A BRIEF INTRODUCTION TO PYTHON

(With content based on the book “A Byte of Python”, by Swaroop [3])

The if-elif-else statement

```
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.
```

Note the importance of indentation

When a control structure affects a single statement, that statement can be on the same line

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

Saída:

```
$ python if.py
Enter an integer : 50
No, it is a little lower than that
Done

$ python if.py
Enter an integer : 22
No, it is a little higher than that
Done

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

The While cycle

```
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')
```

- Notice the optional 'else' clause of While.
  - *this part is executed as soon as the While condition becomes False, (i.e., when you exit the loop)*
  - *just doesn't run when you go out of the loop with the 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
```

The For cycle

```
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
```

- For iterates over a sequence of values (objects)
  - *in this case this sequence is generated by the function of the Python range()*
    - note that the sequence ends in the element before the last
    - if you add a third parameter to the range() function, the increment leaves its unit, to assume this value
  - *works similarly to Java for-each*
- For also contains the optional clause 'else'
  - *is executed as soon as the iteration ends*
  - *just doesn't run when the cycle ends with a break.*

Functions

- Functions are defined using the keyword 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')
```

- The parameters of python functions are passed by "object reference"
  - *this means that a copy of the object reference used in the invocation is passed*
  - *therefore, any change, within the function, of the referenced object, is reflected outside the function*
    - thus ensuring the output behavior to the parameters of the functions, when necessary

Optional Parameters

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

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

- Optional parameters must always be the last
- Notice what effect of multiplying a string by an integer

Saída:

```
Hello
WorldWorldWorldWorldWorld
```

Assigning values to parameters by their name

```
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)
```

It allows us to

- indicate the invocation parameters not respecting their order
- and leave no value previous optional parameters
- *Functionality that comes very handy when we invoke a function with many optional parameters, but we only use some of them.*

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
```

Undetermined number of parameters

```
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
Inge 1560
John 2231
Jack 1123
```

- When we declare a parameter pinned with an asterisk, such as *numbers, then all arguments from that position are saved in a tuple called 'numbers'.
- Simply put, when we declare a parameter fixed with two asterisks, such as **phonebook, then all arguments of type 'key=value' since that position will be saved in a dictionary called 'phonebook'.

Function with return value

```
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
```

- If the function does not end with a return statement, it returns a null result, represented by the reserved word None
- None
  - *represents a variable or null object, or simply the absence of value*
  - *equivalent to the null of other languages*
- Two ways to verify that a value is null:
  - *Value is None*
  - *Value == None*

Example of function with return value and indeterminate number of parameters

```
>>> 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

- Python has a feature called documentation strings, also called simply docstrings.
  - *It is an important tool that helps document the code*
  - *This is the first logical line that arises in the definition of the object in question*
  - *Typically, it is formed by multiple lines of text, in which the first begins with a capital letter and ends with a dot, the 2nd line is left blank, and from the 3rd should appear a more detailed description of the function.*

```
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__)
```

- You can access the docstring through the __doc__da attribute

  (remember that Python treats everything as objects, including functions)

- This is also what python help() does
  - *displays the docstring of the function implicitly invoking the attribute __doc__*

```
help(print_max)
```

- There are several automatic tools that generate documentation this way.

Output:

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

    The two values must be integers.
```

solve exercise #2

from the book of exercises