Variables

With the command `a=3` we store in the computer memory the value of 3, and we can later refer to this value by using the variable name `a` .

In [1]:
```
a=3
```

In [2]:
```
a
```

Out[2]:
```
3
```

In [3]:
```
a+a
```

Out[3]:
```
6
```

In [4]:
```
a = 'mitsos'
```

In [5]:
```
a
```

Out[5]:
```
'mitsos'
```

We can apply operations between variables in the same way we did with constants:

In [6]:
```
a=3
b=4
print (a+b)
```

```
7
```

The spaces do not matter (as long as all the lines start from the same space on the left).

All of the following are equivalent:

In [8]:
```
a=3
a = 3
a                              =
```

But in the following there is a mistake!

In [9]:
```
a=3
 a=3 # Αυτό ξεκινάει με ένα κενό πιο μέσα!
```

```
  File "<ipython-input-9-fc87d32c6889>", line 2
    a=3 # Αυτό ξεκινάει με ένα κενό πιο μέσα!
    ^
IndentationError: unexpected indent
```

With the `print` command we can.. print the values of one or more variables:

In [10]:
```
a="the answer is"
b=7
```

In [11]:
```
print (a,b)
```

```
the answer is 7
```

In [12]:
```
print ("this answer is", b)
```

```
this answer is 7
```

We can add a variable inside a string by adding `{}` inside the string. To declare which variable should be printed, we use the `format` function.

```
In [13]: c = "answer is {}".format(b)
         print (c)
```

```
answer is 7
```

We can have more than one `{}` in a string:

```
In [14]: a = 'James'
         b = 'Bond'
         print ('My name is {}. {} {}.'.format(b, a, b))
```

```
My name is Bond. James Bond.
```

Pay attention to the difference between `=` and `==` :

```
In [6]: a = 3 # We assign the value 3 to the variable a
        a == 3 # Check if the variable a is equal to 3
```

```
Out[6]: True
```

The operation: +=

Suppose we have a variable that has the value 3:

```
In [15]: a = 3
         print (a)
```

```
3
```

How can we increase its value by 1?

```
In [16]: a = a + 1
         print (a)
```

```
4
```

The command `a=a+1` is used very often (in fact every time we "measure" something). So we can write it as: `a += 1` . Similarly we can write `a += 4`

```
In [17]: a += 4 # a = a + 4
         print (a)
```

```
8
```

The same can be done with all other operations. E.g. `a -= 1` is equivalent to `a = a - 1` .

```
In [18]: a -= 1
         print (a)
```

```
7
```

Functions

Functions are a huge part of computer theory. Through functions we can "break" our code into small functional parts and make it more modular and more reusable. E.g. a function that calculates the average can be used in many parts of our code. In python we define a function through def :

```
In [42]: def f():
             print ('hello')

         f()
```

```
hello
```

A function can "return" a value to the caller:

```
In [43]: def f():
             print ('hello')
             return 4
         a=f()
         print (a)
```

```
hello
4
```

A function can take zero, one or more parameters:

```
In [44]: def f(t):
             return t+1
         a=f(3)
         print (a)
```

```
4
```

```
In [45]: def f(q,w,e,r,t):
             return q+w-e/r*t
         a=f(2,3,4,5,6)
         print (a)
```

```
0.1999999999999993
```

Also a function can have parameters with predefined values. If the function is called without giving a value to these parameters then the default value is used:

```
In [46]: def f(a,b=4):
             return a+b
```

```
In [47]: f(2,3)
```

```
Out[47]: 5
```

```
In [48]: f(2)
```

```
Out[48]: 6
```

A function can have many parameters with predefined values:

```
In [49]: def f(a,b=2,c=4):
             return a+b+C
```

However, all these parameters must be declared after the parameters without predefined values:

```
In [50]: def f(a,b=2,c):
             return 42
```

```
  File "<ipython-input-50-7943a91cece0>", line 1
    def f(a,b=2,c):
```

```
^
```

Caution! when we change an argument of the function, then if that argument is string, int, float or bool (these types are called primitive) then this change does not appear from where we called the function:

In [53]:
```
def f(a):
    a = a + 1 # We change a !

a=4
f(a)
print (a) # a here did not change!
```

```
4
```

Functions cannot "see" the primitive data types (int, string, bool) defined outside of them:

In [54]:
```
a=4
def f():
    a=5
f()
print (a)
```

```
4
```

To be able to "see" a function a primitive data type that is defined outside of this function, we can use the word: `global` :

Caution: Some parts of the python programming community believe that we should never use global . If your are an amateur programmer then you might find global variables useful. But as you get more experienced you will realize that passing values between functions and/or using classes is a far better, more efficient and more "organizing" method to "pass around" your data between functions.

In [55]:
```
a=4
def f():
    global a
    a=5
f()
print (a)
```

```
5
```

Caution! anything "below" the `return` is ignored:

In [56]:
```
def f():
    print ("hello")
    return 5
    print ("dsfgsdfg")

f()
```

```
hello
```

Out[56]:
```
5
```

A function that returns nothing returns a value that is `None` .

```
In [57]: def f():
             print ("hello")

         print (f())
```

```
hello
None
```

`None` is a new type of data:

```
In [58]: type(None)
```

```
Out[58]: NoneType
```

A function may contain another function:

```
In [59]: def f(r):
             def g():
                 return r + 5
             return g() + 3

         f(1)
```

```
Out[59]: 9
```

Functions are also `function` variables:

```
In [60]: type(f)
```

```
Out[60]: function
```

We can check if a variable is a function by using the `callable` function:

```
In [61]: callable(f)
```

```
Out[61]: True
```

The `if` syntax

The `if` syntax is a beautiful way to declare that a set of commands will be executed if (and only if) an expression is `True` or `False` .

```
In [20]: if True:
             print ("Hello")
```

```
Hello
```

```
In [7]: if False:
            print ("Hello") # <-- Does not get executed
```

```
In [22]: if 1<3:
             print ("Hello")
```

```
Hello
```

```
In [8]: if 3<1 in [1,2]:
            print ("Hello") # <-- Does not get executed
```

Caution! all strings except empty are `True` :

```
In [24]: if 'mitsos':
             print ("hello")
```

```
hello
```

```
In [9]: if '':
            print ("hello") # <- Does not get executed
```

All numbers except 0 are `True`

```
In [26]: if 3453:
             print ("Hello")
```

```
Hello
```

```
In [27]: if 0: # This is False
             print ("Hello")
```

```
In [10]: if 0.000000000001: # Thisis True !!
             print ("Hello")
```

```
Hello
```

This is not allowed:

```
In [30]: if a = 3:
             print ("Hello")
```

```
  File "<ipython-input-30-9d66b7bd4d9a>", line 1
    if a = 3:
         ^
SyntaxError: invalid syntax
```

This is allowed:

```
In [31]: if a == 3:
             print ("Hello")
```

```
Hello
```

An if command may have "inside" other if commands..

```
In [32]: print ('1')
         if True:
             print ('hello')
             if True:
                 print ('Kostas')

         print ('2')
```

```
1
hello
Kostas
2
```

If for some reason we do not want an `if` command to contain any command, we can use the `pass` command, which does absolute nothing.

```
In [33]: print ('1')
         if True:
             pass
         print ('2')
```

```
1
```

We can declare what we want to happen when the if condition is NOT true with the `else` syntax:

In [1]:
```
print ('1')
if True:
    print ("hello") # <-- It enters here
else:
    print ('world') # <-- It does not enter here
print ('2')
```

```
1
hello
2
```

In [2]:
```
print ('1')
if False:
    print ("hello") # <-- It does not enter here
else:
    print ('world') # <-- It enters here
print ('2')
```

```
1
world
2
```

We can also have many conditions with `elif`. Python checks them one by one and once (and if) finds the first `True`, then it executes the relevant commands inside the indentation:

In [36]:
```
print ('hello')
a=2
if a==1:
    print ('1')
elif a==2:
    print ('2')
else:
    print ('3')
print ('world')
```

```
hello
2
world
```

It is not necessary every if to have an `else` part:

In [37]:
```
print ('hello')
a=3
if a==1:
    print ('1')
elif a==2:
    print ('2')
print ('world')
```

```
hello
world
```

In `if` we can declare more than one condition or use `else` to declare what to do if all the conditions in `if` and `elif` are `False`

In [3]:
```
age = 23
if age<18:
    status = 'minor'
else:
    status = 'adult'

print (status)
```

```
adult
```

Note that the above does not check for the prossiblity of an error:

In [39]:
```
age = -4

if age<18:
    status = 'ανήλικος'
else:
    status = 'ενήλικος'

print (status)
```

```
ανήλικος
```

By having more than one `elif` we can check for many possibilities:

In [4]:
```
age = 50
if age <0:
    status = "Error. Negative value"
elif age < 18:
    status = "minor"
elif age < 120:
    status = "adult"
else:
    status = "Error. Value is too high"
print (status)
```

```
adult
```

Try the above for different `age` values.

Also, it is not necessary to use `else` :

In [5]:
```
age = 150
if age <0:
    status = "Error. Negative value"
elif age < 18:
    status = "minor"
elif age < 120:
    status = "adult"
elif age >= 120:
    status = "Error. Value is too high"
print (status)
```

```
Error. Value is too high
```

In [ ]:
```

```