enganos.

A repetição de código isto introduz alguns **problemas**:

- A fórmula da distância é relativamente curta, mas copiar várias vezes excertos mais complexos e longos tornam o código **difícil de ler**.
- Se detetarmos um erro em código repetido seria necessário substituir em **vários locais**, o que não é desejável, e inclusivamente ficar a faltar em algum sítio.

Qual será a melhor solução?

Uma solução mais adequada seria definir uma função e reutilizá-la:

```
import math
```

*Python*

```
def dist(x1, y1, x2, y2):  
    return math.sqrt((x1 - x2)**2 + (y1 - y2)**2)  
  
print(dist(3, 4, 7, 1))  
print(dist(0, 0, 0, 1))  
# etc.
```

- A ordem dos parâmetros importa (Existem truques para que a ordem não interesse, mas não serão apresentados aqui).
- Com algumas exceções, os nomes não interessam, mas por boa prática deve ser algo alusivo ao comportamento esperado.
- Porque devemos usar funções? Seria necessário ter um botão numa calculadora que dê especificamente o resultado de  $\ln(1 + 5/2)$ ? Em vez disso, o que é realmente necessário são os números, a soma, a divisão, a função de logaritmo, etc.

Para tornar a solução anterior ainda melhor devemos usar **ajudas de tipos e comentários**:

```
import math
```

*Python*

```
def dist(x1, y1, x2, y2: float) -> float:
    """ Calcula a distancia eucliadiana entre dois pontos
        x1, y1 - Coordenadas do ponto 1
        x2, y2 - Coordenadas do ponto 2
        Retorna a distancia como numero real """
    return math.sqrt((x1 - x2)**2 + (y1 - y2)**2)

print(dist(3, 4, 7, 1))
print(dist(0, 0, 0, 1))
```

# Porquê documentar o código?

- Os humanos, no geral, têm capacidade de memória limitada e é expectável que após algumas meses, semanas ou até dias não se lembrem do código que escreveram.
- Aliás, ler código não é uma tarefa fácil, especialmente, código de grandes dimensões desenvolvido por outras pessoas.
- A documentação do código, através de ajudas de tipo e comentários é essencial para facilitar a leitura do código em ambas as situações.

- Os comentários têm como objetivo documentar o código ou por vezes deixar recados e como tal não afetam o resultado das operações.
- Para pequenos comentários ou notas, deve-se usar o comentário de linha (`#`)
- Para comentários de maior dimensão, deve-se usar o comentário multi-linha (ex: `"""..."""`). Nota: tecnicamente são considerados como string, mas para não influenciar o código não devem ser atribuídos a variáveis ou usados em operações com strings.

- Cada Programador pode escolher o seu estilo de documentação
- No entanto, recomenda-se usar comentários multi-linha com a seguinte estrutura:
  - Descrição geral do comportamento da função.
  - Descrição sobre cada parâmetro (Se fizer sentido pode-se juntar mais do que um na mesma descrição).
  - Descrição do(s) valor(es) a retornar.
  - Nota: as frases devem ser preferencialmente curtas.



- Em Python, ao contrário de outras linguagens de programação, não é obrigatório declarar o tipo das variáveis, em especial nos parâmetros,
- No entanto, é possível declarar de forma opcional os tipos dos parâmetros e de retorno de uma função para ajudar na leitura e compreensão do código.
- As ajudas de tipo não são verificadas internamente pelo Python nem produzem erros, por isso é necessário ter cuidado com possíveis enganos ao declarar os tipos.

- Outro aspeto bastante importante no desenvolvimento de funções e de programas, no geral, é verificar que está efetivamente correto.
- A técnica de verificação mais utilizada são testes unitários, que consiste em garantir que os resultados estão corretos para um número suficientemente grande de casos de teste.
- No entanto, para programas minimamente grandes torna-se impossível de verificar manualmente todos os casos possíveis, pois este é muito grande.
- Os testes unitários servem para indicar a presença bugs, mas não garantem a ausência deles (excepto de os testes conseguirem cobrir todos os casos).
- Para garantir a correção são necessários técnicas muito mais avançadas que recorrem a provas lógicas e matemáticas, e que não serão lecionadas neste curso.

```
import math
```

*Python*

```
def test_dist():  
    assert dist(0, 0, 0, 0) == 0  
    assert dist(0, 0, 1, 0) == 1  
    assert dist(0, 0, -1, 0) == 1  
    assert dist(0, 0, 0, 4) == 2  
    assert dist(0, 0, 0, -4) == 2  
    assert dist(3, 4, 7, 1) == 5  
    assert dist(-3, -4, -7, -1) == 5  
    assert dist(-2, 0, 2, 0) == 4  
    x = math.sqrt(2)/2  
    assert dist(-x, x, x, -x) == 2  
test_dist()
```

- Para definir um teste unitário em Python podemos definir uma nova função. O sufixo `test_` não é obrigatório, mas é uma convenção de nomenclatura para facilmente identificar o objetivo da função.
- Devemos também fazer uma chamada à função para garantir que corre e que os testes são aplicados.
- O `assert` é um comando que está à espera de uma comparação, para isso devemos efetuar uma chamada à função a testar com valores concretos e comparar com o valor esperado.
- Normalmente as comparações são feitas através da igualdade já que é a expressão mais forte, mas qualquer operador de desigualdade (`!=`, `<`, `<=`, `>`, `>=`) pode ser usado.

- A função de teste deve conter tantas "classes" de exemplos quanto possíveis ou conhecidas para um dado problema, neste caso:
  - Retas horizontais
  - Retas verticais
  - Retas diagonais "perfeitas"
  - Retas diagonais "não-perfeitas"
  - Do mesmo ponto ao mesmo ponto
  - Pontos com coordenadas positivos
  - Pontos com coordenadas nulas
  - Pontos com coordenadas negativas
  - Etc.
- Como podemos ver, mesmo para uma função relativamente simples, é impossível testar exaustivamente. Por isso devemos ter uma função de teste relativamente grande e variada para nos convencer que a função está correta.

## Condicionais

---

Escreva um programa em Python que expresse a seguinte função:

$$f(x) = \begin{cases} x + 2 & \text{se } x < 0 \\ -x - 4 & \text{caso contrário} \end{cases}$$

Atualmente, poderíamos chegar a esta solução:

```
def ffst_branch(x: float) -> float:
    """ First branch of function f
        Requires x < 0
        Returns x+2 """
    return x+2

def fsnd_branch(x: float) -> float:
    """ Second branch of function f
        Requires x >= 0
        Returns -x-4 """
    return -x-4
```

*Python*

A solução anterior tem alguns problemas:

- Estamos a codificar  $f$  em duas sub-funções, quando na verdade gostaríamos de ter uma só função.
- Para obter o valor correto da função  $f$ , é necessário seleccionar manualmente qual sub-função usar, consoante o valor de  $x$ .
- Nada impede o programador de usar a sub-função errada.

Com isto introduz-se o bloco *if...else*:

```
def f(x: float) -> float:
    if x < 0:
        return x+2
    else:
        return -x+4
```

*Python*



E se a função tiver mais do que 2 casos?

$$g(x) = \begin{cases} -3 - x & \text{se } x \leq -3 \\ x + 3 & \text{se } -3 < x \leq 0 \\ 3 - 2x & \text{se } 0 < x \leq 3 \\ x - 4 & \text{se } 3 < x \end{cases}$$

Deve-se usar tantos *elif* quanto necessários:

```
def f(x: float) -> float:
    if x <= -3:
        return -3-x
    elif x <= 0: #implicitly -3 < x <= 0
        return x+3
    elif x <= 3: #implicitly 0 < x <= 3
        return 3-2*x
    else: #implicitly 3 < x
        return x-4
```

Python

- Qualquer bloco condicional tem que começar com um e um só *if*, caso contrário são considerados como blocos diferentes.
- O *else* é opcional, mas se existir deve ser o último elemento do bloco. Representa o "caso contrário" ou "tudo o resto".
- O *elif* também é opcional e não existem restrições outras restrições sobre o uso do mesmo.
- O bloco condicional tem uma natureza sequencial, isto significa que vai percorrer os vários casos até encontrar o 1º que seja verdadeiro e executa o código correspondente a esse caso. E não verifica mais nenhum caso.
- Outra característica derivada da natureza sequencial, é que permite simplificar as condições de cada caso, já que naquele ponto do bloco as condições anteriores são consideradas como falsas.

- Considera-se um bloco os elementos que respeitem as condições anteriores e estejam ao mesmo nível de indentação.
- A "regra" da exclusão de casos anteriores só se aplica a elementos do mesmo bloco. Cada bloco é independente, mesmo que um tenha superioridade hierárquica devido à indentação.

## Ciclo For

---

Escreva um programa em Python que imprime os números de 0 a 9. Atualmente, poderíamos chegar a esta solução:

```
print(0)  
print(1)  
#...  
print(9)
```

*Python*

E se nos pedissem para escrever os números de 0 a 99? Iríamos escrever manualmente isso tudo?

A resposta é não! Para repetir a mesma instrução (ou conjunto de instruções) várias vezes devemos usar o ciclo for:

```
def imprime_numeros_de_zero_a_n(n: int) -> None:    Python
    for i in range(n):
        print(i)

imprime_numeros_de_zero_a_n(n)
```

A função range() retorna uma sequência de números inteiros, e é necessário ter em conta 3 variantes, com base no número de argumentos:

- range(y) retorna a sequência de 0 a  $y - 1$ .  
**Exemplo:** range(5) corresponde a 0, 1, 2, 3, 4.
- range(x, y) retorna a sequência de x a  $y - 1$ .  
**Exemplo:** range(1, 5) corresponde a 1, 2, 3, 4.
- range(x, y, z) retorna a sequência de x a  $y - 1$  com saltos de z em z.  
**Exemplo:** range(1, 5, 2) corresponde a 1, 3.

# Função range() e ciclo for

- Como referido anteriormente, o ciclo for tem como objetivo repetir as mesmas instruções várias vezes.
- Para estabelecer o número de vezes que o ciclo corre usa-se a função range()
- Dentro do ciclo for define-se, também, uma variável que irá receber os valores da sequência um de cada vez e usá-los em computações.
- Se apenas estivermos interessados no número de vezes que as instruções são repetidas, e não nos valores da sequência, podemos omitir a variável de ciclo com o "underscore" (\_).  
Por exemplo:

```
for _ in range(10):  
    print("Hello World!")
```

*Python*



## Listas

---

As listas são um tipo de dados especial que permitem guardar valores de forma ordenada. E como tal é possível aceder às várias posições da lista da seguinte forma:

```
l = [3, 6, 1, 2] Python  
print(l[0]) # imprime 3  
print(len(l)) # imprime 4  
print(l[4]) # erro - IndexError: list index out of range
```

- Os índices da lista começam em 0 e são números inteiros.
- Para obter o tamanho de uma lista deve-se usar a função `len()`.
- Aceder a um índice superior ou igual ao tamanho da lista resulta num erro que interrompe a execução do programa.
- Alternativamente é possível usar índices negativos entre `-len(l)` e `-1`

Para percorrer os valores (iterar) uma lista de forma segura, isto é sem obter nenhum erro, devemos usar a seguinte estratégia:

```
l = [3, 6, 1, 2] Python  
for i in range(len(l)): # equiv. a range(0, len(l), 1)  
    print(l[i])
```

Alternativamente podemos usar a seguinte abreviatura:

```
l = [3, 6, 1, 2] Python  
for e in l:  
    print(e)
```

A variável de ciclo `e` vai receber os valores da lista um de cada vez. Esta abreviatura é conhecida como ciclo `foreach`.

No entanto, esta abreviatura só funciona se quisermos percorrer **todos** os elementos de **uma** lista. Para iterar sobre parte de uma lista ou sobre mais do que uma lista já não é ideal:

```
l = [3, 6, 1, 2] Python  
for i in range(1, 3): # imprime 6 e 1  
    print(l[i])  
  
v = [3, 5, 1, 7]  
for i in range(len(l)): # imprime 3 e 1  
    if l[i] == v[i]:  
        print(l[i])
```

Estes dois exemplos, não funcionariam com a abreviatura anterior, pois nesse não existe maneira de expressar o valor dos índices.

As listas em Python são consideradas como mutáveis, isto significa que é possível alterar o seu conteúdo interno, como por exemplo:

```
l = [3, 6, 1, 2]                                     Python
l[0] = 7
print(l) # imprime [7, 6, 1, 2]
```

A mutabilidade é uma escolha de desenho das linguagens de programação, que tem as suas vantagens e desvantagens. Isto é um tema bastante complexo e que não vai ser discutido em detalhe, mas é importante reter a ideia de que a grande vantagem da mutabilidade é que lista não tem que ser duplicada em memória, e a grande desvantagem é aquilo que se chama 'aliasing'.

```
l = [3, 6, 1, 2]
v = l
l[0] = 7
print(v) # imprime [7, 6, 1, 2]
```

Python

Diz-se que *v* é um 'alias' de *l*, pois qualquer alteração efetuada sobre *l* também afetará *v* e vice-versa, pois ambos representam o mesmo espaço de memória no computador. O uso de 'aliasing' é desencorajado, pois pode resultar em erros inesperados.

O 'aliasing' não se aplica a tipos de dados não mutáveis, como int, float, bool ou string:

```
num = 4
dup = num
num = 8
print(dup) # imprime 4
```

Python

Existem várias funções para manipular listas. Assuma que existe uma lista *l*:

- *l.append(x)* - Adiciona o elemento *x* ao final da lista *l*.
- *l.extend(v)* - Adiciona os elementos da lista *v*, por ordem, no final da lista *l*.
- *l.insert(i, x)* - Adiciona o elemento *x* no índice *i*. Os elementos já existentes com índice *i* ou superior sofrem um desvio de uma posição para a direita.
- *l.remove(x)* - Remove o primeiro elemento com valor *x*. Se não encontrar resulta em erro.
- *l.pop()* - Remove o primeiro elemento da lista *l*.
- *l.pop(i)* - Remove o elemento com índice *i*.
- *l.reverse()* - Inverte a ordem dos elementos da lista *l*.
- *l.copy()* - Produz uma cópia da lista *l*.

Para resolver alguns problemas pode ser necessário criar uma sub-lista, imagine-se a primeira metade. Uma possível solução seria:

```
l = [4, 2, -1, 5, 5, 3, 9, -4]
res = []
half = len(l)//2
for i in range(half + 1):
    res.append(l[i])
```

*Python*

No entanto, o Python tem uma abreviatura disponível para estes casos:

```
l = [4, 2, -1, 5, 5, 3, 9, -4]
half = len(l)//2
res = l[:half + 1]
```

*Python*



As slices funcionam de forma similar à função `range()`, no entanto, a função `range` devolve uma sequência, enquanto as slices criam uma nova sub-lista, sem alterar a lista original:

- Seja `l = [3, 6, 4, 2, 5, 1, 0]`
- `l[x : y : z]` - Produz a sub-lista a partir dos índices `x` a `y - 1` e com saltos de `z` em `z`.

**Exemplo:** `l[1 : 5 : 2]` produz `[6, 2]`

- `l[x : ]` - Produz a sub-lista a partir do índice `x` até ao fim.

**Exemplo:** `l[3]` produz `[2, 5, 1, 0]`

- `l[: y]` - Produz a sub-lista a partir do início até ao índice `y`.

**Exemplo:** `l[: 4]` produz `[3, 6, 4, 2]`

- `l[: : z]` - Produz a sub-lista de início ao fim, mas com saltos de `z` em `z`.

**Exemplo:** `l[: : 3]` produz `[3, 2, 0]`

- Similar para as restantes combinações...

## Ciclo while

---

- O ciclo for requer um número concreto de iterações, seja este conhecido ou desconhecido à priori, mas representado através de uma variável.
- O ciclo while requer uma condição lógica que é verificada no início de cada iteração (mesmo na primeira). Se a condição for verdadeira então o código dentro do ciclo é executado, caso contrário passa para a próxima instrução a seguir ao ciclo.
- O ciclo while é mais expressivo, pois permite expressar o ciclo infinito (quando a condição é apenas *true*) e ciclos onde não existe um número concreto de vezes que irá executar ou pelo menos um majorante.
- Devido a isto, o ciclo while é perfeito para lidar com o input de utilizadores e números aleatórios.

# Matrizes

---

- Para representar matrizes é possível usar uma lista de listas.
- Sendo uma matriz uma lista de listas, tanto a lista exterior, como as listas interiores (linhas da matriz) usufruem dos mesmos mecanismos que uma lista normal, seja para acesso a uma posição, iteração ou funções.
- Para aceder ao elemento da linha  $i$  e da coluna  $j$  numa matriz  $m$ , usa-se  $m[i][j]$ .