Introduction to Python

Efstratios RAPPOS

efstratios.rappos@heig-vd.ch

Background

- Easy and popular programming language
- Interpreted: must have python installed to use it (already installed in Linux and Mac).
- Two flavors: Python 2.7 and Python 3. Small differences, but not compatible.

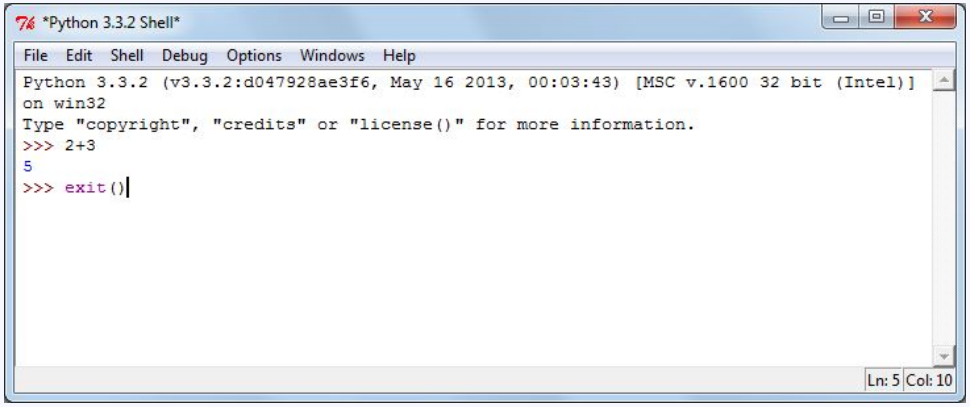
Background

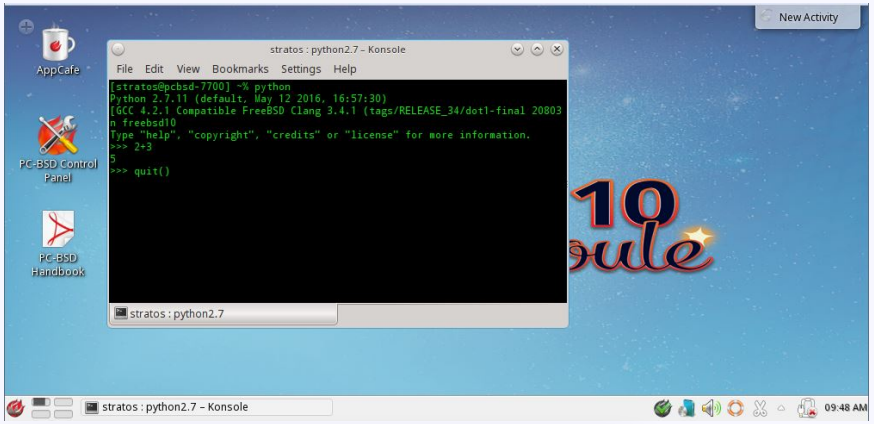
- Write the code in a text file (usually .py)
- Run with `python file.py`
- In linux or mac, can make runnable by adding line `#!/usr/bin/env python` to the top of file
- Then run simply by file.py (or ./file.py etc)
- Can also write commands directly in console (console: type `python`)

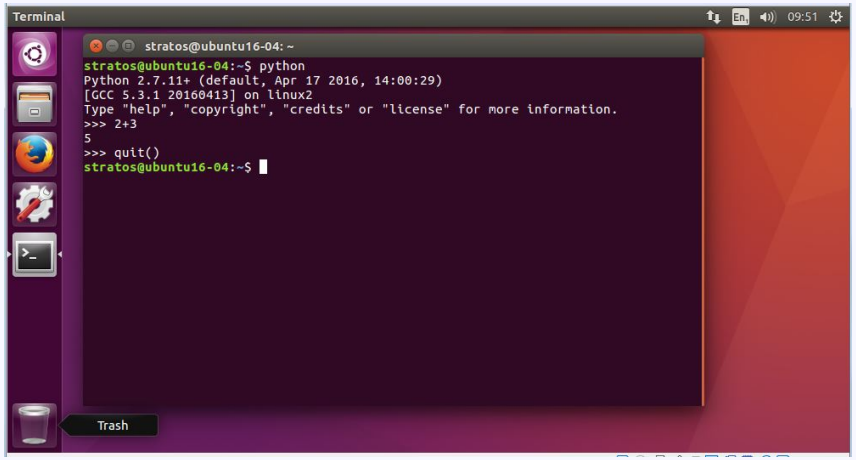
Background

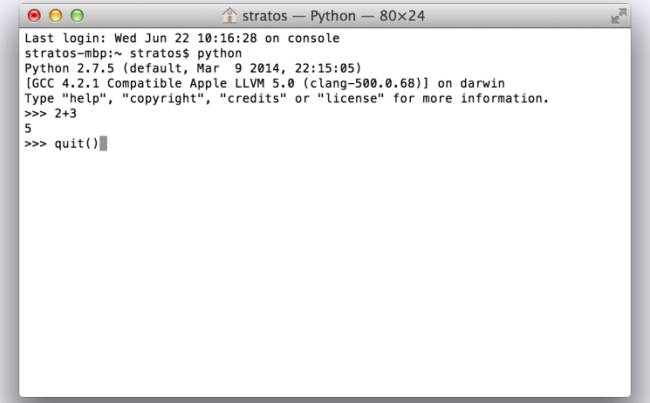
Example: console python

Exit with `exit()` or `quit()` or `control-C`









Background

Whitespace indentation (space, tab at the beginning of a line) is VERY IMPORTANT

Indentation must be consistent, tab is not the same as many spaces

- Indentation decides grouping

```
if x == 3:
....print "X equals 3."
elif x == 2:
....print "X equals 2."
else:
....print "X equals something else."
print "This is outside the 'if'."
```

Exact number of spaces is very important !

most common error for first time users...

Background

Comments start with #

To go to next line prematurely use \

CAPITALS are also IMPORTANT in variable names

Variable assignment

Fundamental variable assignments

A=3

variable (Letters, _ and numbers)

value

```
A=3

B=3.5

C='hello' or C="hello" or C ="""hello"""

D=(3,4,5) or D=(3,'hello',4.5) or D=(3,)
```

Need comma otherwise will be integer 3, not tuple

```
A-> integer
B-> decimal (float)
C-> string
D-> tuple
For Tuples, can retrieve an element via D[0], D[1], D[2]. Elements are
read-only.
```

Variable assignment

Operations

Integer and Float: python converts as needed

```
A = 3 + 2    <= integer
A = 3 + 2.5  <= float (5.5)
B = 2*(1+7) + 3**2        <= integer (25)
C = 100 % 97   <= integer modulo (3)
```

Advantage of Python: integer has no limitation

```
2**200

=>1606938044258990275541962092341162602522202993782792835301
376
```

Float is 8-byte so same limitation as double in other languages (~ 2E-208 – 2E208). Try

```
import sys; sys.float_info
```

Variable assignment

Operations: strings

`C = "hello" + " SU"  <= string joining, a new string is generated and old ones deleted`

`C = "hello" * 3  => "hellohellohello"`

`len(C)` also gives length

For strings one can get the individual characters

`C = "hello" => C[0] is "h"` (a string of len 1)
`C[1:3] is "el"` (a string of len 2)

Variable assignment

Operations: tuples

`D=(3,4,5)` or `D=(3,'hello',4.5)` or `D=(3,)`

Can get individual elements via:

`D[0], D[1] etc`

Can get all the elements in one go via

```
a, b, c = D     <= number of vars must be the same
as the size of tuple
```

```
(a = first element, b = second element)
```

Variable assignment

Operations: tuples

`D=(3,4,5)` or `D=(3,'hello',4.5)` or `D=(3,)`

Elements are read only.

Also cannot add or remove elements from a tuple.

But we can create new tuple with desired elements

E.g., we cannot remove last element of tuple, but nothing prevents us from saying

```
D = D[0:1]  => first 2 elements of tuple
```

We create a new tuple with the fist 2 elements. The old one is deleted.

Variable assignment

Operations: tuples

`D=(3,4,5)` or `D=(3,'hello',4.5)` or `D=(3,)`

Addition

`D = (1, 2, 3) + (4, 5) => (1,2,3,4,5) NEW LIST`

Multiplication

`D = (1,2)*3 => (1,2,1,2,1,2)`

Tuples are very similar to strings

(except tuples can have elements that are other things except characters)

Variable assignment

```
E-> list
F-> dictionary
```

Lists = arrays

```
E=[1,2,3] or E=[1,'abc',3] or E=[1]
```

To retrieve an element, use E[0], E[1], etc. Elements can be modified. Array can expand. Ordering is maintained.

Dictionaries = key-value stores (list of key:value)

```
F={'France':'FR', 'Korea':'KR', 'Switzerland':'CH'}
```

Every Key must be unique in list, using the same key many times => last assignment is remembered

To set/retrieve a value use F[key] eg F['France']. Dictionary can expand. Pairs are not ordered.

Variable assignment

Two more types

A-> integer
B-> decimal (float)
C-> string
~~D-> tuple~~
E-> list
~~F-> dictionary~~

Not often used

Lists are very common.

Can convert from list to tuple

```
li = list(tu)
tu = tuple(li)
```

Variable assignment

Assigning to another variable

```
A=3
B=A
```

↑ new variable ↑ value to get

What happens depends on the type of variable A

A-> integer	new object
B-> decimal (float)	new object
C-> string	new object
D-> tuple	new object
E-> list	same object
F-> dictionary	same object

Variable assignment

Assigning to another variable

A=3
B=A

new variable old value

Effectively the same as:

A=3
B=3

NEW OBJECT:

```
A-> integer                          new object
B-> decimal (float)                  new object
C-> string                           new object
D-> tuple                            new object


A=3       =>       A=3

B=A       =>       B=3               These two '3' are different


X='hello'   =>     X='hello'

Y=X         =>     Y='hello'         These two 'hello' are different
```

Variable assignment

Assigning to another variable

E=[1,2,3]
F=E

↑ new variable ↑ old value

NOT the same as:

~~E=[1,2,3]~~
~~F=[1,2,3]~~

but like this:

E=[1,2,3]
//
F

NEW OBJECT:

```
E-> list                    same object
F-> dictionary              same object
```

```
E=[1,2,3] =>       E=[1,2,3]
F=E       =>       F=[1,2,3]          These two [1,2,3] are the same!
```

Generally, not a good idea to use F=E for lists or dictionaries..
This is because we simply create a duplicate name for the same object, quite confusing and often unnecessary.

Lists (arrays)

Lists are used a lot.

A=[1,2,3,4,5]

- To get an element: A[0], A[1], …
- To change an element in-place: A[1] = 3
- To create a NEW list with a range:
  - A[1:3] => [2,3]

Position of first element to get

Position of element after last element to get

This way, the size of the new list is ALWAYS the diff of the two indexes

  - A[:3] => from beginning,
  - A[3:] to the end,
  - A[:] everything (make a copy)

Lists (arrays)

A=['a','b','c','d','e']

- Length of list: `len(A)`
- To add an element at the end: `A.append('f')`
- To add an element after 2nd element: `A.insert(2,'g')`
- Number of occurrences: `A.count('c')`
- Index of first occurrence: `A.index('c')`
- Reverse a list (in place): `A.reverse()`
- Sort a list (in place): `A.sort()`
- To remove an element (first occurrence): `A.remove('c')`
- To remove an element by index: `del A[1]`
- To remove a range: `del A[1:3]`
- To remove all elements: `del A[:] or A.clear()`(ver.3.3) Same as `A=[]`
- To remove the list (or any variable) and save space: `del A`

Lists (arrays)

A=['a','b','c','d','e'] B=['f','g']

- To combine two lists into a NEW list:

  `C = A+B` C => ['a','b','c','d','e',f','g']

- To add a second list to the CURRENT one:

  `A.extend(B)` A=>['a','b','c','d','e',f','g']

  Note the difference with:

  `A.append(B)` A=> ['a','b','c','d','e', [f','g']]

- *Q: what happens if we run A.extend(A) and A.append(A)?*

Lists (arrays)

A=['a','b','c','d','e']

Remember: `B = A` Does not create a new list, just a new name for existing List.

What if we really want a NEW list (separate to old)?

Solutions

```
B = A[:]     B = A.copy() (ver.3.3)
B = A + []
B = list(A)            #probably fastest
B = copy.copy(A)       # requires 'import copy'
B = copy.deepcopy(A)   # also copies elements of list if needed (eg
                         for list of lists)
```

Control and Loops: if

`If … elif` (=else if) … `else`

Logical comparisons:

`<  >  <=  >=  ==  !=  in  not in`

Combining: `and , or , not`

```
if i==3 or i>10
if i >= 4
if 3 in C                # C = (1,2,3) a tuple, True
if 'a' in D              # D = "abcde"  a string, True
if 3 in E                # E = [1,2,3,4] a list, True
if D == "abcd"           # False
if "hello" < "zello"     # True, can compare strings / tuples
```

Control and Loops: for

For creates loops, but not on a sequence of integers, like other languages

```
words = ['dog', 'cat', 'mouse']
for w in words:
   print w
```

Note w exists after the end of the loop, containing the last value!

If we need to modify the object we are iterating, best to make a copy:

```
for w in words[:]:
   if len(w)>3:
     words.insert(0,w)
```

results in ['mouse', 'dog', 'cat', 'mouse']

Control and Loops: for

To iterate over integers, need to create a sequence via `range()`

```
for i in range(5):
   print i        <= 0, 1, 2, 3, 4
```

Can specify a range

```
   range(2,10)   <= 2,3,4,..,9
```

Can have a step as 3rd parameter

```
    range(2,10,2) <= 2,4,6,8
```

`while` executes a loop if a condition is true

```
i=1
while i < 10:
     print i
     i = i + 1
```

Control and Loops: for

To iterate over a list/tuple, simply

```
for v in ['a','b','c']:
```

To get index and value of a list[] can use `enumerate()`

```
for i,v in enumerate(['a', 'b', 'c']):
```

Q: what do we get from list(enumerate(range(5)))?

To iterate over dictionary, can get key and value at the same time:

```
for k,v in D.items():
        print k
```

No guarantee about the order the items of the dictionary will appear

Control and Loops: for

`break` and `continue` => exit loop or skip iteration

Unique in python: `for` and `while` can have an `else` statement, it is executed if the loop ends normally:

```
for i in range(5):
      print i
else:
      print "end"
```

```
0 1 2 3 4 end
```

`else` will not be executed if the loop terminates due to a `break`

Back to Lists (arrays)

To create a list 'dynamically' ('list comprehensions')

```
squares = []
for x in range(10):
      squares.append(x**2)
```

Same as:

```
squares = [x**2 for x in range(10)]
```

Can also have an optional if at the end

```
squares = [x**2 for x in range(10) if x != 5]
```

```
=> [0,1,4,9,16,36,49,64,81]
```

Functions

If you use a function a lot, can store it using `def`

```
def length(a, b):
    d = (a*a + b*b)**0.5
    return d
```

length(3,4) => 5.0

Modules

Modules are groups of functions defined in separate files and that you can use.

Generally, you will not need to create a module, but you can use existing ones.

Python has many modules already. Other modules can be downloaded/installed from python repositories on the internet

To use a module, first you need to import it (usually at the beginning of your file).

For example, module math

```
import math
A = math.sqrt(81)    =>  9.0
A = math.cos(3.14)   =>  -0.99999
A = math.log(256,2)  =>   8.0
```

Modules

```
import math
A = math.sqrt(81)    =>  9.0
A = math.cos(3.14)   =>  -0.99999
A = math.log(256,2)  =>   8.0
```

To avoid using `math.` before functions all the time, can use:

```
   from math import sqrt, cos, log
```

or

```
   from math import *
```

The we can use

```
from math import *
A = sqrt(81)
A = cos(3.14)
A = log(256,2)
```

Modules

Because code in a module can be run on its own, or (imported) from other modules, a test for `__main__` can be done to determine which case it is.

This is quite common. E.g.

```
def length(a, b):
   d = (a*a + b*b)**0.5
   return d


# This code below runs when file is run on it own, but not when
#           file is imported from another file.

if __name__ == '__main__' :
   a = 5
   b = 6
   print length(a,b)
```

Read and Write files

Read from a file:

```
1 John Brown
2 Emma Lewis
3 Maria Johnson
```

"r" means read, and is optional (default is "r")

```
file1 = open("C:\\Users\\name\\Documents\\input.txt", "r")
for line in file1:          <= reads file one-line-at-a-time
   element = line.strip().split(" ")
     <= element[0] is 1, element[1] is 'John', element 2 is Brown
file1.close()
```

Useful string functions: strip() removes spaces at beginning / end
split() splits a string into many strings

Need to remember to close() the file.
Alternatively, the following version ensures the file is closed automatically when "with" finishes

```
with open('filename') as file1:
   for line in file1:
        element = line.strip().split(" ")
```

Read and Write files

Write to a file:

```
file1 = open("filename", "w")    <= w means write
file1.write("abcdefg\n")  <= does not change line automatically
file1.write("12345\n")
file1.write(str(2))   <= does not convert to string automatically
file1.close()                        need to use str()
```

If file exists, it is replaced.
Just as before, we can use "with", this way the file is closed automatically when "with" finishes

```
with open('filename',"w") as file1:
   file1.write("abcedef\n")
   file1.write("12345\n")
```

Read and Write files - unicode

Read from a unicode file – use `open` from the `codecs` module

```
1 John Brown
2 Hélène Lewis
3 Maria Johnson
```

```
import codecs
file1 = codecs.open("input.txt", encoding = "utf-8")
for line in file1:
   element = line.strip().split(" ")
file1.close()
```

Write to a unicode file

```
with codecs.open('filename',encoding="utf-8", mode="w") as file1:
   file1.write(u"abcedef\n")
   file1.write(u"12345\n")
```

In Python3, the `codecs` functionality is included in the default `open()` function.