Introduction to programming with python

Alexandros Kanterakis kantale@ics.forth.gr

Introduction

All lectures will be available in the form of jupyter notebooks . Jupyter is an environment that allows us to write python and inspect the results of commands directly in your browser. You can save your experiments in a file and share it by mail, etc.

A jupyter notebook consists of cells. Each cell can contain either python code (other languages are also allowed) or markdown . Markdown is a collection of conventions to import formatting into a text file. E.g. if in markdown we write a word between 2 asterisks (eg: `**Alexandros**`) then it will appear as bold, that is: **Alexander** . Complete list of all markdown contracts .

You can also load a notebook in your browser, the same way that you open word document. Even better you can save a notebook on the Internet for free! As a gist .

A first taste

print

```
In [6]: print ('something')
```

```
something
```

We can use single ('), double quotes (") or triple quotes (''') or ("""), to denote the start and the end of a string.

```
In [1]: print ("hello")
```

```
hello
```

```
In [9]: print ("""hello""")
```

```
hello
```

```
In [10]: print ('''hello''')
```

```
hello
```

We can have an "Enter" inside a string with the special character: "\n". `n` stands for "new line".

```
In [4]: print ("hello\nworld")
```

```
hello
world
```

Similarly we can use single or double quotes in a string. Depending on what we use to declare a string (single or double quotes) we should use `'` or `''` :

```
In [5]: print ("his master's voice")
```

```
his master's voice
```

```
In [6]: print ('his master\'s voice')
```

```
his master's voice
```

```
In [7]: print (" I am \"fear\"  ")
```

```
 I am "fear"
```

```
In [8]: print ('i am "fear"')
```

```
i am "fear"
```

If we use triple quotes we can have many lines in one string (multiline strings):

```
In [12]: print ("""ἄνδρα μοι ἔννεπε, μοῦσα, πολύτροπον, ὃς μάλα πολλὰ
         πλάγχθη, ἐπεὶ Τροίης ἱερὸν πτολίεθρον ἔπερσεν·
         πολλῶν δ᾽ ἀνθρώπων ἴδεν ἄστεα καὶ νόον ἔγνω,
         πολλὰ δ᾽ ὅ γ᾽ ἐν πόντῳ πάθεν ἄλγεα ὃν κατὰ θυμόν,
         ἀρνύμενος ἥν τε ψυχὴν καὶ νόστον ἑταίρων.""")
```

```
ἄνδρα μοι ἔννεπε, μοῦσα, πολύτροπον, ὃς μάλα πολλὰ
πλάγχθη, ἐπεὶ Τροίης ἱερὸν πτολίεθρον ἔπερσεν·
πολλῶν δ᾽ ἀνθρώπων ἴδεν ἄστεα καὶ νόον ἔγνω,
πολλὰ δ᾽ ὅ γ᾽ ἐν πόντῳ πάθεν ἄλγεα ὃν κατὰ θυμόν,
ἀρνύμενος ἥν τε ψυχὴν καὶ νόστον ἑταίρων.
```

(we will come back later)

Comments

In any line, anything that follows the character `#` is considered a comment and is ignored:

```
In [1]: # This is a comment
        print ('This is not a comment') # But this is!
```

```
This is not a comment
```

Mathematical operations and expressions:

Python can do operations with integers with any number of digits:

```
In [13]: 24328470239847502934672098347520349867 * 234573458729835619384756398456
```

```
Out[13]: 5706813409766902302233897564852365380972748955782671565995978605352
```

We have the classic operations: addition, subtraction, multiplication and division:

```
In [15]: 3+2
```

```
Out[15]: 5
```

```
In [16]: 3-2
```

```
Out[16]: 1
```

```
In [17]: 3*2
```

```
Out[17]: 6
```

Decimal division:

```
In [12]: 3/2
```

```
Out[12]: 1.5
```

Integer division:

```
In [13]: 3//2
```

```
Out[13]: 1
```

Caution!

```
In [18]: 1/0
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-18-9e1622b385b6> in <module>()
----> 1 1/0

ZeroDivisionError: division by zero
```

We also have some additional operations:

The remainder of the division:

```
In [19]: 15 % 4
```

```
Out[19]: 3
```

That is 15 = 3 * 4 + **3**

The exponential:

```
In [20]: 4**2
```

```
Out[20]: 16
```

Attention : The exponential is NOT `^` :

```
In [21]: 4 ^ 2
```

```
Out[21]: 6
```

`^` Is another operation called XOR and will not concern us in this lesson.

In python every operation has a priority. For example, multiplications and divisions are performed **before** additions and subtractions:

```
In [22]: 10+6/2
```

```
Out[22]: 13.0
```

Note: It is better not to rely on the priority of actions since it might not be that obvious. To clearly set the priority of an operation we use parenthesis:

```
In [23]: 10+(6/2)
```

```
Out[23]: 13.0
```

```
In [24]: (10+6)/2
```

```
Out[24]: 8.0
```

If an operation anywhere has a division **or** a decimal number, the result is decimal, otherwise it is an integer:

```
In [25]: 2/2
```

```
Out[25]: 1.0
```

```
In [26]: 2*2
```

```
Out[26]: 4
```

```
In [27]: 2 + 2.0
```

```
Out[27]: 4.0
```

```
In [48]: 2+2
```

```
Out[48]: 4
```

Python has one:

- Synonym for 1: `True`
- Synonym for 0: `False`

```
In [49]: True + 1
```

```
Out[49]: 2
```

```
In [50]: False + 1
```

```
Out[50]: 1
```

We will see more about `False` and `True` a bit later!

Although integers can have an unlimited number of digits, decimal numbers have a certain accuracy:

```
In [156… 5.123456789012345678901234567
```

```
Out[156… 5.123456789012345
```

Why is this happening? An integer, no matter how large, can be represented in memory without losing accuracy. But for some decimals it is simply impossible to have infinite accuracy. For example:

```
In [157… 1/3
```

```
Out[157… 0.3333333333333333
```

`1/3` has infinite decimal places! How do we store this in memory? The solution is to store a certain number of decimal places. Fortunately, there is an international IEEE-754 standard that defines how we store decimal numbers. However, you can instruct python not to use this template and handle decimals as accurately as you want (sacrificing memory and computing speed a bit).

Alphanumeric (or otherwise: strings)

```
In [14]: "mitsos"
```

```
Out[14]: 'mitsos'
```

We can add two strings:

```
In [15]: 'a' + 'b'
```

```
Out[15]: 'ab'
```

We can multiply a string by an integer:

```
In [16]: 'a' * 10
```

```
Out[16]: 'aaaaaaaaaa'
```

There is also the empty string

```
In [238… ''
```

```
Out[238… ''
```

`len` returns the size of a string

```
In [228… len("abcdefg")
```

```
Out[228… 7
```

```
In [239… len('')
```

```
Out[239… 0
```

`Count` returns how many times there is a string inside another string.

```
In [230… "zabarakatranemia".count('a')
```

```
Out[230… 6
```

```
In [231… "zabarakatranemia".count('ra')
```

```
Out[231… 2
```

```
In [232… "zabarakatranemia".count('c')
```

```
Out[232… 0
```

`index` returns the index of the first occurence of a sub-string in a string.

```
In [234… "zabarakatranemia".index('anemia')
```

```
Out[234… 10
```

```
In [2]:
# "ra" exists twice but index always returns the index
# of the first occurence.
"zabarakatranemia".index('ra')
```

```
Out[2]: 4
```

If it does not exist then it raises an error!

```
In [237…
"zabarakatranemia".index('c')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-237-1515cc1d7dbe> in <module>()
----> 1 "zabarakatranemia".index('c')

ValueError: substring not found
```

Caution! Two strings declared next to each other are considered one!

```
In [271…
"Hello" "world"
```

```
Out[271… 'Helloworld'
```

A string can have characters in any language!

```
In [29]:
a = "σε οποιαδίποτε γλώσσα ذا كيور (بالإنجليزية: The Cure) 19. واجهت الفرقة عِدة تغيرات؛ مع
print (a)
```

```
σε οποιαδίποτε γλώσσα ذا كيور (بالإنجليزية: The Cure) هي فرقة روك إنجليزية، تم تكوينها في كرول
ي، غرب ساسكس عام 1976. واجهت الفرقة عِدة تغيرات؛ مع
```

Yes, emoji are included:

```
In [277…
print ("\U0001F621")
```

```
😡
```

After this very brief introduction in python, we can talk in a more theoretical perspective:

Operators

Operators are symbols or reserved words with which we apply basic operations to various expressions. For more you can read here: https://en.wikipedia.org/wiki/Operator_(computer_programming)

Some of the most basic operators that python supports are:

- +
- -
- /
- //
- *
- %

- <
- >
- <=
- > =
- ! =
- ==
- and
- or
- not
- in

The + operator

The expressions that can include the `+` operator are:

```
In [32]: 3+2
Out[32]: 5

In [33]: 3+2.0
Out[33]: 5.0

In [34]: 3+0.0
Out[34]: 3.0

In [35]: 'ab' + 'cde'
Out[35]: 'abcde'

In [36]: True + True + False
Out[36]: 2

In [37]: True + 2
Out[37]: 3

In [38]: True + 0.0
Out[38]: 1.0

In [3]: [1,2,3] + [4,5,6] # Lists, we will talk later about them!
Out[3]: [1, 2, 3, 4, 5, 6]
```

The operator -

The operations allowed by the '-' operator are:

```
In [40]: 3-2
```

```
Out[40]: 1
```

```
In [41]: 3-7
```

```
Out[41]: -4
```

```
In [42]: 4-6.0
```

```
Out[42]: -2.0
```

```
In [43]: True - True
```

```
Out[43]: 0
```

```
In [44]: True - 6.6
```

```
Out[44]: -5.6
```

The operator *

The operations allowed with the '*' operator are:

```
In [45]: 6*7
```

```
Out[45]: 42
```

```
In [46]: 6.6*2
```

```
Out[46]: 13.2
```

```
In [47]: True * 2
```

```
Out[47]: 2
```

```
In [51]: True * False
```

```
Out[51]: 0
```

```
In [52]: True * 2.3
```

```
Out[52]: 2.3
```

```
In [53]: 6 * 'hello'
```

```
Out[53]: 'hellohellohellohellohellohello'
```

```
In [55]: [1,2,3] * 5  # Λίστες, θα τα δούμε αργότερα..
```

```
Out[55]: [1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
```

The '/' operator

Οι πράξεις που επιτρέπονται με τον τελεστή '/' είναι:

```
In [56]: 4/5
```

```
Out[56]: 0.8
```

CAUTION!!

```
In [57]: 5/0
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-57-0106664d39e8> in <module>()
----> 1 5/0

ZeroDivisionError: division by zero
```

```
In [58]: True/True

Out[58]: 1.0
```

```
In [59]: 6/3

Out[59]: 2.0
```

```
In [60]: 6.5/3

Out[60]: 2.1666666666666665
```

The operator '//'

This operator gives us the result of integer division

```
In [62]: 5//2

Out[62]: 2
```

```
In [63]: 11//3

Out[63]: 3
```

```
In [64]: 6.5 // 2.1

Out[64]: 3.0
```

```
In [65]: True // 2

Out[65]: 0
```

The `%` operator

This operator gives us the remainder of the integer division

```
In [66]: 5%2

Out[66]: 1
```

```
In [67]: 5.2 % 2

Out[67]: 1.2000000000000002
```

In [68]:

```
True % 2
```

Out[68]:

```
1
```

The `%` operator is used (not so often) to insert strings inside strings

In [70]:

```
'my name is %s nice to meet you' % "mitsos"
```

Out[70]:

```
'my name is mitsos nice to meet you'
```

You can find here how you can use this operator for strings with more examples

The operator **

This operator returns the exponential a^b

In [71]:

```
3**2
```

Out[71]:

```
9
```

In [72]:

```
3.2**2.3
```

Out[72]:

```
14.515932837559118
```

In [73]:

```
True ** 2
```

Out[73]:

```
1
```

Logical Operators

We saw before the constants `True` and `False` . What is their purpose? So far we have seen operators who can generate numbers. For example the `+` operator can produce any number. But there is a group of operators that can only produce 2 different values. These values are `True` and `False` . These operators are:

- The comparison operators < , > , <= , >=
- The equality operators `==` , `!=`
- The operators `and` , `or` , `not`
- The `in` , `is` operators (we will talk about them later)

  These operators **always** return True or False to **whatever** we apply them!

  a < β checks if α is less than β:

In [74]:

```
2<3
```

Out[74]:

```
True
```

In [76]:

```
3<2
```

Out[76]:

```
False
```

In [77]:

```
3<3
```

```
Out[77]: False
```

α <= β checks if α is less than or equal to β:

```
In [78]: 2<=3
```

```
Out[78]: True
```

```
In [79]: 3<=2
```

```
Out[79]: False
```

```
In [80]: 3<=3
```

```
Out[80]: True
```

α > β checks if α is greater than β:

```
In [82]: 3 > 2
```

```
Out[82]: True
```

```
In [85]: 2 > 3
```

```
Out[85]: False
```

```
In [86]: 3 > 3
```

```
Out[86]: False
```

α >= β checks if α is greater than or equal to β:

```
In [87]: 3 >= 2
```

```
Out[87]: True
```

```
In [88]: 2 >= 3
```

```
Out[88]: False
```

```
In [89]: 3 >= 3
```

```
Out[89]: True
```

The operator α == β checks if α is equal to β

```
In [90]: 3==2
```

```
Out[90]: False
```

```
In [91]: 3==3
```

```
Out[91]: True
```

```
In [92]: 3==3.0
```

```
Out[92]: True
```

```
In [93]: "3" == 3
Out[93]: False
```

```
In [114… 16**0.5 == 4
Out[114… True
```

```
In [115… 'mitsos' == 'mits' + 'os'
Out[115… True
```

```
In [116… 3 == 6/2
Out[116… True
```

```
In [117… True == True or False
Out[117… True
```

```
In [118… 3 == True + True + True + False
Out[118… True
```

```
In [119… 3 == 'mits' + 'os'
Out[119… False
```

```
In [121… [1,2,3] == [1,2,3] # Λίστες, θα τα δούμε αργότερα
Out[121… True
```

```
In [122… [1,2,3] == [2,1,3]
Out[122… False
```

The operator α != β if α is **not** equal to β

```
In [95]: 3 != 2
Out[95]: True
```

```
In [108… 2 != 2
Out[108… False
```

```
In [109… 2 != 2.0
Out[109… False
```

```
In [110… 1 != True
Out[110… False
```

```
In [111… 'hello' != ' hello '
Out[111… True
```

```
In [112…  '' != ''
Out[112…  False
```

```
In [113…  '' != ' '
Out[113…  True
```

We can also use the same operators `<` , `>` , `<=` , `>=` more than once:

```
In [97]:  2<3<4
Out[97]:  True
```

```
In [98]:  2<3<3
Out[98]:  False
```

When we apply these operators to strings, then we compare them alhabetical. "Smaller" is considered the one that ib an alphabetical ordering has the lower index.

Your phone uses exactly this method to sort your contacts!

```
In [99]:  'ab' < 'fg'
Out[99]:  True
```

```
In [100…  'ab' < 'b'
Out[100…  True
```

```
In [101…  'ab' < 'ac'
Out[101…  True
```

```
In [102…  'ab' < 'a'
Out[102…  False
```

The empty string has the lowest possible value

```
In [103…  '' < '0'
Out[103…  True
```

```
In [104…  "A" < "a"
Out[104…  True
```

```
In [105…  "05456745674" < "5"
Out[105…  True
```

```
In [106…  '8' < '09'
Out[106…  False
```

Sometimes we want to make a decision depending on whether 2 or more reasonable values are true. For example:

I will go out if the weather is good **and** I have free time.

I will go out if the weather is good **or** I have free time.

So we have two additional logical operators: `and` , `or` .

A and β are `True` if α and β are `True` , if one of α, β is `False` (or both), then the result is `False` :

In [123…
```
True and True
```
Out[123… True

In [124…
```
True and False
```
Out[124… False

In [125…
```
False and True
```
Out[125… False

In [126…
```
False and False
```
Out[126… False

In [127…
```
(1==1) and (2==3-1)
```
Out[127… True

In [128…
```
(1==1) and (2==3)
```
Out[128… False

In [129…
```
(1>=1) and (2<=2)
```
Out[129… True

Similarly the result of the operation α or β is `True` if one of α, β (or both) is `True` , otherwise it is `False` :

In [130…
```
True or True
```
Out[130… True

In [131…
```
True or False
```
Out[131… True

In [132…
```
False or True
```
Out[132… True

In [133…
```
False or False
```

```
Out[133… False
```

```
In [134… 1==2 or 1<=1
```

```
Out[134… True
```

```
In [136… 1>2 or 2<1
```

```
Out[136… False
```

```
In [137… 0==1 or True
```

```
Out[137… True
```

Finally, there is the `not` operator. This operator has the peculiarity that it is applied to a single value. `not` a, results in `False` if a is `True` and `True` if a is `False` :

```
In [138… not True
```

```
Out[138… False
```

```
In [144… not False
```

```
Out[144… True
```

```
In [151… not 0
```

```
Out[151… True
```

```
In [152… not 0.0000000001
```

```
Out[152… False
```

```
In [146… not ''
```

```
Out[146… True
```

```
In [147… not 1
```

```
Out[147… False
```

```
In [148… not ' '
```

```
Out[148… False
```

```
In [140… not 3==4
```

```
Out[140… True
```

```
In [141… not 3==3
```

```
Out[141… False
```

```
In [142… not "mitsos"=="Mitsos"
```

```
Out[142… True
```

In [149…:
```
not "mitsos" == "mitsos"
```
Out[149…:
```
False
```

Fun fact: Your computer which does all sort of incredible things, in reality it can actually do only one operation: `not (a and b)` . This operation is called NAND. For example when the computer does a mathematical operation (eg `14.2 * 51.1`), it "breaks" that operation into NAND operations. That is, the processor has billions of circuits that do this and that alone. But they are organized so that when combined in the right way they do all the arithmetic operations! More . Even when your computer does something more complex (streaming, playing a game, controlling a nuclear reactor, guiding a spaceship) it still breaks all the operations needed in NAND operations.

Back to strings!

All capitals:

In [32]:
```
"abcde".upper()
```
Out[32]:
```
'ABCDE'
```

All lower:

In [33]:
```
"ABCDE".lower()
```
Out[33]:
```
'abcde'
```

Replace one piece of string with another:

In [173…:
```
"hello world".replace('l', "QQQ")
```
Out[173…:
```
'heQQQQQQo worQQQd'
```

We can remove the empty strings from the end and from the start of a string:

In [178…:
```
"    hello    ".strip()
```
Out[178…:
```
'hello'
```

In [179…:
```
"+++hello+++".strip('+')
```
Out[179…:
```
'hello'
```

In [180…:
```
not "     "
```
Out[180…:
```
False
```

In [182…:
```
not "     ".strip()
```
Out[182…:
```
True
```

Check if one string starts with another string:

In [183…:
```
"heraklio".startswith('her')
```
Out[183…:
```
True
```

Check if one string ends with another string:

```
In [184…  "alex".endswith("lex")
```

```
Out[184…  True
```

Indexing

From strings (as with lists as we will see later), we can extract a subset by using `[]` . This feature is called indexing.

```
In [194…  print ("hello")
```

```
hello
```

Caution! The numbering starts from 0!

```
In [195…  "hello"[0]
```

```
Out[195…  'h'
```

```
In [196…  "hello"[1]
```

```
Out[196…  'e'
```

Caution! The numbering should not exceed the size of the string!

```
In [197…  "hello"[100]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-197-e6ccf1afaf71> in <module>()
----> 1 "hello"[100]

IndexError: string index out of range
```

The index (or "numbering") can get negative values! `-1` is the last element. The `-2` is one before the last etc ..

```
In [199…  "hello"[-1]
```

```
Out[199…  'o'
```

```
In [200…  "hello"[-2]
```

```
Out[200…  'l'
```

```
In [201…  "hello"[-100]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-201-552ff00ad524> in <module>()
----> 1 "hello"[-100]

IndexError: string index out of range
```

Indexing spaces

We can get a subset of a string based on the intervals that we define in `[]`

In [202…

```
"hello"[1:3]
```

Out[202…

```
'el'
```

When we write `[a:b]` we mean "start from the first element (the numbering starts from 0!) And stop at the second element, BUT WITHOUT TAKING THIS !!"

In [204…

```
"hello"[1:4]
```

Out[204…

```
'ell'
```

If we want to get a subset starting from the beginning of the string then we can write either `[0:b]` or `[:b]`

In [206…

```
"hello"[0:2]
```

Out[206…

```
'he'
```

In [207…

```
"hello"[:2]
```

Out[207…

```
'he'
```

If we want a subset that ends at the end of the string then we can write `[a:]`

In [208…

```
"hello"[2:]
```

Out[208…

```
'llo'
```

Indexing spaces with steps

We can use `[a:b:c]` for indexing. This means: go from `a` to `b` (without taking `b` !) with step: `c` .

In [212…

```
"abcdefgij"[1:7:2]
```

Out[212…

```
'bdf'
```

In [213…

```
"abcdefgij"[1:7:3]
```

Out[213…

```
'be'
```

If we omit the first element then by default it assumes the value 0 (the beginning)

In [215…

```
"abcdefgij"[:7:3]
```

Out[215…

```
'adg'
```

If we omit the second then by default it assumes the end of the string

In [216…

```
"abcdefgij"[1::3]
```

Out[216…

```
'bei'
```

We can skip both so it will take from the beginning to the end of the string

In [217…

```
"abcdefgij"[::3]
```

```
Out[217… 'adg'
```

If we omit the third then by default it uses 1

```
In [218… "abcdefgij"[1:7:]
```

```
Out[218… 'bcdefg'
```

Step cannot be 0!

```
In [219… "abcdefgij"[1:7:0]
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-219-96e6dd4da4bc> in <module>()
----> 1 "abcdefgij"[1:7:0]

ValueError: slice step cannot be zero
```

Negative indexing steps.

Step `c` can be negative!

```
In [220… "abcdefgij"[7:1:-1]
```

```
Out[220… 'igfedc'
```

```
In [221… "abcdefgij"[7:1:-2]
```

```
Out[221… 'ifd'
```

```
In [222… "abcdefgij"[7::-2]
```

```
Out[222… 'ifdb'
```

```
In [223… "abcdefgij"[::-2]
```

```
Out[223… 'jgeca'
```

```
In [224… "abcdefgij"[::-1] # Reverse a string!
```

```
Out[224… 'jigfedcba'
```

Useful when we have a cDNA sequence!

```
In [225… "ACGT"[::-1]
```

```
Out[225… 'TGCA'
```

Of course, we can use variables in these indexing spaces:

```
In [45]: a=3
         "abcde"[0:a]
```

```
Out[45]: 'abc'
```

Special Characters

We have said that with single or double quotes we can declare a string. But what happens when we want to include in a string a single or double quote? In that case we can use \ or else backslash:

In [243…

```
print("mitsos")
```

```
mitsos
```

In [244…

```
print("My name is \"mitsos\"")
```

```
My name is "mitsos"
```

In [245…

```
print('My name is "mitsos"')
```

```
My name is "mitsos"
```

In [248…

```
print ('My name is \'Mitsos\'')
```

```
My name is 'Mitsos'
```

In [249…

```
print ("My name is 'Mitsos'")
```

```
My name is 'Mitsos'
```

There are also the following special characters:

- New line: `\n` (n = New line)
- Tab: `\t`

In [250…

```
print("Line 1\nLine 2")
```

```
Line 1
Line 2
```

In [251…

```
print ("Col 1\tCol2")
```

```
Col 1	Col2
```

In case we want to write a large string that has many special characters inside (quotes, new lines, etc ..) we can use the triple single or double quotes:

In [4]:

```
print( '''
"Be realistic - demand the impossible!"
    Soyez réalistes, demandez l'impossible! - Anonymous graffiti, Paris 19
''')
```

```
"Be realistic - demand the impossible!"
    Soyez réalistes, demandez l'impossible! - Anonymous graffiti, Paris 196
8
```

In [5]:

```
print("""
"I have the simplest tastes. I am always satisfied with the best."
       Oscar Wilde
""")
```

```
"I have the simplest tastes. I am always satisfied with the best."
       Oscar Wilde
```

Combination of variables of different types

float + int result in float:

```
In [15]: 3+0.0

Out[15]: 3.0
```

```
In [16]: 0 + 0.0

Out[16]: 0.0
```

The division always results in float:

```
In [17]: 5/2

Out[17]: 2.5
```

```
In [18]: 6/2

Out[18]: 3.0
```

float/int και string δεν επιτρέπεται

```
In [19]: 4.5 + "μίτσος"

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-19-835a49c7937c> in <module>()
----> 1 4.5 + "μίτσος"

TypeError: unsupported operand type(s) for +: 'float' and 'str'
```

when we mix float / int with boolean then True corresponds to 1 and False to 0:

```
In [21]: 4 + True

Out[21]: 5
```

```
In [22]: 4 * False

Out[22]: 0
```

```
In [23]: 6 / True

Out[23]: 6.0
```

We can also do the following:

```
In [24]: 'Μήτσος' * True # είναι το ίδιο με 'Μήτσος' * 1

Out[24]: 'Μήτσος'
```

```
In [26]: 'Μήτσος' * False #  είναι το ίδιο με 'Μήτσος' * 0

Out[26]: ''
```

We can add True / False variables to each other!

And in general we can do any mathematical operation

```
In [27]: True + True
```

```
Out[27]: 2
```

```
In [28]: True + False + True
```

```
Out[28]: 2
```

```
In [29]: (True + False) / (True + True)
```

```
Out[29]: 0.5
```

```
In [30]: True * True * True * True * True * True
```

```
Out[30]: 1
```

```
In [31]: True * True * True * True * False * True
```

```
Out[31]: 0
```

The `and` and `or` operators with variables that are NOT boolean

Remember the operators `and` and `or`. E.g:

```
In [65]: True and False
```

```
Out[65]: False
```

What if I use them with variables (or constants) that are NOT boolean?

The expression `A and B and C and ... and Z` will return the first expression which is False. If there is none that is False, it will return the last one:

```
In [69]: 5 and '' and 'Μήτσος'
```

```
Out[69]: ''
```

```
In [72]: 5 and 'Μήτσος' and 0.0
```

```
Out[72]: 0.0
```

```
In [73]: 5 and 'Μήτσος' and 3.2
```

```
Out[73]: 3.2
```

But why is this happening? Because when in an expression of the form: `A and B and C`, `B` is False, then it does not make sense to see what value is `C`. Whether `C` is True or False, the result will always be False. So in essence python returns the value of the expression it last evaluated.

This technique is called short-circuit evaluation

Similary the following expression: `A or B or C or ... or Z`, will return the first value which is True. If there is none that is True, then it will return the last one:

```
In [75]: 0 or 5.3 or 'Μήτσος'
```

```
Out[75]: 5.3
```

```
In [76]: 0 or 5.3 or ''
```

```
Out[76]: 5.3
```

```
In [77]: 0 or False or ''
```

```
Out[77]: ''
```

Check the type of a value

The `type` function returns a string that contains the type of a value:

```
In [162… type(2)
```

```
Out[162… int
```

```
In [163… type(2.0)
```

```
Out[163… float
```

```
In [164… type('')
```

```
Out[164… str
```

```
In [165… type('mitsos')
```

```
Out[165… str
```

```
In [166… type(True)
```

```
Out[166… bool
```

```
In [168… type(1==2)
```

```
Out[168… bool
```

```
In [169… type(1+2)
```

```
Out[169… int
```

Type conversion

These are some special functions to convert variables from one type to another:

- `int` converts to integer
- `float` converts to decimal
- `bool` converts to binary
- `str` converts to alphanumeric

Some examples:

```
In [32]: int('42')
```

```
Out[32]: 42

In [33]: int('42.4')

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-33-c0c93863b08a> in <module>()
----> 1 int('42.4')

ValueError: invalid literal for int() with base 10: '42.4'

In [34]: int(42.4)

Out[34]: 42

In [35]: int(True)

Out[35]: 1

In [36]: int(False)

Out[36]: 0

In [37]: int(42)

Out[37]: 42

In [39]: int('mitsos')

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-39-24e8b5b4a1dd> in <module>()
----> 1 int('mitsos')

ValueError: invalid literal for int() with base 10: 'mitsos'

In [40]: int('                                 42')

Out[40]: 42

In [41]: int('42                                  ')

Out[41]: 42

In [42]: int('               42                    ')

Out[42]: 42

In [43]: float('3.4')

Out[43]: 3.4

In [44]: float('3')

Out[44]: 3.0

In [45]: float('')

---------------------------------------------------------------------------
```

```
ValueError                                Traceback (most recent call last)
<ipython-input-45-45d756431581> in <module>()
----> 1 float('')

ValueError: could not convert string to float:
```

In [46]:
```
float('mitsos')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-46-a78f2c30f998> in <module>()
----> 1 float('mitsos')

ValueError: could not convert string to float: 'mitsos'
```

In [47]:
```
float('3.4             ')
```

Out[47]:
```
3.4
```

In [48]:
```
float('   3.4           ')
```

Out[48]:
```
3.4
```

In [49]:
```
float('               3.4')
```

Out[49]:
```
3.4
```

In [50]:
```
float(3)
```

Out[50]:
```
3.0
```

In [51]:
```
float(3.4)
```

Out[51]:
```
3.4
```

In [52]:
```
float(True)
```

Out[52]:
```
1.0
```

In [53]:
```
float(False)
```

Out[53]:
```
0.0
```

In [55]:
```
bool(2)
```

Out[55]:
```
True
```

In [56]:
```
bool(0)
```

Out[56]:
```
False
```

In [57]:
```
bool(3.3)
```

Out[57]:
```
True
```

In [58]:
```
bool(0.0)
```

Out[58]:
```
False
```

```
In [59]: bool(0.000000000001)

Out[59]: True

In [60]: bool('mitsos')

Out[60]: True

In [61]: bool('')

Out[61]: False

In [62]: bool(' ')

Out[62]: True

In [63]: bool(True)

Out[63]: True

In [64]: bool(False)

Out[64]: False
```

Help and instructions

But this is a lot! How will I remember all of these?

You do not need to remember much .. You always have google .. if you ask "how to ... in python" usually the first result will have a very good answer! Recently the creator of python said that he uses google himself to find out how to do some things in .. python.

Nevertheless python contains some basic instructions and documentation with the `help` function:

```
In [300…  help(len)

Help on built-in function len in module builtins:

len(obj, /)
    Return the number of items in a container.
```

```
In [301…  help("".count)

Help on built-in function count:

count(...) method of builtins.str instance
    S.count(sub[, start[, end]]) -> int

    Return the number of non-overlapping occurrences of substring sub in
    string S[start:end].  Optional arguments start and end are
    interpreted as in slice notation.
```

Try also:

```
In [171…  ?len
```