

# Introdução ao Python

## Python Journey

Pedro Gasparinho

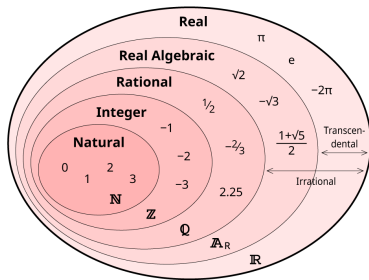
April 29, 2025

```
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 e noutras linguagens de Programação, existem conjuntos de valores, denominados tipos de dados.
- No entanto, estes não se limitam apenas a números, existem também formas de representar texto, verdadeiros ou falsos, entre outros.
- Um valor pode ter diferentes representações, mas cada representação apenas está associada a um tipo de dados. Por exemplo, 1 é um valor inteiro, enquanto 1.0 é considerado como um número real (float).

- Números Inteiros (Integers)
  - Denominado *int*
  - Exemplos: *1, -4, 0*
- Números Reais (Floats)
  - Denominado *float*
  - Exemplos: *1.0, -6.2, 3.1415926*
- Valores lógicos (Booleans)
  - Denominado *bool*
  - Exemplos: *True, False*

- Texto (Strings)
  - Denominado *str*
  - Exemplos: `'1'`, `"Hello, world"`, `"""Olá"""`
- Listas
  - Denominado *list*
  - Exemplos: `[1, 2, 3]`, `[]`, `["Olá", "Olé"]`, `["Olá", 1, ["ok"], true]`
- Entre outros...

- Não é possível representar fidedignamente todos os números reais, como por exemplo, os números com dízimas infinitas, já que os computadores têm espaço de armazenamento finito.
- Outro exemplo clássico é somar  $0.1$  com  $0.2$ , que resulta em  $0.30000000000000004$ .
- Isto deve-se à norma IEEE 754, usada para representar números reais, mas que não será discutida nestes materiais.

- 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.

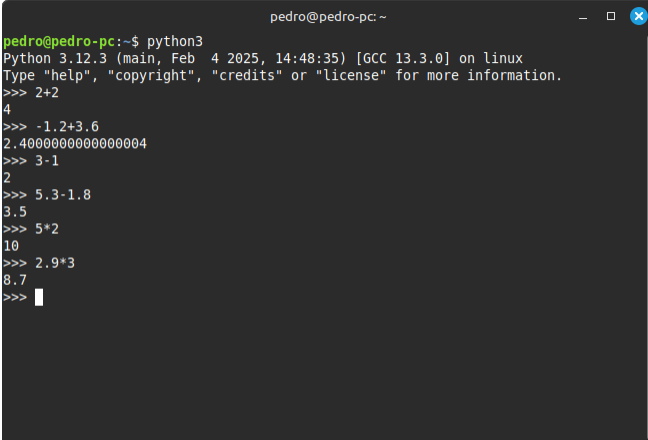
## Expressões aritméticas

---

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

- $+$  representa a soma. Ex:
  - $2 + 2$ , que é igual a 4
  - $-1.2 + 3.6$ , que é igual a 2.4000000000000004
- $-$  representa a subtração. Ex:
  - $3 - 1$ , que é igual a 2
  - $5.3 - 1.8$ , que é igual a 3.5
- $*$  representa a multiplicação. Ex:
  - $5 * 2$ , que é igual a 10
  - $2.9 * 3$ , que é igual a 8.7

# Expressões aritméticas



```
pedro@pedro-pc: ~  
pedro@pedro-pc:~$ python3  
Python 3.12.3 (main, Feb  4 2025, 14:48:35) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> 2+2  
4  
>>> -1.2+3.6  
2.4000000000000004  
>>> 3-1  
2  
>>> 5.3-1.8  
3.5  
>>> 5*2  
10  
>>> 2.9*3  
8.7  
>>> 
```

Interpretador de Python, equivalente à janela no canto inferior direito do Spyder

- `/` representa a divisão real. Ex:
  - `4/2`, que é igual a 2.0
  - `5.4/0.25`, que é igual a 21.6
- `//` representa a divisão inteira. Ex:
  - `4//2`, que é igual a 2
  - `5//2`, que é igual a 2
- `%` representa o resto da divisão inteira. Ex:
  - `4%2`, que é igual a 0
  - `5%2`, que é igual a 1
- `**` representa a potência. Ex:
  - `2**4`, que é igual a 16
  - `9**0.5`, que é igual a 3.0

# Expressões aritméticas

```
pedro@pedro-pc: ~  
pedro@pedro-pc:~$ python3  
Python 3.12.3 (main, Feb  4 2025, 14:48:35) [GCC 13.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> 4/2  
2.0  
>>> 5.4/0.25  
21.6  
>>> 4//2  
2  
>>> 5//2  
2  
>>> 4%2  
0  
>>> 5%2  
1  
>>> 2**4  
16  
>>> 9**0.5  
3.0  
>>> █
```

Interpretador de Python

- Tal como na Matemática, a divisão por 0 não está definida em Python e caso seja efetuada irá produzir um **erro**.
- A divisão real produz obrigatoriamente um valor do tipo **float**
- Tanto o resto como a divisão inteira produzem valores do tipo **int**.
- O tipo de dados do resultado da potência **depende** do tipo de dados da base e do expoente, se pelo menos um deles for float o resultado será float, se ambos forem inteiros será inteiro.

O Python por si só oferece as operações referidas anteriormente, no entanto existe uma biblioteca de Matemática que contém muitas outras operações, como logaritmos e a raiz quadrada (alternativa à potência de expoente  $1/2$ ), e também constantes, como o pi ou o número de neper.

Para fazer uso desta biblioteca deve-se inicialmente fazer o *import* e depois usar o prefixo *math.* para aceder aos valores e funções da biblioteca:

```
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.

Escrever expressões aritméticas longas numa só linha torna-se difícil de ler e pode levar a erros. Em vez disso devemos usar variáveis para guardar valores intermédios:

```
import math
```

*Python*

```
raiz = math.sqrt(6**4 + 7*2**3 + 5**2 - 8)
```

```
fracao = 1/(math.log(9**2 + 1))
```

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

Nota: tanto a escolha de nomes como o número de variáveis e os seus valores são apenas simbólicas. Era perfeitamente aceitável haver mais variáveis intermédias, por exemplo para os termos dentro da raiz quadrada.

- A definição da palavra variável difere na matemática e na informática.
- Na matemática é um símbolo usado para descrever um valor ainda desconhecido ou que pode assumir múltiplos valores.
- Na informática é um símbolo usado para referir um valor guardado na memória do computador, que, por norma, pode ser alterado durante a execução do programa.

# 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!

No entanto, isto introduz alguns problemas:

- A fórmula da distância é relativamente simples, mas copiar pedaços de código mais complexos e com múltiplas linhas torna o código difícil de ler.
- Se detetarmos um erro numa fórmula 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))
```



- 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.

## Condicionalis

---

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.

- 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  $m[i][j]$ .