Image Processing and Computer Graphics

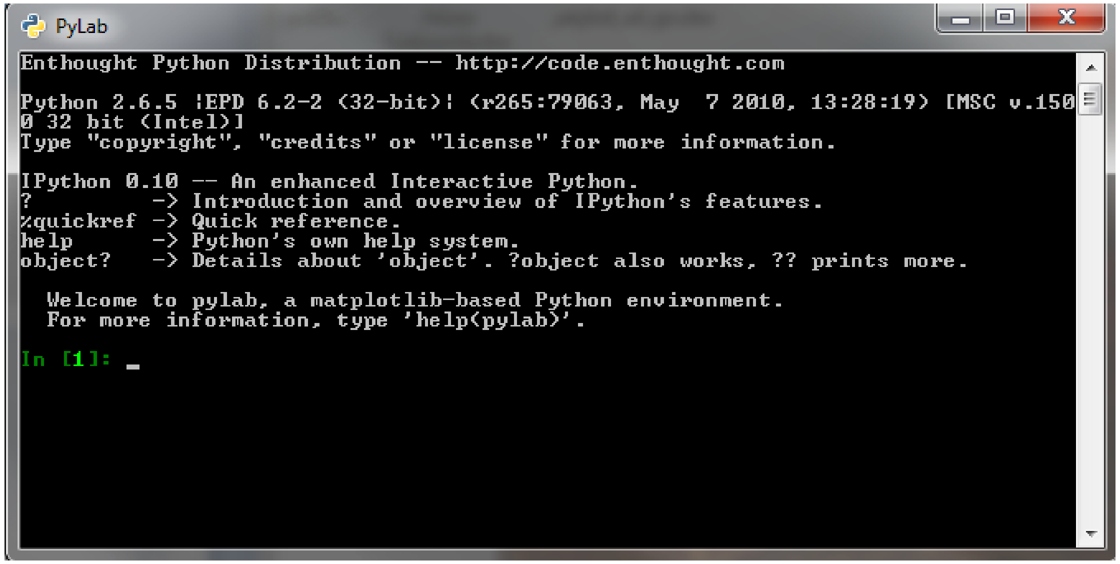
Part 1

Adam Sankowski, Marek Kociński

2012

Introduction to Python

- Run PyLab from Enthought folder in start menu
- On MacOS type **ipython --pylab** in the Terminal



```
Enthought Python Distribution -- http://code.enthought.com

Python 2.6.5 |EPD 6.2-2 (32-bit)| (r265:79063, May  7 2010, 13:28:19) [MSC v.150
0 32 bit (Intel)]
Type "copyright", "credits" or "license" for more information.

IPython 0.10 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object'. ?object also works, ?? prints more.

  Welcome to pylab, a matplotlib-based Python environment.
  For more information, type 'help(pylab)'.

In [1]: _
```

- Type **?** to read brief introduction to this environment

Introduction to Python

- Typing **object?** or **?object** will print information about object
- Double **?** Will display more information
- There is also full python help. Just type **help(object)**
- Pressing *TAB* during typing will display any available methods or variable names
- History of commands is available by pressing *Ctrl + p* to print previous command or *Ctrl + n* to print next
- Type **whos** to print all variables you created
- Type **reset** to clear all variables, functions and imports

Introduction to Python

- Type in command to print out simple string and press Enter:

  print ”Hello, World”

- You can also use python shell like calculator, just type in:

  2+2   18/3   18/7

- You need to force float type to get correct answer:

  18.0/7   18/7.   18./7.

- To raise to the power any numbers simply type:

  22   3**3   8**3   -3**15**

Variables

- To set a variable integer and float

  x = 18 **y = 12.0** or **y=12.**
- To check a variable's type

  type (x) **type(y)**
- An arbitrary long integer

  a = 12345678901234567890 **type(a)**
- Now you can type:

  x + y
- To get variable value from user type:

  a = input(„Enter number: „)
- After typing value you can use it:

  a4** **b= a + 20**
- Remove 'a' from namespace:

  del a

Functions and Modules

- Python has many functions. Type:

  pow? abs? floor? sqrt? max? round?
- Use those functions
- Type in **help(math)**. It will print all functions from that module
- Find used functions and look for more
- Functions from modules can also be used like:

  math.floor(12.4) math.sqrt(9)
- Casting

  int(2.718281828) float(2) 1+2.0

Scripts

- Instead of typing every command you can create and save script with extension “.py”
- First go to folder where you want to save program using:

  ls   cd ..   cd folder/folder   cd c:/

- To create new program type:

  edit program_name.py

- By default program is open with notepad
- To try out program type:

  print „My first program”

- Program will execute when you save and close notepad

Scripts

- If you want to run script form command line type:

  run program_name.py

- **raw_input()** is used to get string variable from the user. Type in your program:

  x = raw_input(„Enter name : „)

  print „Hey „ + x

- This code will get string value form user as string variable **x** and print on screen two combined strings

Strings

- Go back to command line
- Strings can be written in two ways:

  „Hey” **‚Hey’**

- Both ways has same effect, but when you use single quotation in string like:

  ~~**‘He’s a student’**~~

- String will be seen as **‘He’** and rest will return an error. That’s why in this case we will use double quotation:

  „He’s a student”

Strings

- Another way is to use backslash before single quotation to say python compiler to treat next character as part of a string:

```
‘He\’s a student’
```

- *Backslash* works the same way with double quot. Try:

```
„He said \”Hey\” to me”
```

- There is simple method to combine two strings. Try:

```
a = „Image „
b = „Processing”
print a + b
```

Strings

- When you need to print string combined with number variable like:

```
num = 42
```

- You can’t just type:

```
print „My mom is „ + num
```

- Because string and integer cannot be concatenated. We can create another variable:

```
str_num = str(num)
print „My mom is” + str_num
```

- Another way is to put integer in signs above *TAB* key:

```
print „My mom is „ + `num`
```

Strings

- String can also have blank spots where some string values can be inserted:

  welcome = ‘Hello %s, have a nice day’

- **%s** means that it will be replaced with some string. There are two ways of doing it:

  name = ‘John’

  print welcome %name

Or

print welcome %’John’

print ‘Hello %s, have a nice day’ %name

List

- Lists are structures witch store data. List can have multiple elements. Lets try to create list of strings:

  family = [‘mom’, ’dad’, ’bro’, ’sis’, dog’]

- Once you created a list, all elements in it are numbered. Numeration starts from 0. Now you can use elements from list by typing their number:

  family [2] **family[4]**

List

If your list is long you can refer to its elements from end to beginning. Last element is numbered as -1, second last as -2 etc.

family = [‘mom’, ’dad’, ’bro’, ’sis’, dog’]

```
[0]   [1]   [2]  [3]  [4]
[-5]  [-4]  [-3] [-2] [-1]
```

- String also can be a sequence, and numeration is the same, for example:

‘graphics’[3] **‘graphics’[-5]**

- Will return ***‘p’***.

List

- List also is capable of storing different types of values: integers, floats, strings and even other lists. For example:

  list = [‘car’, 4, 3.5, [3, 2,1, ’Go!’]]

- Now look into your list and check if list is correct:

  print mylist or **mylist** or **mylist[:]**

- And try to call few elements of list:

  list[2]

  list[3]

  list[3][3]

Slicing

- Slicing is extracting part of the list. To show how its done, let’s create example list:

  example = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

- Now we can view just a part of that list using command:

  example [x:y]

- Where **x** is first position of the list we want to view and **y** is position before which we want to end. For example:

  example [4:8]

- Will return:

  [4, 5, 6, 7]

Slicing

- When you leave **x** or **y** empty then it will show all elements to the end or to the beginning. Try:

  example[4:]   example [:7]   example[-5:]   example[:]
- There is also third optional parameter of slicing. It show how much you want to increment slicing. For example:

  example [1:8:2]   **example[10:0:-2]**

  (or **example[10::-2]**   **example[::-1]**)
- Try few different combinations. You can also leave some parameters empty
- Does the list contain x?

  45 in example   **45 not in example**

Lists functions

- There are functions to manage lists:

  len(example) returns length of list

  max(example) returns biggest element of list

  min(example) returns smallest element of list
- After typing every code below view list (**example[:]**) for changes and try your own

  del example[3] deletes third element of list

  example [2] = 12 replace element value in list type:

  example [4:4] = [5, 6] put two elements after forth

  example [2:4] = [] delete elements second and third

List Methods

- Many structures have additional options called methods. Methods are used after dot. Lets make some list for example:

  list = [23, 3, 12, 45, 76, 3, 7]

- These are example methods:

  list.append[45] add 45 at the end of list

  list.count[3] count how many 3 are in list

  list.sort()

  list.extend(example) add example list at the end

- View **list[:]** to see result

  List.append(example) add example list at the end

- View **list[:]** to see result

Methods

- Type **help(list)** to find more methods. Look and try methods:

  index

  insert

  pop

  replace

  remove

  reverse

  sort

  find

  lower (you need to create string list to test it)

Tuple

- Tuple is constant list. It means that tuple once created cannot be changed. It’s used when we don’t want to accidently change some element in structure. Tuple doesn’t have methods like **pop**, **remove**, **reverse**. All you can do with tuple is get elements from it. Tuple is created similar to list, but instead of **[]** we use **()**:

  tuple = (4, 7, 23, 87, 24) or simple **tuple = 4, 7, 23, 87, 24**

  str_tuple = (‘cat’, ’dog’, ’mouse’)

- Now all you can do with it is to call elements:

  tuple[2]   **str_tuple[1]**

Dictionary

- Dictionary is another structure. That structure has couples of elements called keys and values. What this means is that instead of index like in lists or tuples, you can use strings or integer to find values. For example:

  dict _ages= {‘Dad’:47,’Mom’:45,’Bro’:19}

  dict _ages1= {47:‘Dad’,’45:’Mom’,19:’Bro’}

- Keys are strings named as members of family and values are integers representing age. To find age of mom type:

  dict_ages[‘Mom’]   **dict_ages1[45]**

- Type **help(dict_ages)** and try out dictionary methods like:

  copy **has_key** **clear** **get** **update**

Working with files

- To use a file you need to create an object. Object must be equal to open functions. This function takes two parameters: path to your file and what can you do with it (read or write)

  fObj = open (‘c:/your_path/a.txt’, ’w’)

- This line will also create that file if there was no such file in that path. We called that file with attribute **‘w’** so we can write some text in it:

  fObj = write(‘Hello I\’m just a text in a file’)

- To save changes in that file you need to close it:

  fObj = close()

- Now check your file for changes

Working with files

- Now open your file so you can read it:

  fObj = open (‘c:/your_path/a.txt’, ’r’)

- To read from file we will use function **read()**. This function can take as a parameter number of bytes you want to read :

  fObj.read(5)

- It should print **‘Hello’**. Using function without parameter will read file to the end:

  fObj.read()

- This should print **‘ I’m just text in a file’**

Working with files

- You can also read file line by line (first you need to write few lines in your file) using **readline()** function, or read all lines using **readlines()** function
- If you want to write lines of text into file use **\n** in your string to end current line.
- Type **help(fObj)** to view other file functions, and try them as well

If statement

- Create new script or edit existing
- **If** statement can have block of instructions performed only when statement is true:

```
fish = tuna
if fish == tuna:
        print „This fish is tuna”
```

- Single equal sign assign value to variable. Double equals are used to compare. If variable ***fish*** has value ***tuna*** then print out string ***„This fish is tuna***”. Execute program. Now change in your code value of **fish** to something else and try again.

Elif and else

- Thanks to **elif** (else if) if statement can have few cases, and if none of cases is correct, then we can use **else**.

```
fish = tuna
if fish == tuna:
        print „This fish is tuna”
elif fish == sardine:
        print „This is sardine”
else:
        print „I don’t know what that is”
```

- Change **fish** value to test all statements. You can also nest more statements inside existing ones.

Statements

- Statements can be something more than just a comparison (***==***). You can also use : **>**, **<**, **>=**, **<=**,**!=**. There is also possibility to use logical functions like **and**, **or** :

```
number = 5
if number >= 3 and number <= 8:
        print „This number is between 3 and 8”
```

While

- While loop repeats a block of code until statement becomes false. This is a simple counting to 10 example:

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

- While will execute ten times and print values from 1 to 10. **b+=1** is a shortcut from **b = b+1** which is incrementing variable **b** by 1.

Break

- In some cases we want to have infinite loop until some event happens. This is an example of how break works:

```
while 1:
    text= raw_input(‘Type \’quit\’ to exit’ )
    if text==‘quit’:
        break
```

- This loop will repeat itself until user type in proper string. **while 1:** creates infinite loop.

For

- For loops through the objects like lists or tuples. Starts with first element and ends after last one, for example:

```
grocery = [‘bread’, ’butter’, ’milk’, ’cheese’]
for food in grocery:
    print ‘I want ’ + food
```

- First we created **grocery** list of strings. Next variable **food** is created and it is taking values from **grocery** list.

For

- Let’s try another example of for loop. This time with numbers:

```
for item in range (6):
          print item
```

- This loop will print numbers from 0 to 5.
- You can also treat string as a list and type:

```
for item in ‘abcde’
          print item
```

- Loop will take letters from string like elements from list, and print letters from **a** to **e**

Functions

- To create simple function you need to type a name and arguments taken by your function. After that type in a code:

```
def plusten (x):
        return x + 10
```

- This function adds 10 to a number. You can also set default parameter if user don’t type any:

```
def plusten (x = 0):
        return x + 10
```

- Calling function like: **plusten()** will return 10

Tuple parameters

- When you need to build a function but you don't know how many parameters will be needed, then set tuple as parameter, for example:

```
def food_list (*food):
    print food
```

- Star before parameter means that this parameter will be a tuple and it will take every number of parameters. Try to use this function couple times with different number of string parameters:

```
food_list ('apples')     food_list('bananas', 'peaches')
```

Dictionary parameters

- Another type of parameter of your function is dictionary:

```
def food_list (**items):
    print items
```

- Double star means that dictionary will be a parameter in this function. To use this function you need to type some dictionary as parameter, for example:

```
food_list (apples=4, bananas=2, peaches=6)
```

- Function will print out whole dictionary

Mixed parameters

- Every parameter type can be added next to another. To prove that let’s try:

```
def shopping(shop_name, *busses, **foodlist):
        print ‘Go to :‘
        print shop_name
        print ‘Use those bus lines :’
        print busses
        print ‘Buy :’
        print foodlist
```

- Now enter

```
shopping (‘biedronka’, 12, 23, 34, bacon=3, eggs=10)
```

Classes and objects

- Previously we used methods from lists, tuples and dictionaries. Now we will learn how to create methods. Methods are the same thing as functions but they are inside of classes. Class is a part of code containing variables and methods, which can be used by many objects you will create. First let’s create a simple class:

```
class myClass:
    def createName(self,name):
        self.name = name
    def printName(self)
        return print self.name
```

Classes and objects

- We created class called **myClass**. This class contains two methods. Those methods are similar to functions, but you have to remember to add one extra parameter called **self**. This is a temporary placeholder for objects we will create now:

  first = myClass()

  second = myClass()
- Now when we use methods from **myClass** in those two objects, **self** will become **first** for first object, and **second** for second object

Classes and objects

- Type in:

  first.createName('Adam')

  second.createName('Grzegorz')

- Objects **first** and **second** used method **createName** from class **myClass**. Now use second method from **myClass**:

  first.printName()

  second.printName()

- As you can see single class can be used separately by different objects and hold individual values.

__init__

- Classes can also have methods which are executed immediately after creating object corresponding to that class. If you want to put that kind of method into your class you have to name it **__init__**(*underscore, underscore, init, uderscore, underscore*)

```
class someClass:
    def __init__(self):
        print „This is __init__ method"
```

- Now create new object and watch what happens:

```
newObj=someClass()
```

Parent and child classes

- Classes can inherit variables and methods from other classes. It’s useful when you want to use variables from other class in new one, but you don’t want to type again whole structure form other class. Let’s write an example:

```
class Dad:
        var1=‘Hey I\’m dad’
class Mom:
        var2=‘Hey I\’m mom’
        var3=‘I want you to clean your room son’
```

- Those two classes will be parent classes with variables

Parent and child classes

- Now we will create child class which will inherit variables from both parent classes:

```
class Child(Mom, Dad):
    var4='I'm a child'
    var3='My room is clean'
```

- To take variables from single or multiple parent classes simply type them one after another in class definition. Now let's create child object:

```
childObject = Child()
```

Parent and child classes

- Print all child object variables:

  childObject.var1

  childObject.var2

  childObject.var3

  childObject.var4

- As you can see variables **var1** and **var2** has the same values as in parent classes, but **var3** has changed value because we overwritten it in **chlidClass**