A instrução if-elif-else

```
number = 23
guess = int(input('Enter an integer : '))

if guess == number:
    # New block starts here
    print('Congratulations, you guessed it.')
    print('(but you do not win any prizes!)')
    # New block ends here
elif guess < number:
    # Another block
    print('No, it is a little higher than that')
    # You can do whatever you want in a block ...
else:
    print('No, it is a little lower than that')
    # you must have guessed > number to reach here

print('Done')
# This last statement is always executed,
# after the if statement is executed.
```

Repare-se na importância da indentação

Saída:

```
Enter an integer : 50
No, it is a little lower than that
Done

Enter an integer : 22
No, it is a little higher than that
Done

Enter an integer : 23
Congratulations, you guessed it.
(but you do not win any prizes!)
Done
```

Quando uma estrutura de controle afeta uma única instrução, essa instrução pode ficar na mesma linha

Exemplo: else: print('No, it is a little lower...')

O ciclo While

```
number = 23
running = True

while running:
    guess = int(input('Enter an integer : '))

    if guess == number:
        print('Congratulations, you guessed it.')
        # this causes the while loop to stop
        running = False
    elif guess < number:
        print('No, it is a little higher than that.')
    else:
        print('No, it is a little lower than that.')
else:
    print('The while loop is over.')
    # Do anything else you want to do here

print('Done')
```

- Repare-se na cláusula opcional 'else' do While.
  - *essa parte é executada logo que a condição do While se torne falsa, (i. e., quando se sai do ciclo)*
  - *só não é executada quando se sai do ciclo com um break.*

Saída:

```
Enter an integer : 50
No, it is a little lower than that.
Enter an integer : 22
No, it is a little higher than that.
Enter an integer : 23
Congratulations, you guessed it.
The while loop is over.
Done
```

O ciclo For

```
for i in range(1, 5):
    print(i)
else:
    print('The for loop is over')
```

Saída:

```
1
2
3
4
The for loop is over
```

- O For itera sobre uma sequência de valores (objetos)
  - *neste caso essa sequência é gerada pela função do Python range()*
    - repare-se que a sequência termina no elemento anterior ao último
    - se se adicionar um terceiro parâmetro na função range(), o incremente deixa de ser unitário, para assumir esse valor
  - *funciona de forma semelhante ao for-each do Java*
- O For contém também a cláusula opcional 'else'
  - *é executada logo que termine a iteração*
  - *só não é executada quando o ciclo termina com um break.*

Definição de funções

- As funções são definidas usando a palavra-chave def.

```
def say_hello():
    # block belonging to the function
    print('hello world')
# End of function

say_hello()  # call the function
say_hello()  # call the function again
```

```
def print_max(a, b):
    if a > b:
        print(a, 'is maximum')
    elif a == b:
        print(a, 'is equal to', b)
    else:
        print(b, 'is maximum')
```

- Os parâmetros das funções em Python são passados por "referência de objeto"
  - *isso significa que é passada uma cópia da referência do objeto usado na invocação*
  - *logo, qualquer alteração, dentro da função, do objeto referenciado, reflete-se para fora da função*
    - estando dessa forma assegurado o comportamento de saída aos parâmetros das funções, quando necessário.

Parâmetros Opcionais

```
def say(message, times=1):
    print(message * times)

say('Hello')
say('World', 5)
```

- Os parâmetros opcionais têm sempre de ser os últimos
- Repare qual o efeito de se multiplicar uma string por um inteiro

Saída:

```
Hello
WorldWorldWorldWorldWorld
```

Passagem de valores aos parâmetros pelo respetivo nome

```
def func(a, b=5, c=10):
    print('a is', a, 'and b is', b, 'and c is', c)

func(3, 7)
func(25, c=24)
func(c=50, a=100)
```

Permite-nos

- indicar os parâmetros de invocação não respeitando a sua ordem
- e deixar sem valor parâmetros opcionais anteriores

Funcionalidade que dá muito jeito quando invocamos uma função com imensos parâmetros opcionais, mas só usamos alguns deles.

Saida:

```
a is 3 and b is 7 and c is 10
a is 25 and b is 5 and c is 24
a is 100 and b is 5 and c is 50
```

Número indeterminado de parâmetros

```
def total(a=5, *numbers, **phonebook):
    print('a', a)

    #iterate through all the items in tuple
    for single_item in numbers:
        print('single_item', single_item)

    #iterate through all the items in dictionary
    for first_part, second_part in phonebook.items():
        print(first_part,second_part)

total(10,1,2,3,Jack=1123,John=2231,Inge=1560)
```

Output:

```
a 10
single_item 1
single_item 2
single_item 3
Jack 1123
John 2231
Inge 1560
```

- Quando declaramos um parâmetro prefixado com um asterisco, tal como *numbers, então todos os argumentos desde essa posição são guardados num tuplo, neste caso chamado 'numbers'.
- De maneira similar, quando declaramos um parâmetro prefixado com dois asteriscos, tal como **phonebook, então todos os argumentos do tipo 'chave=valor' desde essa posição serão guardados num dicionário, neste exemplo chamado phonebook ‘.
  - *Só a partir do Python 3.7, é que os dicionários passaram a preservar a ordem de inserção*

Função com valor de retorno

```
def maximum(x, y):
    if x > y:
        return x
    elif x == y:
        return 'The numbers are equal'
    else:
        return y

print(maximum(2, 3))
```

Output:

```
3
```

- Se a função não terminar com uma instrução return, a mesma devolve um resultado nulo, representado pela palavra reservada None
- None
  - *representa uma variável ou objeto nulo, ou simplesmente a ausência de valor*
  - *mais ou menos equivalente ao null de outras linguagens*
- Duas formas de verificar se um valor é nulo:
  - *valor is None*
  - *valor == None*

Exemplo de função com valor de retorno e com um número indeterminado de parâmetros

```
>>> def powersum(power, *args):
...     '''Return the sum of each argument raised to the specified power.'''
...     total = 0
...     for i in args:
...         total += pow(i, power)
...     return total
...
>>> powersum(2, 3, 4)
25
>>> powersum(2, 10)
100
```

DocStrings

- O Python possui um recurso chamado strings de documentação, também designadas simplesmente por docstrings.
  - *É uma importante ferramenta que ajuda a documentar o código*
  - *Pode ser usado em funções, classes, módulos e métodos*
  - *Trata-se da 1ª linha lógica que surge na definição do objeto em causa*
  - *Tipicamente, é formada por múltiplas linhas de texto, em que a primeira começa com uma letra maiúscula e termina com um ponto, a 2ª linha é deixada em branco, e a partir da 3ª deve surgir uma descrição mais detalhada da função.*

```
def print_max(x, y):
    '''Prints the maximum of two numbers.

    The two values must be integers.'''
    # convert to integers, if possible
    x = int(x)
    y = int(y)

    if x > y:
        print(x, 'is maximum')
    else:
        print(y, 'is maximum')

print_max(3, 5)
print(print_max.__doc__)
```

- É possível aceder à docstring através do atributo `__doc__` da função

  (recorde-se que o Python trata tudo como objetos, incluindo as funções)

- É também isso que faz o help() do Python
  - *exibe o docstring da função invocando de forma implícita o atributo `__doc__`*

    `help(print_max)`

- Existem várias ferramentas automáticas que geram a documentação desta maneira.

Output:

```
$ python function_docstring.py
5 is maximum
Prints the maximum of two numbers.

    The two values must be integers.
```