Python
180225_1131_IAI ®

Version 2.0

Sela College

This course material has been prepared by:

Sela Software Labs Ltd.

14-18 Baruch Hirsch St. Bnei Brak 51202 Israel

Tel: 972-3- 6176666 Fax: 972-3- 6176667

The Use of this book

The material in this book is designed to assist the student during the course. It does not include all of the information that will be referred to during the course and should not be regarded as a replacement for reference manuals.

Limits of Responsibility

Sela Software Labs Ltd invests significant effort in updating this book, however, Sela Software Labs Ltd is not responsible for any errors or material which may not meet specific requirements. The user alone is responsible for decisions based on the information contained in this book.

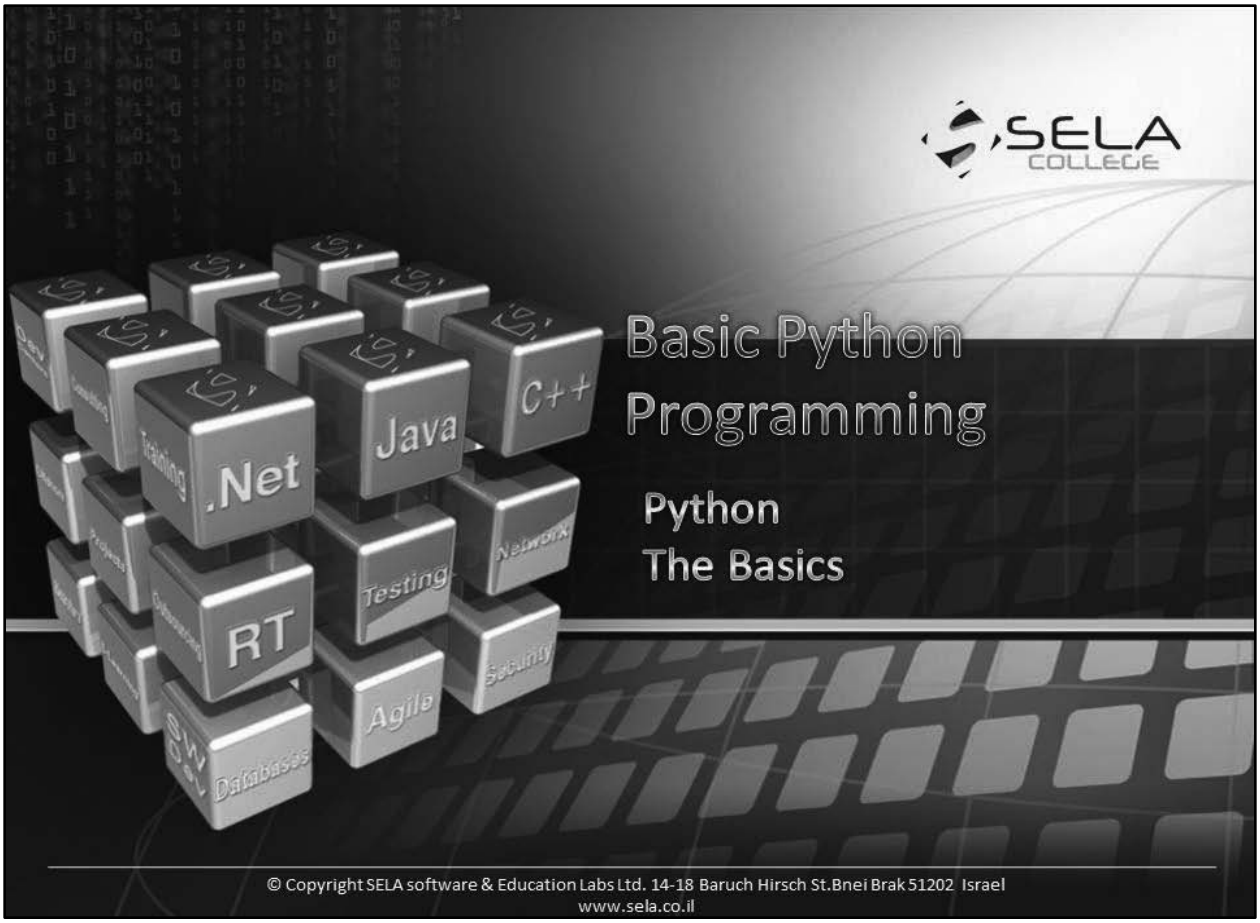
Protected Trademarks

In this book, protected trademarks appear that are under copyright. All rights to the trademarks in this material are reserved to the authors.

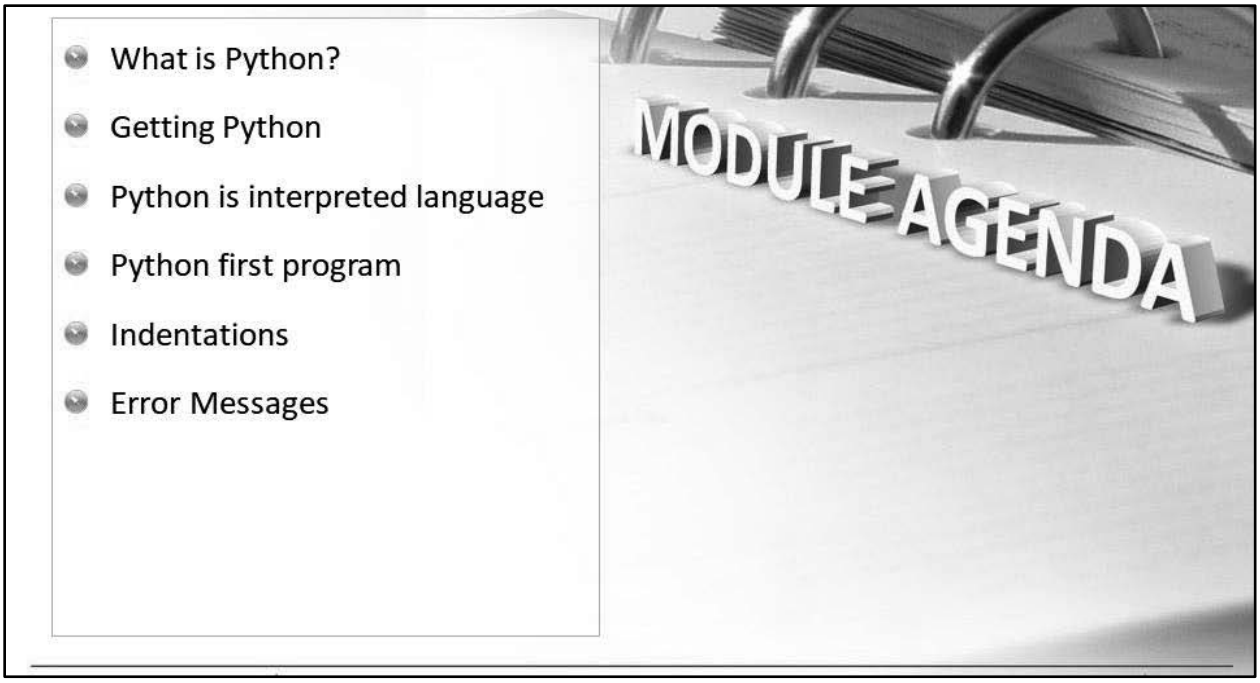
SELA wishes you success in the course!

Table of Contents

Basic Python Programming



Agenda



What Is Python?

- Python is a widely used, interactive, general purpose interpreted language
- Python is stable, cross platform programming language
- Python is a high level and object-oriented programming language
- Python is an Open Source software, distributed under a liberal license
- Python is simple, intuitive, dynamic and is easy to read and understand
- Designed by Guido Rossum in the late eighties
- The final python 2.x version is 2.7
- New major release after python 2.7 is 3.x. Python 3 is not backward compatible

Getting Python

- Where Do I Get Python?
- The most up-to-date and current source code, binaries, documentations, news, etc. are available at the official website of Python:
  - **Python Official Website :** http://www.python.org/
- Python documentation from the following site:
  - www.python.org/doc/
- Python download for Windows, Solaris, Linux:
  - http://www.python.org/download/
- If you are running a Linux (or most UNIX) system, you probably already have some python version installed on it.

Python is interpreted language

- Python is an interpreted language. It means that our code processed at a real time, by interpreter, without compiling it before the execution.
- Interpreted code runs slower than a compiled one, because the interpreter must analyze each statement of the program each time it is executed.
- This run-time analysis is known as "interpretive overhead".

Python First program

- One of the ways to run python code is to write a script file that contains python commands
- Python scripts usually have an extension of **.py**
- Any text editor can match this purpose
- There are several IDEs (integrated development environment) for python language. They are used for editing, debugging, running and testing our code. The most common IDEs are: PyCharm, Spyder, Pydev, IDLE …
- Lets start with a simple python script – first.py
  - Open any text editor or IDE
  - Save the first program as first.py
  - Type the following command in it:

print ("Hello Python!")

  - Run the script. If the commands are correct, python will execute the script

Python First program

- Another way to run python commands is to run python shell – the interactive mode
- To do so, open shell or command processor (like *cmd*) and type python exe path and you will see the following output:

```
Python 3.4.2 (v3.4.2:ab2c023a9432, Oct  6 2014, 22:15:05) [MSC v.1600
32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print ("Hello Python")
Hello Python
>>>
```

- In interactive mode, python interpreter is waiting for users commands and executes them as given
- Whenever you exit the interactive mode, the commands are gone

Indentation

- One of the most important things for flow controls, functions definitions, classes and other block-defined statements is the fact that there is no braces to indicate blocks of code.
- Blocks of code are denoted by line indentation
- Indentations are represented by any number of spaces/tabs, but all statements within the same block must be indented the same.

Error Messages

- If you incorrectly type a statement or placed the statement in incorrect place, the interpreter will detect the error and tell you what it is and where it is located.
- For example, look at following code (This program contains one small error).

```
print("hello")
    print("world!")
```

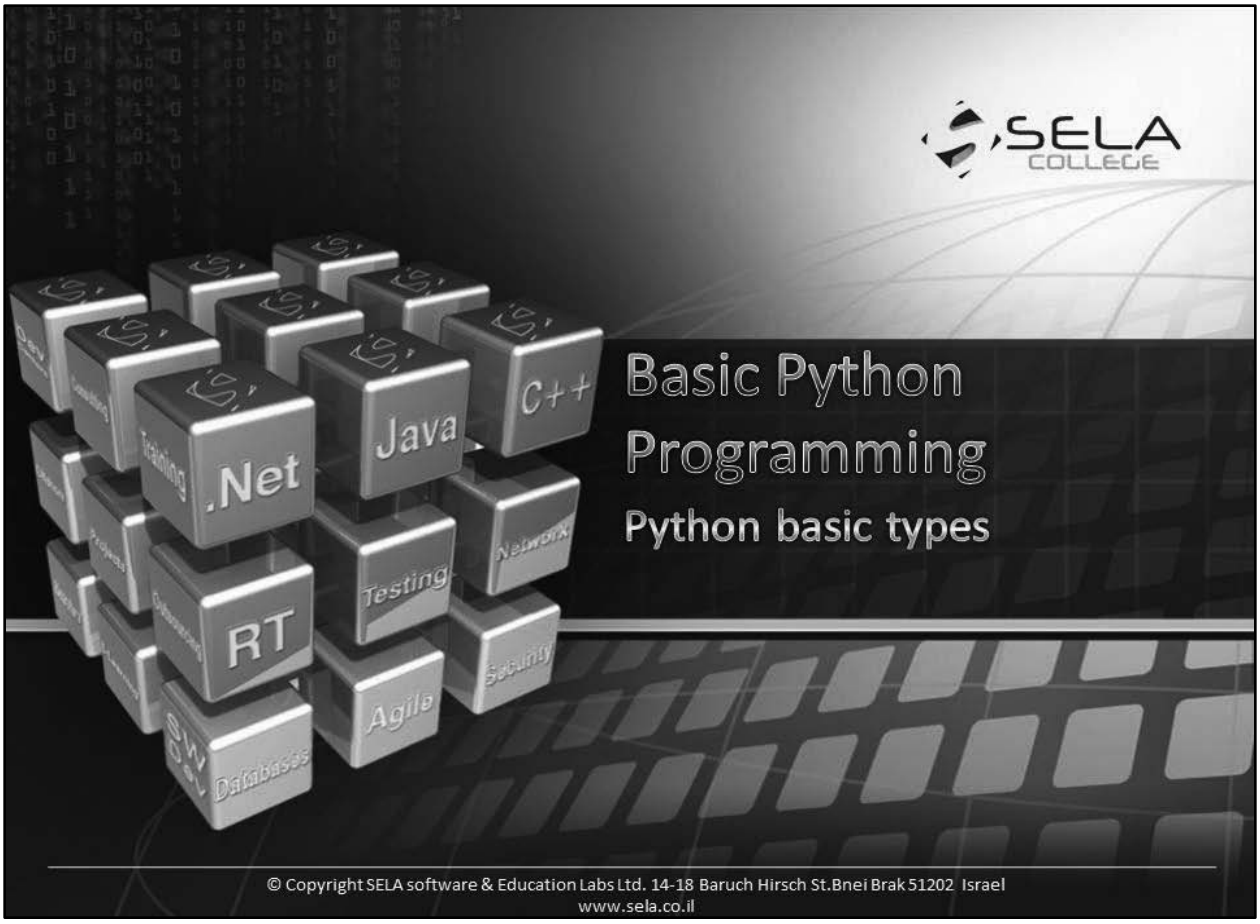
---------- the following output is received ----------

```
File "hello.py", line 2
    print("world!")
    ^
IndentationError: unexpected indent
```

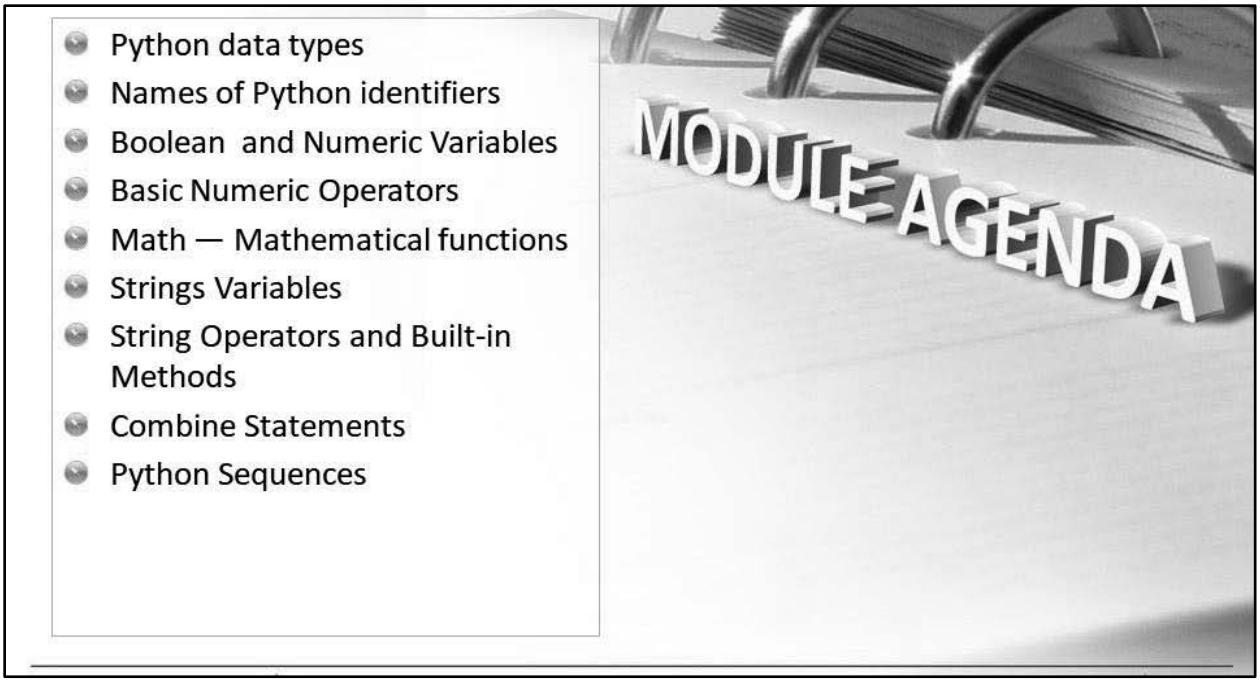
Error Messages – Cont'd

- Non-treated will terminate the process with non-zero exit status
- The interpreter prints an error message and its stack trace

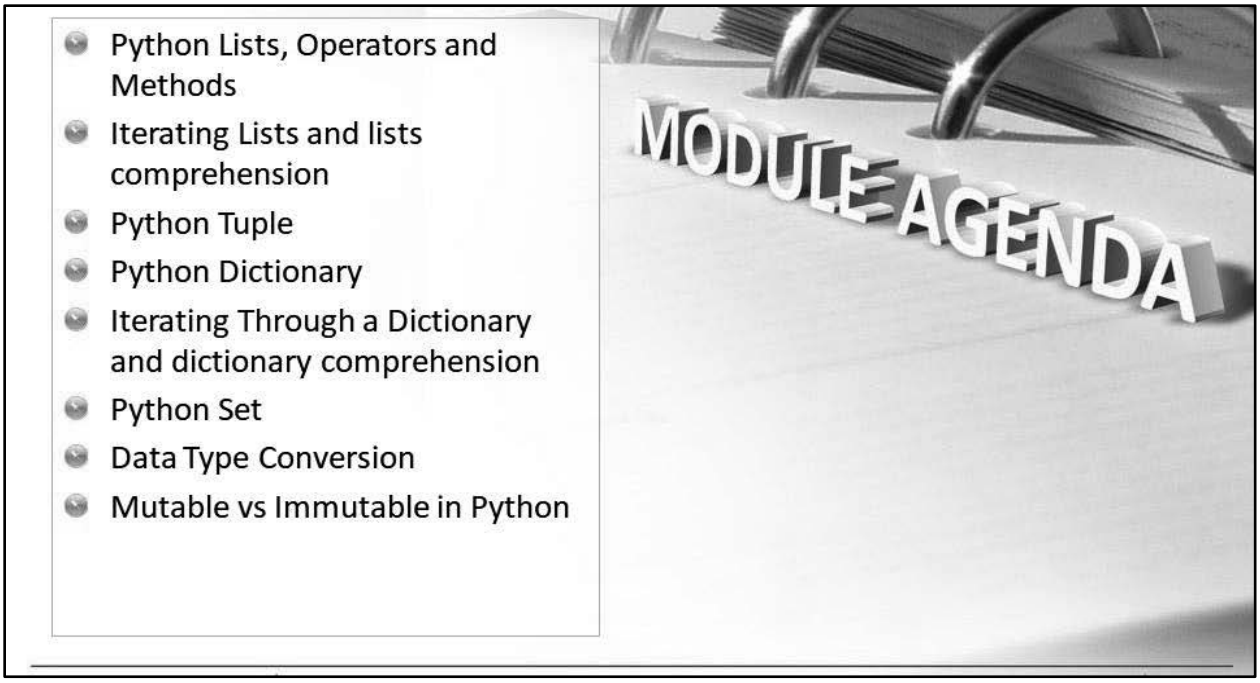
Basic Python Programming



Agenda



Agenda



Python data types

- Python is dynamic language, so you don't need to declare your variables
- Variables must be assign before we can use them
- Python built-in types:
  - Boolean
  - Numeric types
    - Integers - equivalent to C longs in Python 2.x
    - Long integers of non-limited length; exists only in Python 2.x
    - float: Floating-Point numbers, equivalent to C doubles
    - complex: Complex Number
  - Sequences:
    - String
    - List
    - Tuple
    - Set
    - Dictionary

Names of Python identifiers

- Rules for a python identifiers (variable name, function, class, module):
  - A identifier name may contain:
    - Digits (0 to 9).
    - Letters, both lower case and upper case ('a'-'z' and 'A'-'Z').
    - Underscores ('_').
  - The first character must not be a digit and preferable not underscore
  - It is extremely important to choose meaningful names for identifiers and not preserved python name.
  - All identifier names are case-sensitive.

Names of Python identifiers – cont'd

- Naming convention for Python:
  - All identifiers should be lowercased with underscore as words separator (*sum_of_digits*)
  - Class names start with an uppercase letter and all other identifiers, like variables and functions with a lowercase letter.
  - Starting an identifier with a single leading underscore indicates, by convention, that the identifier is meant to be private.
  - If the identifier starts and ends with two trailing underscores, the identifier is a language-defined special name.

Boolean Variables and None

- Boolean variables:
  - Boolean variables can hold only True or False values.
  - For example:

    flag = True

    isOk = False

- None value (of NoneType)
  - var = None

  None is used to represent the absence of a value

Numeric types

- In python 2.X integer can be represented by both *int* and *long* types. *int* is for small values (4 bytes), *long* have unlimited digits count
- In python 3.x there is only one integer type – *int, with* unlimited digits count (line *long* in python 2.x)

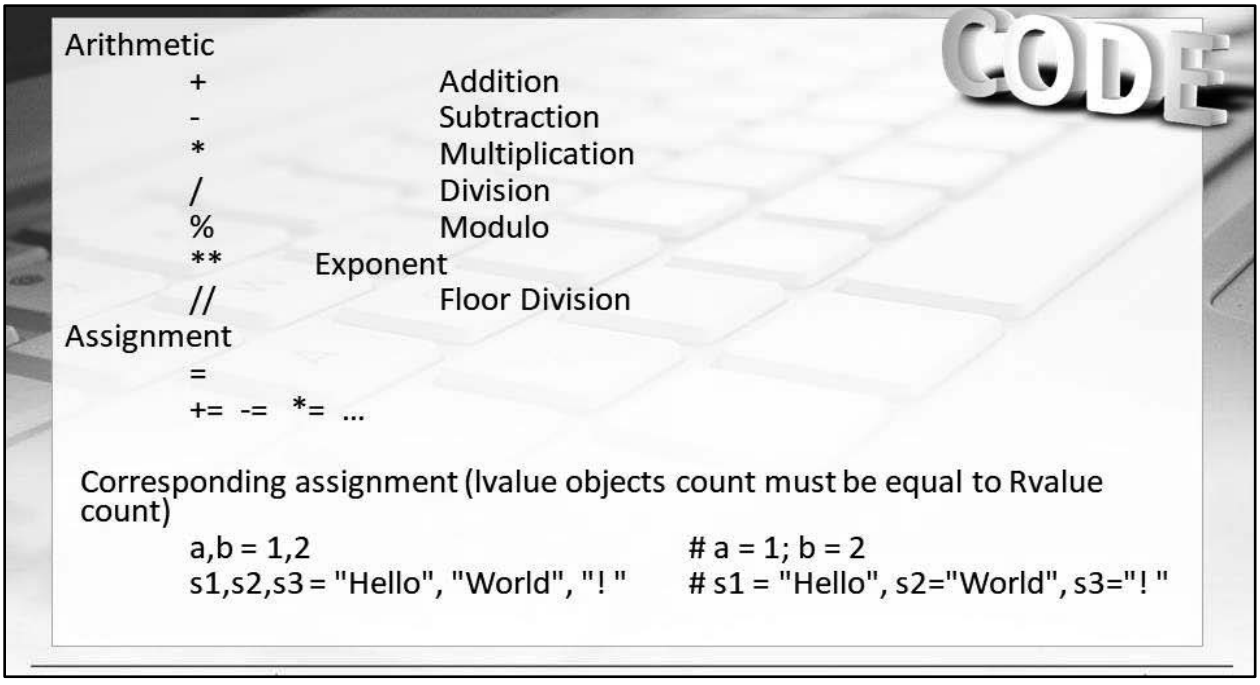
int: 123, -78, 0x5A, 0567

long: 51924361987678, 0xDEFABCECBDAECBFBAEl

float: 0.5, -77.99, 12.3+e45

complex: 3.14j, -0.6545+0J, 4.53e-7j

Basic Numeric Operators



Arithmetic

+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo
**	Exponent
//	Floor Division

Assignment

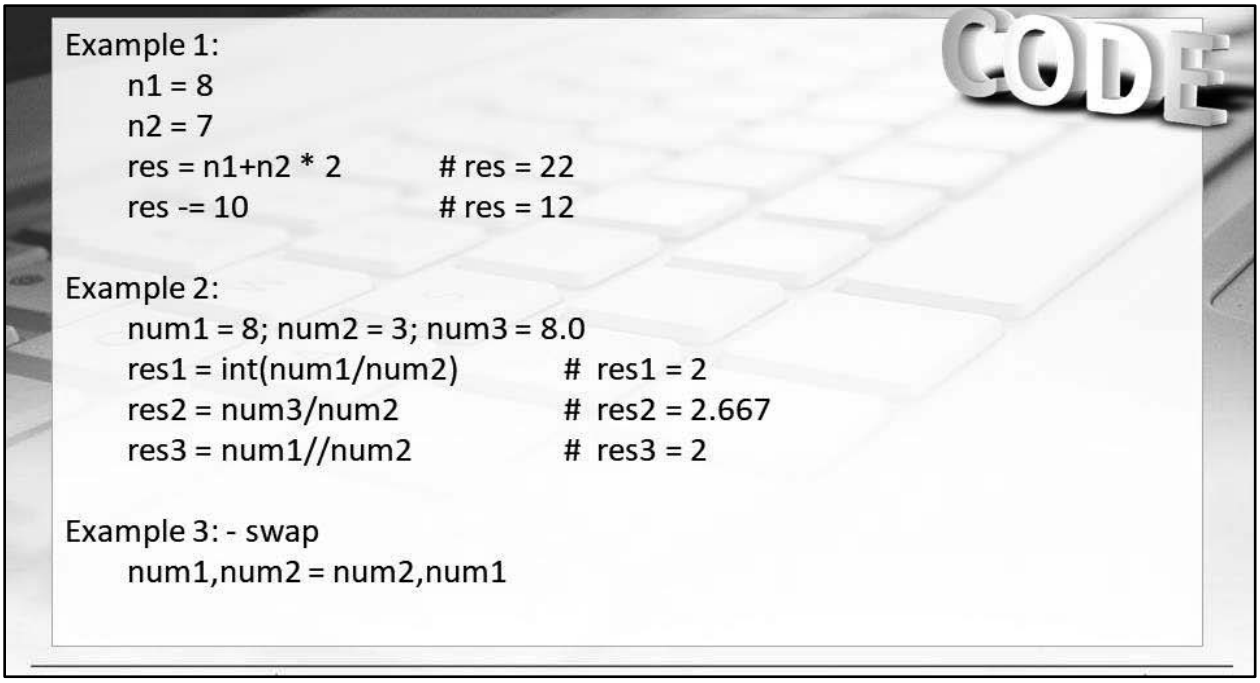
=

+= -= *= ...

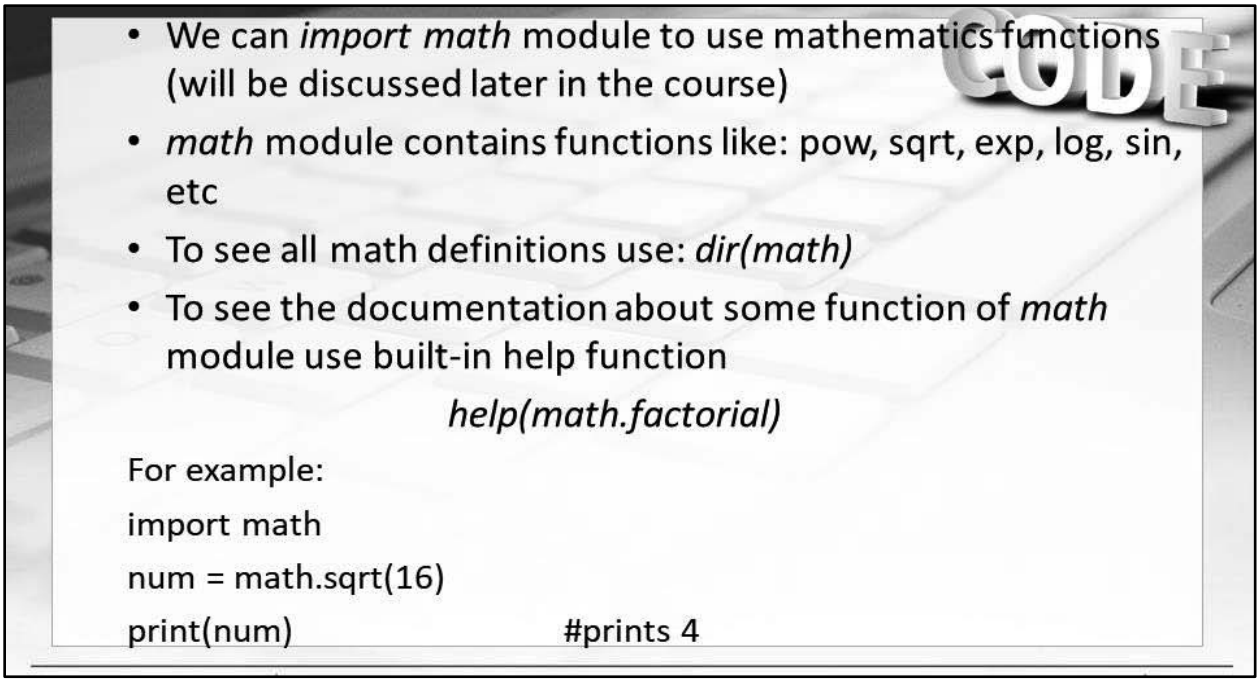
Corresponding assignment (lvalue objects count must be equal to Rvalue count)

```
a,b = 1,2                          # a = 1; b = 2
s1,s2,s3 = "Hello", "World", "! "  # s1 = "Hello", s2="World", s3="! "
```

Basic Numeric Operators – cont'd



Math — Mathematical functions



Python index-base Sequences

- The most basic data structure in Python is a **sequence**.
- Python has several built-in types of sequences: lists, tuples, dictionaries, sets and strings
- The lists, tuples and strings are index-based sequences.
- The index-based sequences are support index access and index-range access
- positive and negative indexes are supported (positive indexes starts with 0, negative with -1)
  - A negative indexes are counted from the end. So we can access the last element is with -1 index, the second to the last element would be -2, and so on.

Strings Variables

- Strings in Python can be created using single quotes, double quotes, and triple quotes.
- Python treats single quotes and double quotes strings the same:
- For example:
  - s = "Hello World!"
  - name = 'Daniel Kohen'
- An escape character interpreted in both single-quoted and double-quoted strings.

Strings Variables – triple quotes strings

- Triple quotes strings can span for several lines and they consist of three consecutive single or double quotes.

- For example:

```
s = """this is a long string with several lines
and escape character  like tab: \t and newline: \n.
the end"""
Or:
s = '''this is a long string with several lines
and escape character  like tab: \t and newline: \n.
the end'''
```

Strings Variables – raw strings

- Python can define raw strings, that blocks escape characters
- Putting the ***r*** character before single, double or triple quotes string will define them as raw-string
- Raw-strings are used to backslash escape characters

- For example:

```
path = "C:\\temp\\newDir\\file.txt" – double back-slash for
                each back-slash character
or
path = r"C:\temp\newDir\file.txt"
```

String Operators

+	Concatenation
*	Repetition
[]	Slice
[:]	Range Slice
in/not in	Membership
%	Format

- Example :

```
s1 = "abc"; s2 = "defgh"
res1 = s1 + s2          # res1 = "abcdefgh"
res2 = s1 * 2           # res2 = "abcabc"
res3 = s1[0]            # res3 = "a"
"bc" in s1              # returns True
```

String ranges

```
s = "abcdef"
print(s[1:4])       #  "bcd"
print( s[:4])       #  "abcd"
print( s[1:])       #  "bcdef"
print( s[1:-1])     #  "bcde"
print( s[:])        #  "abcdef"
print( s[::-1])     #  "fedcba"
print( s[::2])      #  "ace"
print( s[1::2])     #  "bdf"
```

String as Input

- python 2.x
  - raw_input(*text*) – reads string from as input, after prompting user with a *text*

    name = raw_input("Please enter string ")

    suppose you entered: *input* the output will be: **input**

    suppose you entered: *1+1* the output will be: **1+1**
  - input(text) – reads and evaluates the input string after prompting user with a *text*

    name = input("Please enter some string ")

    suppose you enter: *input* the output will be: **input**

    suppose you enter: *1+1* the output will be: **2**
- python 3.x
  - The only function for input strings in python 3.x is *input* and it acts like raw_input of python 2.x

Immutability of strings

- Python strings are immutable
- Direct assignment to its items is not supported

  s = "abc"

  s[0] = "a" - ERROR
- There are a lot of functions for replacing, sub-stringing, etc. They all built and return the changed string

  s = "abac"

  s1 = s.replace("a", "A")

  print("s = {}, s1 = {}".format(s, s1)) #s = "abac", s1="AbAc"

String Built-in Methods

- There are many built-in string functions in python, here some the most common:
- Case functions:
  - s.capitalize() - Capitalizes first letter of string
  - s.lower(), s.upper() - returns the lowercase /uppercase version of the string
- Test functions:
  - s.islower(), s.isupper() - Returns true if all cased characters in string are lowercase/uppercase and false otherwise
  - s.isalpha()/s. isalnum()/s.isdigit()/s.isspace()... - tests if all the string characters are in correct state
  - s.startswith(suffix[, beg=0, end=len(string)])
  - s.endswith(suffix[, beg=0, end=len(string)]) - tests if the string starts/ends with the given *suffix*

String Built-in Methods – cont'd

```
s  = "abcd"
if s.islower():
    s = s.capitalize()                    # s = "Abcd“

    ---------- one more example ----------


if s.startswith("ab"):
        print( "starts with ab! “)          # "starts with ab!
if s.startswith("ab",1,len(s)):
        print( "starts with ab!")           # prints nothing, the
                                           # condition does not match
```

String Built-in Methods – cont'd

- Search and replace functions:
  - s. find(substr[, beg, end=len])/s.rfind - searches for *substr* in given string s and returns start index of the first/last appearance ,-1 if not found
  - s.replace(old, new[, max]) - returns a string where all/max occurrences of *old* have been replaced by *new*

- Example:

```
s = "python string ring"
sub = "ing"
ind1 = s.find(sub)              #ind1 = 10
ind2 = s.find(sub,12)           #ind2 = 15
s = s.replace(sub, "ong", 1)    #s = "python strong ring"
```

String Built-in Methods – cont'd

- Mix functions:
  - s.count(str, [beg,end]) - Counts how many times *str* occurs in string
  - s.strip() - returns a string with whitespace removed from the start and end
  - s.split(str [, num]) - Splits string according to delimiter *str* (space if not provided) and returns list of substrings; split into at most num substrings if given
  - s.join(seq) - Merges (concatenates) the string representations of elements in sequence seq into a string, with separator string

String Built-in Methods – cont'd

- Example:

```
s = "python+strings+example"
sub = "n"
cnt = s.count(sub)                    #cnt = 2
lst = s.split("+")                    #list =
["python","strings","example"]

str ="-"
lst = ["a","b","c","d"]
str = str.join(lst)                   #"a-b-c-d"
```

String format function

- Use format function to build formatted strings
- Curly-brackets are place holders. They are zero based

```
n1,n2 = 4,9
print("{0} + {1} = {2}".format(n1,n2, n1+n2))      # 4 + 9 = 13

print("{1} + {0} = {2}".format(n1,n2, n1+n2))      # 9 + 4 = 13
```

- Starting from python 2.7, the indexes can be omitted

```
print("{} + {} = {}".format(n1,n2, n1+n2))      # 4 + 9  = 13
```

- The same item can have multiple references

```
print("{0} is {0}".format(n1)             # 4 is 4
```

String format function – cont'd

- Place holders can be named

```
print("first value is {firstVal}, second value is {secondVal}".format(
   secondVal=1, firstVal=2))          #first value is 1, second value is 2
```

- Width and alignment can be specified

```
print("{0:<20}.".format("A"))        # A                   .
print("{:^20}.".format("A"))         #           A         .
print("{:>20}.".format("A"))         #                    A.
```

- Floating point precision can be specified

```
res = 10.0/3
print("{:.2f}".format(res))          # 3.33
```

Python Built-in function

- Python has a few built-in function, some of them work on sequences
- len(seq) – returns number of items in sequence-*seq*
- max(seq) – returns an item with maximal value in sequence-*seq*
- Etc

- Unpack assignment is also supported on sequences:

```
s = "klm"
c1,c2,c3 = s    # c1 = "k", c2 = "l", c3 = "m"
```

Combine Statements

- Multi-Line Statements:
  - Statements in Python typically end with a new line. Python does, however, allow the use of the line continuation character (\) to denote that the command should continue. For example:

```
total = item_one + \
        item_two + \
        item_threeStatements
```

  - Statements contained within the [], {} or () brackets do not need to use the line continuation character. For example:

```
days = ["Monday", "Tuesday", "Wednesday",
 "Thursday", "Friday"]
```

- Multiple-statements in one line
  - Use ";" to separate multiple statements in one code line.

```
n1 = 9; n2 = 11; n3 = -6
```

Python Lists

- The list is created using the square brackets [].

```
actors = ["Jack Nicholson", "Antony Hopkins",
 "Adrien Brody"]
```

- Python lists are mutable
- Python lists can hold mixed types

```
list = [11, "hello", -3.14, 23]
```

- List items can be accessed by index:

```
firstName = actors[0]          # firstName = "Jack Nicholson"
lastName = actors[-1]          # lstName = "Adrien Brody"
```

Python Lists Operators

+	Concatenation
*	Repetition
[]	Slice
[:]	Range Slice
in/not in	Membership
del	Delete List Elements Or entire list

```
l1 = [1,3,5,7]
l2 = l1 + [9,11]          # l2 = [1,3,5,7,9,11]
l3  = l1 * 2              # l3 = [1,3,5,7,1,3,5,7]
5 in l1                   # returns True
l1[2] = 6                 # l1 = [1,3,6,7]
print( l1[0:2])           # prints: [1,3]
del l1[2]                 # l1 = [1,3,7]
del  l3[1:7]              # l3 =[1,7]
```

Python Lists Operators – cont'd

```
values = [1, 3, 5, 7, 9, 11, 13]
print( values[:])          # prints all, like print(values)
print( values[1:])         # prints all except first element
print( values[:1] )        # prints only first element
print( values[2:4])        # prints values at index 2 and 3

del values[:]          # deletes all list values, values = []
del values             # undefined list, list doesn't exists now
print( values)         # generates an error, the list is
                       # longer exists
```

Python Lists Methods – cont'd

- Add/ Remove elements:
  - list.append(obj) – appends *obj* to the list
  - list.extend(seq) – appends *seq* to the list
  - list.insert(index, obj) – inset *obj* to the list at *index* index
  - list.pop() - removes and returns last object from list
  - list.remove(obj) – removes *obj* from list

```
lst = ["a","b","c","d"]
lastElm = lst.pop()          # lastElm = "d"
print( lst)                  # lst = ["a","b","c"]
lst.insert(2,"new")          # lst = ["a","b","new","c"]
```

Python Lists Methods – cont'd

- Reorganize functions:
  - list.sort() – sorts list items, IN PLACE
    - Sort function can receive call-back to specify custom sort order and attributes (will be discussed later)
  - list.reverse() – reverse the order of list items, IN PLACE
  - list.index(obj) – returns first index of obj in list, -1 otherwise
  - list.count(obj) – returns the number of times obj appears in list
- Example:

```
lst = ["c","b","a","d"]
lst.reverse()                #lst = ["d", "a","b", "c"]
lst.sort()           #lst = ["a","b","c","d"]
```

Iterating Lists

- Iterating values

```
for elm in li:
        print( elm)
```

- Iterating indexes:

```
li = ['a', 'b', 'c', 'd', 'e']
for i in range(len(li)):
        print( li[i])
```

Lists comprehension

- Lists comprehension used to construct new list from existed sequence in a very natural, easy way

```
l1 = range(1,11)                    #l1 = [1,2,3,4,5,6,7,8,9,10]
list = [i*2 for i in l1]            #list = [2,4,6,8,10,12,14,16,18,20]
l2 = ["ab", "cd", "xyz"]
print ([str(x) + str(x)[::-1] for x in  l2]) # ["abba", "cddc", "zyzzyx"]

L3 = [char for char in "python"]        #l3 = ['p', 'y', 't', 'h', 'o', 'n']
l4  = [char for string in l2 for char in string]    # ['a', 'b', 'c', 'd', 'x', 'y', 'z']
```

Python Tuples

- A tuple is index based sequence, just like list.
- Python tuples are enclosed in parentheses ( )
- Tuples are immutable and cannot be updated.
- Tuples can be thought of as read-only lists
- Tuples they more effective that lists

- For example:

```
tuple = ( "abcd", 786 , 2.23, "john", 70.2 )
```

Tuples can't be changed

```
tup = ("a", "b", "c", "d")

del tup[1]          # generates an error
tup[0] = "e"        # generates an error

But reassignment is supported:
tup = (1, 2, 3)     #correct
```

Tuples Operators

- Tuples have the same operators as lists and they behave the same:

  +, *, [], [:], in/not in, Unpack assignment

Example 1:

```
tup = (1, 2, 3, 4, 5, 6, 7 )
resTup = tup[1:5]            # resTup = (2,3,4,5)
```

Example 2:

```
tup1 = ("12", "234")
tup2 = ("34", "567", "8")
tup3 = tup1 + tup2           # tup3 = ("12", "234", "34",
"567", "8")
```

Python Dictionary

- Python's dictionaries are kind of hash table that can be found in lots of different programming languages
- Dictionaries consist of key-value pairs.
- Dictionaries are enclosed in curly braces { }
- Dictionary values have no restrictions. They can be any built-in or user-defined type
- There are two important points to remember about dictionary keys:
  - Keys must be immutable (for built ins) or hashable (for user-defined)
  - no duplicate key is allowed. When duplicate keys encountered during assignment, the last assignment wins
- A dictionary is mutable type

Python Dictionary – cont'd

- Python dictionaries have they own internal keys organization, this is why the order of its items(pairs) is unpredictable and should be treated like "random"

```
dict = {'Name': 'Zara', 'Age': 7, 'Name': 'Manni'}
print(dict)          #prints dict = {'Age': 7, 'Name': 'Manni'}
```

- Values can be assigned and accessed using square braces [] for keys
  - When non-existed key is assigned with a value, this key-value pair is added to dictionary
  - When existed key is assigned with a value, its value is updated
  - When access non-existed key, the error is generated

Python Dictionary – examples

- Create and Updating dictionary
  - Way 1: step by step creation

```
dict = {}                          # empty dictionary
dict["one"] = "value of one"       # add new pair: "one"- "value of one"
dict[2] = "value of 2"             # add new pair: 2- "value of 2"
```

  - Way 2: all at once creation

```
dict1= {"name": "john", "lastName":"Smith", "age": 33}
dict1["age"] = 23                  # updating value at key "age"
print( dict1["AGE"])               # generats KeyError
```

Python Dictionary – cont'd

- Delete Dictionary Elements:
- You can either remove individual dictionary elements, clear the entire contents of a dictionary or delete entire dictionary in a single operation.

```
dict= {"name": "john","lastName":"Smith", "age": 33}
del dict["name "]          # remove entry with key "name "
dict.clear()               # remove all entries
del dict                   # delete entire dictionary
```

Python Dictionary – cont'd

- The following methods defined for dictionary:

len(dic) – returns the number of key-value pairs

dict.clear() - removes all elements of dictionary *dict*

dict.get(key,[default for non-existed key]) - for *key* key, returns its value or *default* if *key* doesn't not exists. Default value for *default* is None

dict.items() -returns a list of *dict*'s (key, value) tuple pairs

dict.keys() -returns list of dictionary *dict's* keys

dict.values() -returns list of dictionary *dict*'s values

Python Dictionary – cont'd

```
dict= {"name": "john", "lastName":"Smith", "age": 33}
print( dict["AGE"])          # generates KeyError
print( dict.get("AGE"))      # prints None
print( dict.get("AGE", -1))  # prints -1

print( dict.keys()           # prints ['lastName', 'age', 'name']

print( "Key-Values pairs : {}".format(dict.items())
The Output:
Key-Values pairs : [('lastName', 'Smith'), ('age', 33), ('name',
'john')]
```

Iterating Through a Dictionary

- d = {'x': 1, 'y': 2, 'z': 3}

- Iterating keys:

```
for key in d.keys():
    print( key, d[key])
```

Iterating items:

```
for k, v in dict.items():
    print( "{}, {}".format (k, v))
```

The Output:

y, 2

x, 1

z, 3

Iterating Through a Dictionary – cont'd

- Iterating values:

```
dict= {"name": "john", "lastName":"Smith", "age": 33}

print( "\n".join(["%s=%s" % (k, v) for k, v  in dict.items()]))
```

The output:

```
lastName=Smith
age=33
name=john
```

Dictionary comprehension

- Dictionary comprehension supported in python starting from python 2.7

```
keys = [1,2,3]
values=[4,5,6]
dict = {keys[i]:values[i] for i in range(len(keys))}
print(dict)                                    # {1: 4, 2: 5, 3: 6}

d1 = {k: 0 for k in ['a','b','c']}             # {'a': 0, 'c': 0, 'b': 0}
d2 = {n: n**2 for n in [10, 11, 12]}           #{10:100, 11:121, 12:144}
```

Sets

- Set is unordered collections of unique elements
- Sets are dictionary based, it is collections of keys without values
- All the restrictions for dictionary keys applied on set's values
- Common uses:
  - Removing duplicates from a sequence
  - Intersection
  - Union
  - Difference
  - etc
- Use {...} to define set
- Use set() to define an empty set

```
st = {1, 77 ,4, 81, 10, 77, 4}
print(st)                # { 81, 10, 1, 4, 77}
```

Set's functions

st.issubset(t)	test whether every element in st is in t
s.issuperset(t)	test whether every element in t is in st
s.union(t)	return new set with elements from both s and t
s.intersection(t)	return new set with elements common to s and t
s.difference(t)	return new set with elements in s but not in t
s.symmetric_difference(t)	return new set with elements in either s or t but not both
s.update(t)	update set s with elements added from t
s.difference_update(t)	update set s after removing elements found in t
s.add(x)	add element x to set s
s.discard(x)	removes x from set s if present
s.clear()	remove all elements from set s

Set's functions examples

```
items = set()
items.add("cat")
items.add("dog")
items.add("gerbil")
print(items)          # {'gerbil', 'dog', 'cat'}

s1 = {1,2,4,5,6}
s2 = {2,3,5,7,8}
s2.update(s1)
print (s2)            #{1, 2, 3, 4, 5, 6, 7, 8}
```

Set's functions examples – cont'd

```
s1 = {1,3,4}
s2 = {4,5,6}
s3 = {4,3,2,1,0}

print(s1.issubset(s3))                  #  True
print(s1.issubset(s2))                  #  False
print (s1.union(s2))            # {1, 3, 4, 5, 6}
print (s1,s2)                           # {1, 3, 4} {4, 5, 6}
print (s1.intersection(s2))             # {4}
print (s1.difference(s2))               # {1, 3}
```

Mutable vs Immutable in Python

- An immutable variable is an object whose value can't be modified after it is created
- If change of immutable variable is supported, the change will always create a new, updated value
- Python immutable type are: None, bool, int, long (if exist), complex, string, tuple
- Python mutable types are: list, dictionary, set, classes

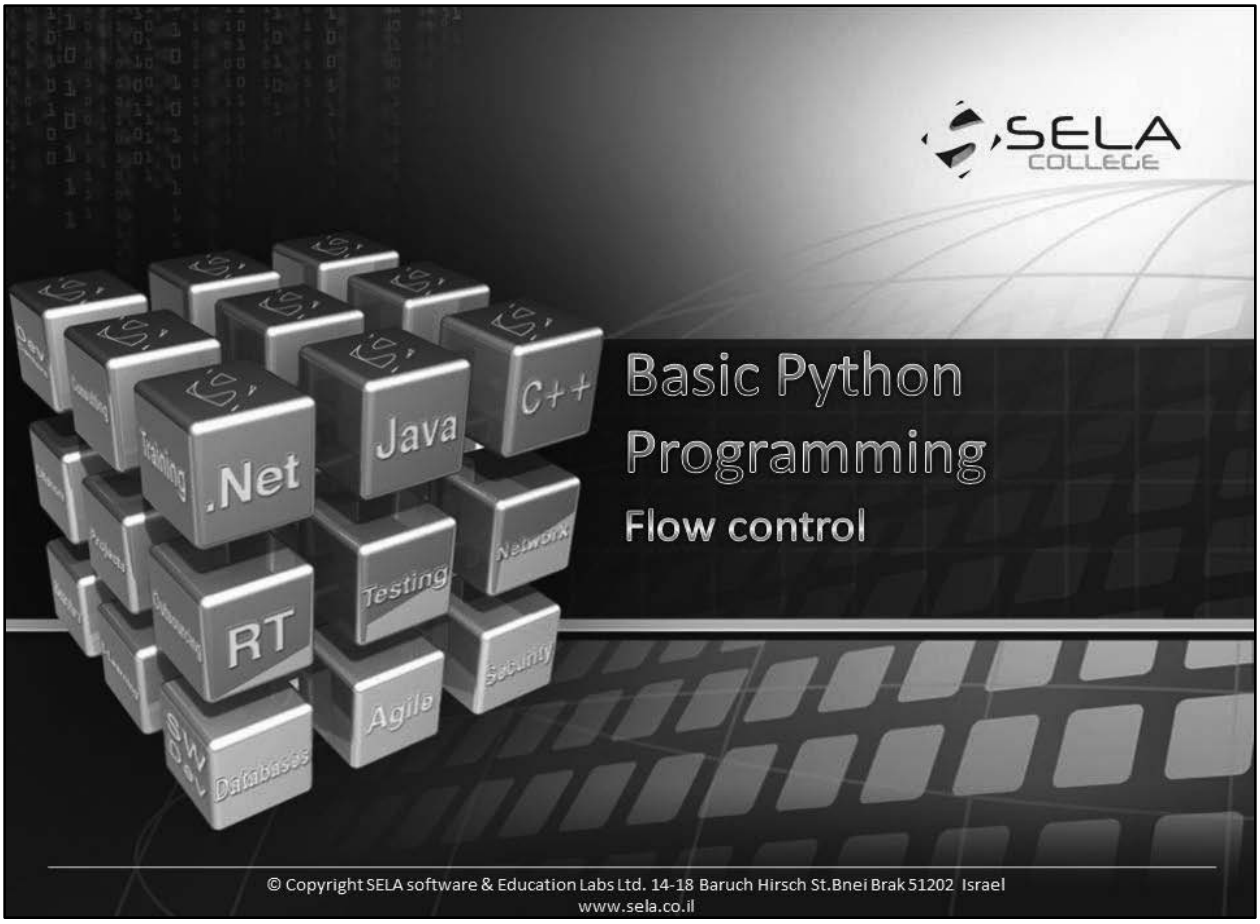
Mutable vs Immutable in Python – cont'd

- Look at the following code:

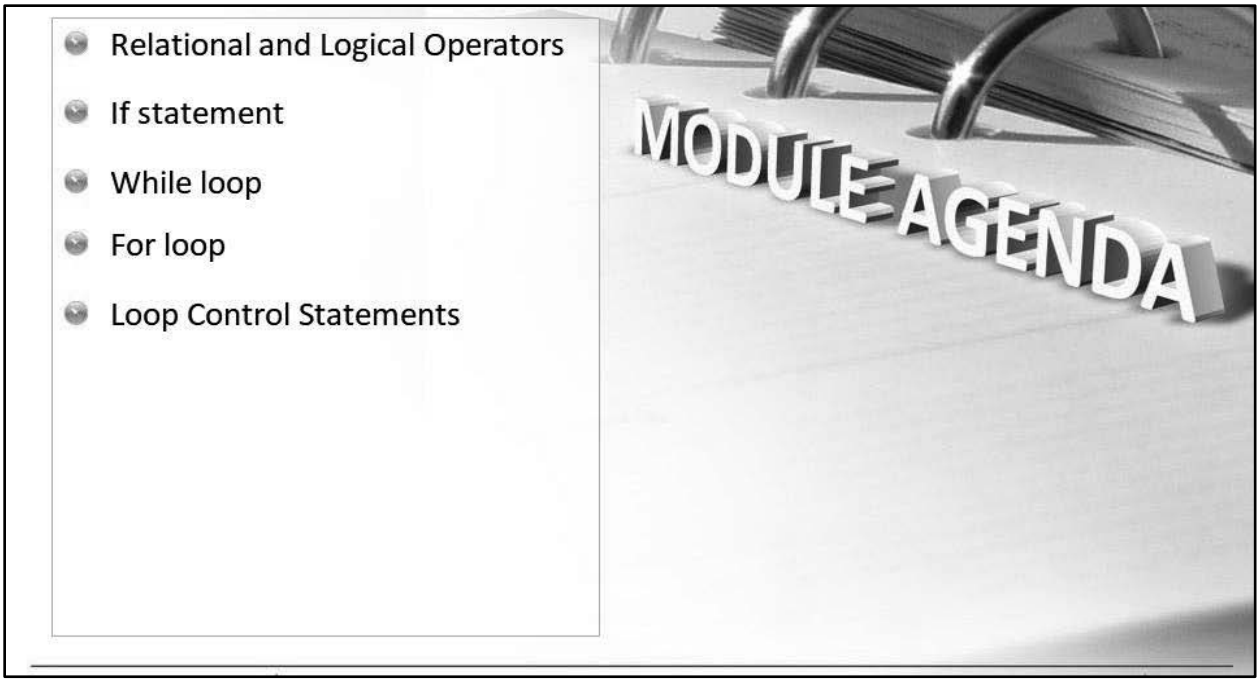
```
l1 = [11, 22, 33]
l2 = l1;
l1.append(44)
print( l2)
```

What is the output for this code?

Basic Python Programming



Agenda



Relational and Logical Operators

- Logical Operators:
  - and - Logical AND
  - or - Logical OR
  - not - Logical NOT
- Two sets of relational operators:

Comparison	Numeric
Equal	==
Not equal	!=
Less than	<
Greater than	>
Less than or equal to	<=
Greater than or equal to	>=

If statment

```
if statement1:
        ...
elif statement2:
        ……..
else:
        ……
```

- *elif* and *else* statements are optional
- Example:

```
number = 10
if number %2 == 0:
   print ("The number is even“)
else:
      print ("The number is odd“)
```

while loop

- Python while loops are used for repeating sections of code - but unlike a for loop, the while loop will run as long as defined condition is met.
- while expression:
  statement(s)

Example 1:

```
count = 0
while  count < 9:
        print ('The count is:', count)
         count += 1
```

Example 2:

```
n = raw_input("Please enter 'hello':")
while n.strip() != 'hello':
      n = raw_input("Please enter 'hello':")
```

For Loop

- Python for loops iterates over the member of a sequence in order

```
Example 1:
for letter in 'sentence':
    print (letter)                    #prints:  s e n t e n c e

Example 2:
for num in range(10,20):              #prints 10, 11, 12 ...19
    print(num)

Example 3:
for num in range(10,21, 2):           #prints 10, 12, 14 ...20
    print(num)
```

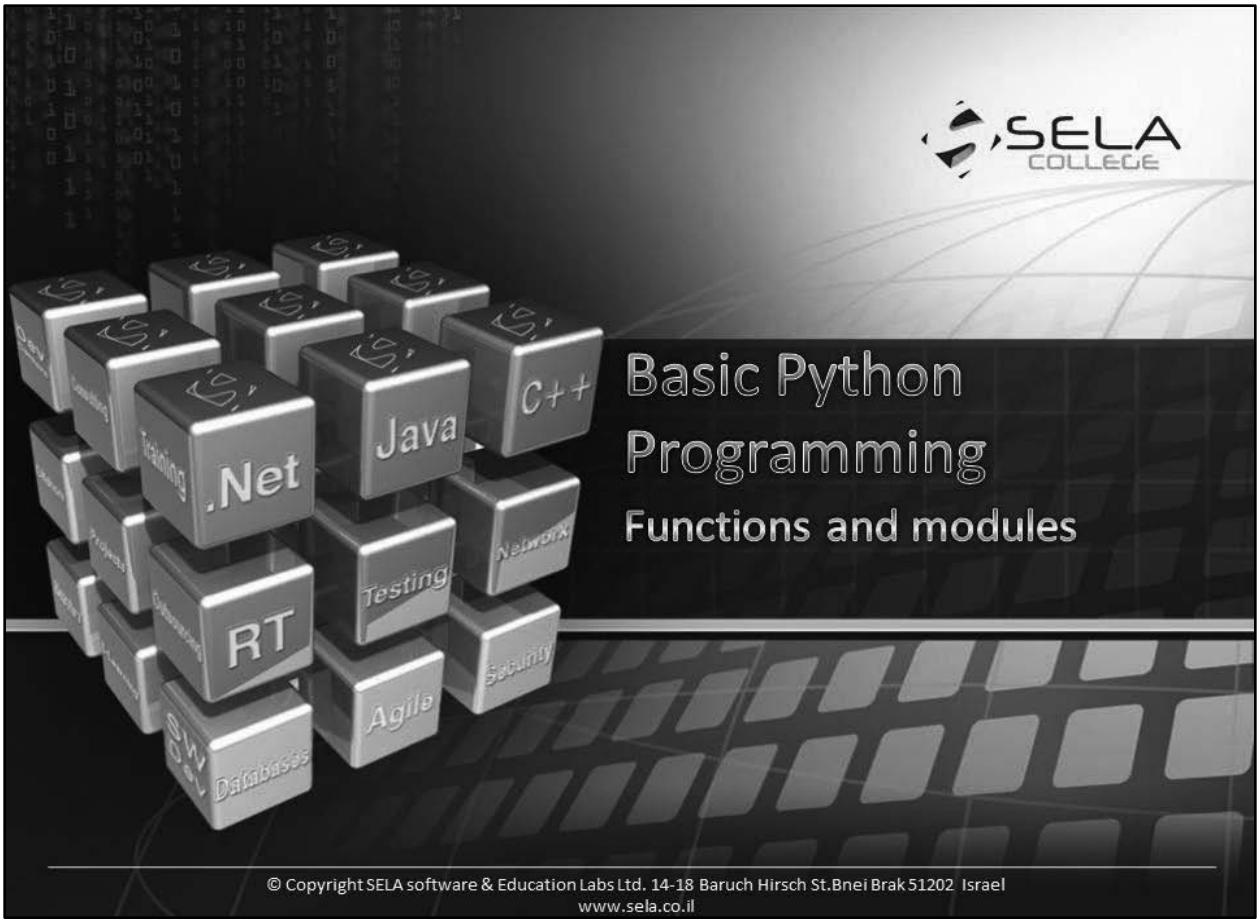
Loop Control Statements

- break
  - The **break** statement in Python terminates the current loop and resumes execution at the next statement, just like the traditional break found in C.
- continue
  - The **continue** statement in Python returns the control to the beginning of the while loop.
- pass
  - The **pass** statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.
  - The **pass** statement is a *null* operation; nothing happens when it executes. The **pass** is also useful in places where your code will eventually go, but has not been written yet

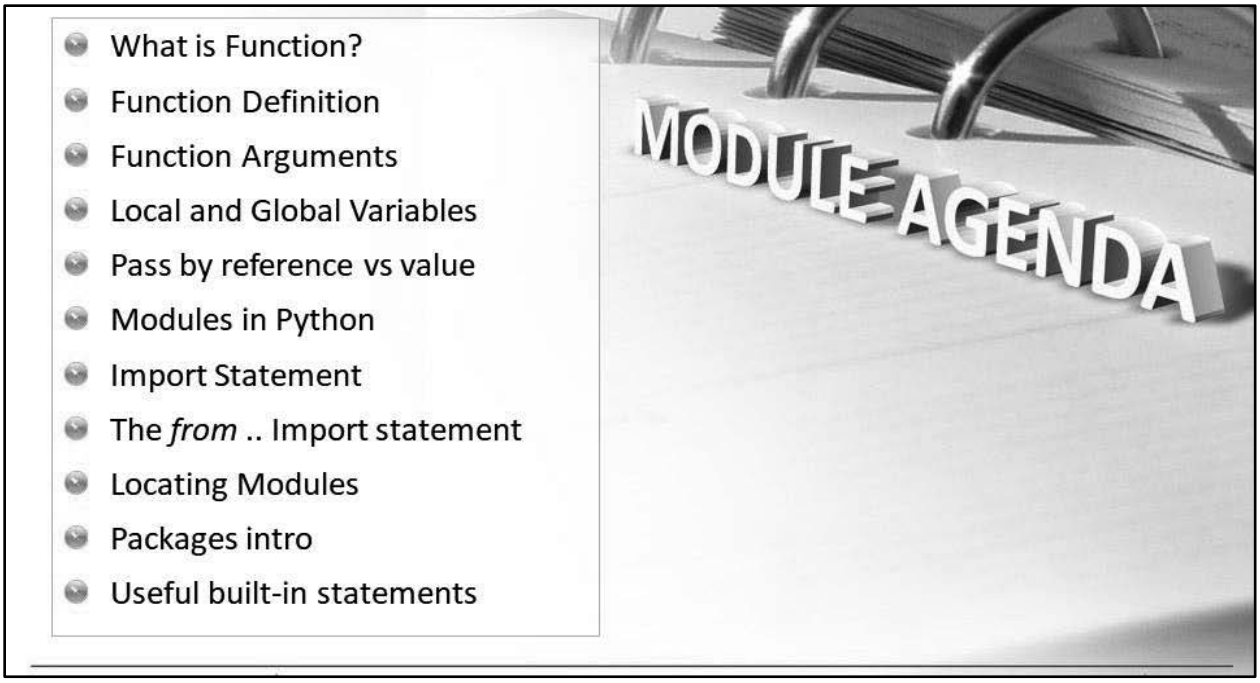
Loop Control Statements – cont'd

```
import math

i = 5
print("I will print the square root of 5 non-negative numbers (0 to exit)“)

while i:
        s = raw_input("Enter the next number => ")
        num = int(s)
        if num == 0:    #user wants to exit
                break
         if num < 0:
                continue
        i-=1
        print(math.sqrt(num))
```

Basic Python Programming



Agenda



What is a Function?

- Function is a small sub program that perform a single task
- Functions provide better modularity and high code reuse
- Python has many build-in functions but user can define his own function
- Function definition starts with *def* keyword, function name and then list of parameters in () parentheses, : at the end
- Functions in python can use return statement.
  - return can come with expression, the one that we want to return
  - return can be empty
- return exits the function.

Functions documentation strings

- First function statement can be the function documentation called *docstring.*
- Python documentation string is used like a comment and provide a convenient way to access this documentation even in run time
- Python docstrings can be placed in functions, classes, modules
- In python docstring can be accessed with *help* function or *obj.__doc__*

```
help(myFunc)          prints myFunc function docString
s = myFunc.__doc__    returns prints myFunc function docString
```

Function Definition

```
def funcName(parameters):
        "docstring"
        command
        .
        .
        command
        [return expr]

For example:

def simple_print(str):
        "function prints value of parameter"   <-- docstring
         print ("parameter is: ", str)

simple_print("Python functions is great!!")     <-- Function call
```

Function overloading

- Python does not support function overloading
- When function with the same name defined more than once, they are treated as function redefinition and last one wins
- Python has many different ways to overcome this obstacle, like default arguments, variable-length parameters, etc (will be discussed later)

Function Arguments

- Required arguments:
  - Required arguments must be passed to a function in correct order and exact amount.

For example:

```
def print_data(color, size):
        print ("color: ", color, " and size : ", size)
print_data("Red", 4)          # prints color: Red and size : 4
print_data("Blue")            #generates arguments count error
```

Function Arguments – cont'd

- Keyword arguments:
  - Arguments can be pass by keyword (name). We can pass arguments in any order when passing them by names.
  - Non-keyword arguments are allowed after keyword ones

Example 1:

```
print_data(size=6,color="Blue")
print_data(color="Blue", size=6)
```

The output for both is: color: Blue and size : 6

Example 2:

```
def func(x=1,y=2):
  print(x,y)

func(2,6)        #prints 2,6
func(2)          #prints 2,2
func(x=2,6)      # generates the error
```

Function Arguments – cont'd

- Default Arguments:
  - A default argument is an argument that defines a default value. The argument holds this default value, unless other value is passed
  - If function have both default and formal arguments, formal arguments must be defined first.

```
def print_data(color, size=10):
        print("color: ", color, " and size : ", size)

print_data("Yellow")                    # color: Yellow and size : 10
print_data(size = 3, color="Yellow")     color: Yellow and size : 3
```

Function Arguments – cont'd

- Variable-Length parameters:
  - The special syntax, *args and **kwargs in function definitions is used to pass a variable number of arguments to a function.
  - The single asterisk form (*args) is used to pass a *non-keyworded*, argument list, that passed as tuple
  - The double asterisk form (**kwargs) is used to pass a *keyworded*, variable-length argument list, that passed as name-value dictionary

Function Arguments – cont'd

Variable-Length *args example:

```
def calc_aver(*args)
   sm = 0
   for elem in args:
        sm+=elem
   return float(sm)/len(args)

aver = calc_aver(1,2,3,4);
print (aver)                          #prints 2.5
aver = calc_aver(1,2,3,4,52);
print(aver)                           #prints 12.4
```

Function Arguments – cont'd

Variable-Length **kwargs example:

```
def print_named_variables(title, **dict):
    print ("title: ", title)
    cnt = 1
    for varName,varValue in dict.items():
        print ("{}) {}={}" .format(cnt, varName, varValue))
        cnt+=1

print_named_variables("all values", x=1,y="abc",z=-3.5)
```

```
title:  all values
        1) y=abc
        2) x=1
        3) z=-3.5
```

Local and Global Variables

- The scope of a variable determines where it can be accessed.
- Variables that are defined inside a function body have a local scope. Local variables can be accessed only inside the function in which they are declared
- Variables that are defined outside any function have a global scope. Global variables can be accessed throughout the program body by all functions.

Local and Global Variables – cont't

```
def f():
    s = "I am globally unknown"    # define local variable s
    print (s)                      # prints "I am globally not known"


f()
print(s)                           # generates  error
```

Local and Global Variables – cont'd

Example 1:

```
def func():
        print ("in func: ", glob)

glob = 10

func()
print ("in main space: ", glob)
```

The Output:
in func: 10
in main space : 10

Example 2:

```
glob = 10

def func():
   glob=11
   print ("in func: ", glob )


func()
print ("in main space: ", glob)
```

The Output:
in func: 11
in main space : 10 – Why??

Local and Global Variables – cont'd

Lets add one new line code to the *func* function from the previos example:

```
glob = 10

def func():
   print ("in func: ", glob)
   glob=11
   print ("in func: ", glob )

func()
print ("in main space: ", glob)
```

What will be the output now?

Python "assumes" that we want a local variable due to the assignment to *glob* inside of func(), so the first print statement throws this error message.

Local and Global Variables – cont'd

- Any variable which is changed or created inside of a function is local unless it hasn't been declared as a **global** variable.

```
glob = 10

def func():
   global glob
   print ("in func: ", glob)
   glob=11
   print ("in func: ", glob)

func()
print ("in main space: ", glob)
```

Pass by reference vs value

- All parameters in the Python language are passed by reference.
- If we change what a parameter refers to within a function, the change also reflects back in the calling function. For example:

```
def chang_list( lst ):
   "This changes a passed list into this function"
   lst.append(4)
   print ("inside the function: ", lst)

list = [1,2,3]
chang_list(list )
print ("outside the function: ", list)

The Output:
inside the function:  [1, 2, 3, 4]
outside the function:  [1, 2, 3, 4]
```

Pass by reference vs value – cont'd

```
def chang_list( lst ):
   "This changes a passed list into this function"
   lst = [4]
   print ("inside the function: ", lst )

list = [1,2,3]
chang_list(list )
print ("outside the function: ", list)

The Output:
inside the function:  [4]
outside the function:  [1, 2, 3]
```

Pass by reference vs value – cont'd

```
def chang_string( s ):
    s = "is fun"
    print ("inside the function: ", s)

s = "python"
chang_string(s)
print ("outside the function: ", s)

The output:
inside the function:  is fun
outside the function:  python
```

Modules in Python

- As your program gets longer, you may want to split it into several source files for easier maintenance
- File that contains definitions only can be treated as module and can be used in python programs.
- Module can define functions, classes and global variables
- Definitions from a module can be *imported* into python program and other modules

Import Statement

- *import moduleName* imports all the *moduleName*'s definitions under namespace

  import myModule
- The default namespace name is module name.
- Namespace name can be changed with *as* statement in import line

  import myModule as MM
- All module's definitions can be accessed through namespace only

  myModule.myFunc(…)

Import Statement - example

- Lets say we have utils.py file with 2 function:
  - def doSomthing1():

    ….
  - def doSomthing2():

    ….

```
import utils
utils.doSomthing1()
```

The from...import Statement

- Python's *from* statement lets you import specific attributes or all attributes from a module into the current namespace.
- The *from...import* has the following syntax: from moduleName import name1, name2, ... nameN

```
from utils import doSomthing1
from utils import *
```

- Namespaces are omitted using *from...import* add only direct access is possible

```
doSomthing1()
utils.doSomthing1()          #generates error
```

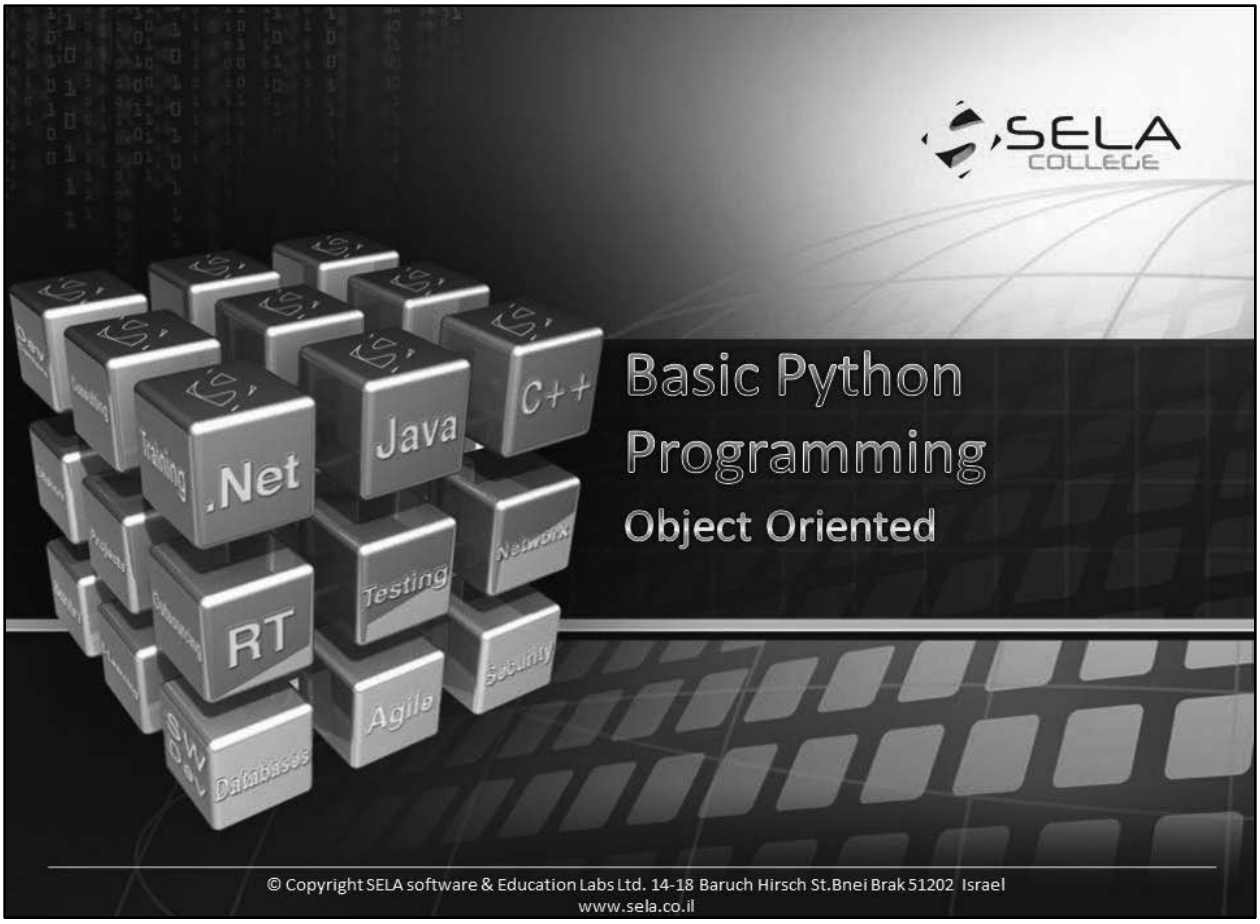
Locating Modules

- When importing modules the interpreter searches:
  - First interpreter searches in the current directory.
  - If the module isn't found, Python then searches each directory in the shell variable PYTHONPATH
  - If the module isn't found, the interpreter searched in standard installation path. (In Linux/Unix for example in /usr/local/lib/python)

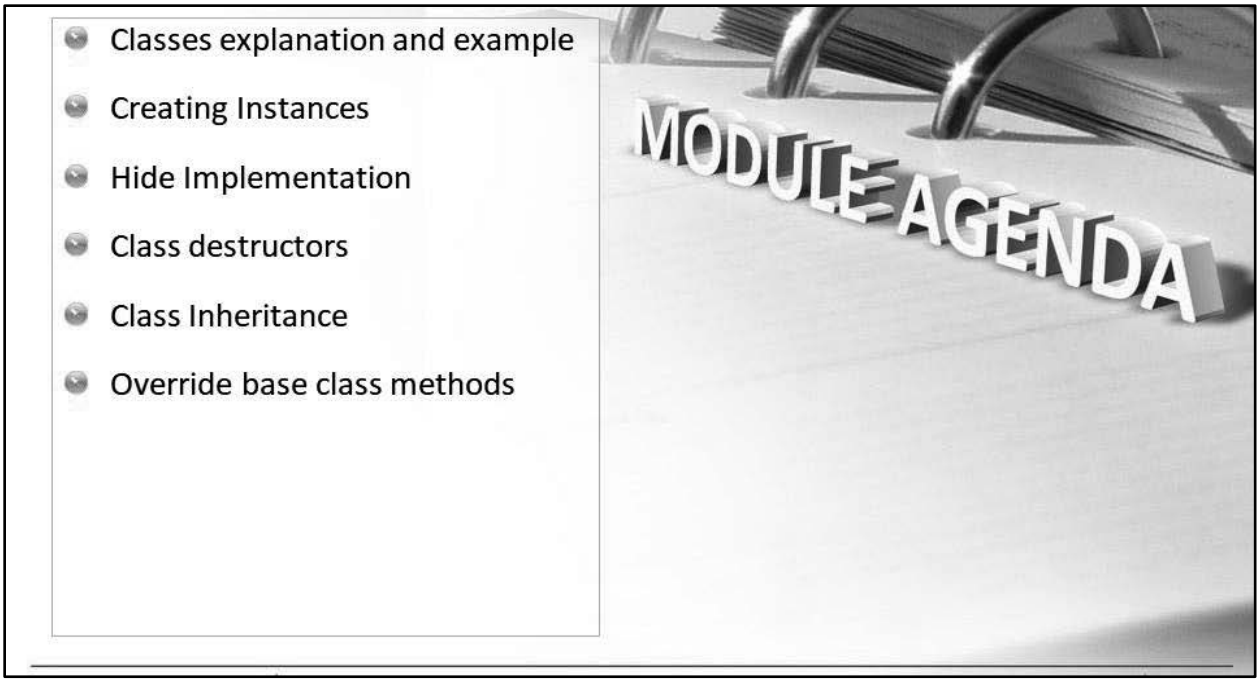
Python packages

- To help organize modules and provide a naming hierarchy, python has a concept of packages
- The __init__.py files are required to make Python treat the directories as package
- Access to modules inside a package through the full name: *packageName.ModuleName*
- Packages can contains sub-packages

Basic Python Programming



Agenda



class

- Python defines a set of predefined types of objects, like int, string, list, method etc.
- User can define its own, user defined type of object using *class* keyword
- The name of the class immediately follows the keyword *class* followed by a colon. The name of class is a new user defined type:

```
class Test:
        'Optional class documentation string'
        pass
```

class – cont'd

- Python classes can contain different attributes, like methods, data members, docstring, etc
- classes may define special methods, with predefined names and meaning and format like __XXX__.
  - They usually used for operators overloading and built-ins overriding
  - They are automatically invoked
  - For example:
    - __init__ - responsible for class instantiation for the newly-created class instance
    - __str__ - returns string representation of an object

self in Class's methods

- Python implements methods in a way that makes the instance, to which the method belongs, be *passed* automatically, but not *received* automatically
- The first parameter of methods is the instance the method is called on
- This parameter usually called *self*

class BankAccount Example

```
Lets create Bank Account class
class BankAccount (object):
    commission = 5.40            # class variable shared by all instances
    def __init__(self, clientName, clientId, balance):
        self. _clientName = clientName
        self. _clientId = clientId
        self. _balance = balance

    def  withdraw (self, amount):
        if self._balance – amount > 0:
                self. _balance -= (amount + BankAccount.commission)
                return True
        return False
```

instance variables unique to each instance

class BankAcount – cont'd

```
    def  deposit(self, amount):
        self. _balance += amount - BankAccount.commission
        return True

    def  __str__(self):
        return "client: {} has {} $".format (self. _clientName,
                        self. _balance);
```

class BankAcount – explanation

- The variable *commission* is a class variable whose value would be shared among all instances of a this class – static attribute
- It can be accessed as *BankAccount.commission* from inside or outside the class.
- The first method __init__() is a special method automatically called by python to initialize newly-created class instance

Creating Instances

- To create a new instance of a class simply specified a class name with all its parameters and assigned the result to some variable

ba = BankAccount("Tal Moshe", 12345678, 1000)

- This statement invokes the __init__ method
- Now we can access all the instance attributes

```
ba.deposit(500)
print(ba)                    #prints Tal Moshe has 493.60 $"
```

Hide Implementation

- Python defines private attributes by convention
- Attributes, whose name starts with an underscore (e.g. _spam) should be treated as a non-public and should be never accessed outside the class
- Attributes, whose name starts with two leading underscores (e.g. __spam) will be treated as strongly private (Python simply changes their names in outside class access)
- It is still possible to access or modify a variable that is considered to be non-public

classes clean-up

- Since the memory in python is managed, destruction/cleanup is usually needed for resources
- One of the ways to manage object cleanup is by defining the __del__() method
- The __del__() is called when the instance is about to be destroyed by GC

```
class File:
    def __init__(self):
        self._fileObject = open("some file")
        # ...
    def __del__(self):
        self._ fileObject.close()
```

classes clean-up – cont'd

- The problem with __del__ is that it will be called at unpredictable time (if ever) for objects with circular referencing

```
class Foo:
    def __init__(self, x):
        self.x = x
        # x => bar instance

    def __del__(self):
        print ("end of Foo")
```

```
class Bar:
    def __init__(self):
        self.foo = Foo(self)

    def __del__(self):
        print ("end of Bar")
```

```
bar = Bar()
foo = Foo(bar)
bar = None  #will not call the __del__
```

classes clean-up – cont'd

- The better solution for object clean-up and the recommended one is to add to the class support of context manager (*with* statement)
- Not like __del__() it has no side effects
- To use the with statement, create a class with the following methods:
  - __enter__
  - __exit__

Class Inheritance

- When we need to extend the existed class functionality and to add an extra features with a smart reuse of existed class – the solution is inheritance
- In inheritance, the class that performs the inheritance called derived class and the one who we inherits (extends) from - called base class
- The child class inherits all attributes of its parent class
- A derived class can override any method of its base class, and a method can call the method of a base class with the same name

```
class derivedClass (BaseClass1 [, BaseClass2, ...]):
    'Optional class documentation string'
    commands
```

Class Inheritance – cont'd

- Student Bank Account example:

```
class StudentBancAccount(BankAccount):
    def __init__(self, clientName, clientId, balance, collegeName)
        BancAccount.__init__(self, clientName, clientId, balance)
            sela.collegeName = collegeName

        .
        .
        def __str__(self):
            return "{} {}".format (BancAccount.__str__(self),
            self.collegeName
```

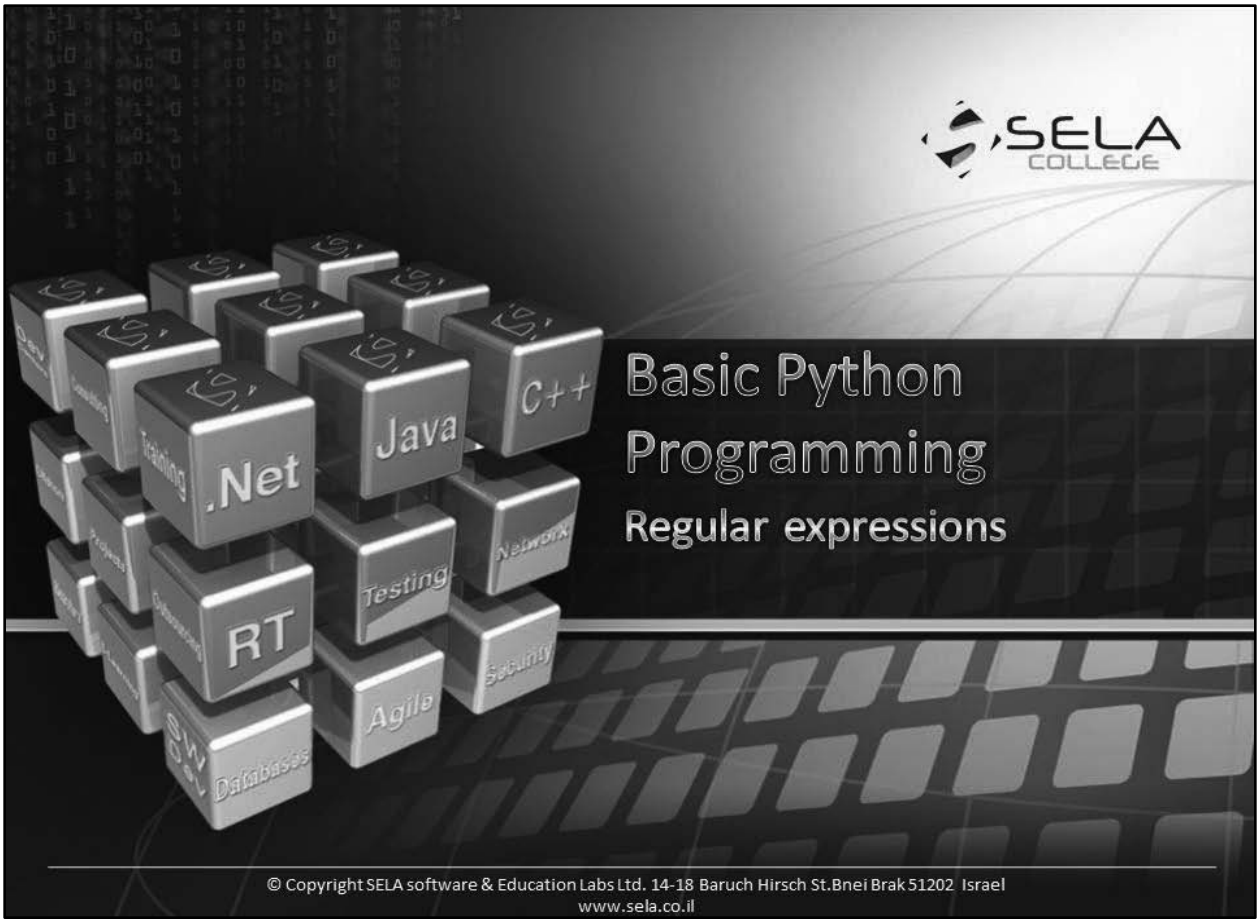
Class Inheritance – cont'd

- Base method overriding can be done using *super:*

```
class StudentBancAccount(BankAccount):
        def __init__(self, clientName, clientId, balance, collegeName)
                super(StudentBancAccount, self).__init__(
clientName, clientId, balance)
                sela.collegeName = collegeName
```

- Note:
  - super() was introduced in version 2.7
  - you can only call super() if one of the parents inherit from a class that eventually inherits object.
  - In python 3.x super syntax is much easier: super().__init__(….)
- *isinstance(obj, type)* - return true if the *obj* argument is an instance of the *type* argument, or of a subclass thereof.

Basic Python Programming



Concepts About Regular Expressions

- A *regular expression* is a pattern - a template - to be matched against a string.
- Matching a regular expression against a string either succeeds or fails.
- Sometimes, the success or failure may be all you are concerned about and sometimes we to process or to replace the matched pattern.
- Regular expressions are widely used by many programs and languages
- The module **re** provides full support for regular expressions in Python

Regular- expression characters

- There is the basic set of regular-expression meaningful characters in Perl.

Character	The character meaning	Example
^	Match the beginning of the line	^a
$	Match the end of the line (or before newline at the end)	a$
.	Match any character (except newline)	… ^…$
[ ]	Character class	[aeiouAEIOU] [a-zA-Z0-9_] [^0-9]
|	Alternation	abc|123
()	Grouping	(abc)+
\	Quote the next metacharacter	^\.

Regular- expression characters - Cont'd

- There is the basic set of quantifiers characters:

Character	The character meaning	Example
*	Match 0 or more times	^ab*c$
+	Match 1 or more times	^[A-Z]+
?	Match 1 or 0 times	[.?!]?$
{n}	Match exactly n times	.{20}
{n,}	Match at least n times	^A.{20,}
{n,m}	Match at least n but not more than m times	^[0-9]{4,9}$

Regular- expression characters - Cont'd

- There is the extended set of Python characters:

Character	The character meaning	Example
\w	Match a "word" character (alphanumeric plus "_")	^\w{5}$
\W	Match a non-"word" character	^\W.*\W$
\s	Match a whitespace character	\s
\S	Match a non-whitespace character	^\S+$
\d	Match a digit character	\d$
\D	Match a non-digit character	^\D

re search

re.search - Scan through string looking for the first location where the regular expression pattern produces a match, and return a corresponding MatchObject instance

matchObj = re.search(pattern, string, flags=0)

pattern – regular expression

string – string to look *pattern* into

flags – possible flags

matchObj will be None if pattern didn't match

re Matching Object

- matchObj has the following attributes:
  - matchObj.group(i) - returns matched group number *i* (one-based)
  - matchObj.start(i) - returns the offset in the string of the start of group number *i*
  - matchObj.end(i) returns the offset of the character beyond the match of group number *i*

You can use the m.start() and m.end() to slice the string:
string[m.start(i):m.end(i)]

re search

```
import re

line = "my age is 22"
m = re.search(r'(\d+).*',line)
if m:
        print("matched string is {} in index ({},{})".format(
         (m.group(1), m.start(1), m.end(1))
else:
        print ("No match!!")
```

re search – cont'd

```
import re

line = "27:11:2004"
m = re.search(r'(\d+):(\d+):(\d+)', line)
if m:
        print ("matched day is {} in index({},{})".format(
                m.group(1), m.start(1), m.end(1))
        print ("matched month is {} in index({},{})".format(
                m.group(2), m.start(2), m.end(2))
        print ("matched year is {} in index({},{})".format(
                m.group(3), m.start(3), m.end(3))
else:
        print ("No match!!“)
```

re sub

- **re.sub** – replaces all (or max) occurrences of the pattern in string. This method would return modified string

- re.sub(pattern, repl, string, max=0)

  pattern – regular expression

  repl – replacement string

  string – string to look *pattern* into

  max – maximum replacements

re sub – cont'd

```
import re
phone = "2004-959-559"

 # Remove anything other than digits
newPhone = re.sub(r'\D', "", phone)
print ("Phone num now is : ", newPhone )
#Phone num. now is : 2004959559

# Replace '-' with space
newPhone = re.sub(r'-', " ", phone)
print ("Phone num now is : ", newPhone)
#Phone num now is :  2004 959 559
```

re split

- **re.split** - Split string by the occurrences of pattern

re.split(pattern, string, maxsplit=0, flags=0)

pattern – regular expression

string – string to look *pattern* into

maxsplit – maximum splits

flags – possible flags

re split – cont'd

```
import re
value ="one is 1, two is 2"
result = re.split("[, ]+", value)

for element in result:
 print(element)
```

The output

```
one
is
1
two
is
2
```

re split – cont'd

```
value = "one 1 two 22 three 3"
result = re.split("\D+", value)

for element in result:
    print(element)
```

The output

```
1
22
3
```

re finditer

- **re.finditer** - Return an MatchObject iterator for all matched patterns in string

```
re.finditer(pattern, string, flags=0)
        pattern – regular expression
        string – string to look pattern into
        flags – possible flags
```

re finditer – cont'd

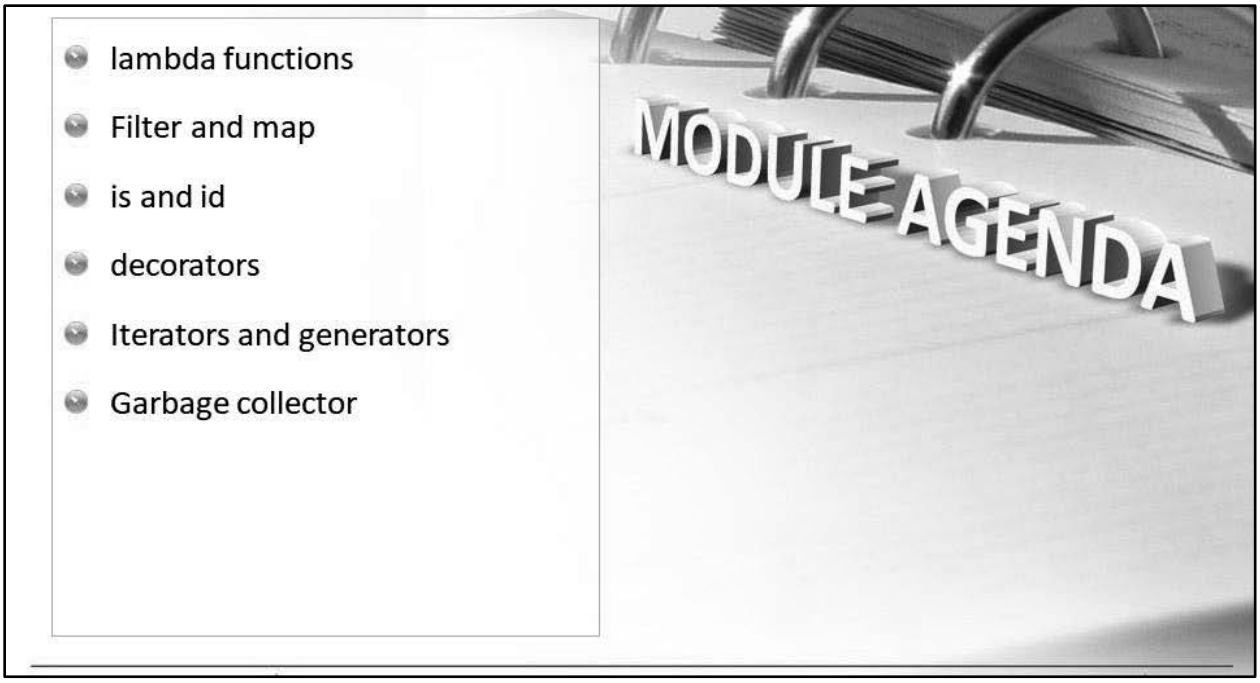
```
import re
text = "this is a long sentence with a lot of words"
for m in re.finditer(r"(\w+)", text):
   print('{}-{}: {}' .format(m.start(1), m.end(1), m.group(1)))

 0- 4: this
 5- 7: is
 8- 9: a
....
35-37: of
38-43: words
```

re flags

re.I	Performs case-insensitive matching.
re.M	Makes $ match the end of a line (not just the end of the string) and makes ^ match the start of any line (not just the start of the string).
re.S	Makes a period (dot) match any character, including a newline.
re.U	Interprets letters according to the Unicode character set. This flag affects the behavior of \w, \W, \b, \B.
re.X	Permits "cuter" regular expression syntax. It ignores whitespace (except inside a set [] or when escaped by a backslash) and treats un-escaped # as a comment marker.

Advanced Python Programming

Agenda



Lambda

- Python supports the creation of anonymous functions (i.e. functions that are not bound to a name) at runtime, using a construct called "lambda".
- This is a very powerful concept that well integrated into Python
- The most common use of lambda functions is as call-backs

Lambda definition

```
def f (x):
        return x**2

print( f(8))    # print 64
```

Function handler

Function parameters

Function body

```
g = lambda x: x**2

print (g(8))  # print 64
```

filter function

- *resultIter = filter(function, iterable)*
- filter function returns an iterator of elements of *iterable* for which *function* returns true.
- *iterable* may be either a sequence or an iterator
- The filter iterates through all elements of *iterable,* sends them(one by one) to *function* and includes in *resultIter* only elements for which the *function* returns True
  - If the *function* is not None, elements from input *iterable* will be include in a result only if them evaluates as True

filter function – example

For example:

```
list= [2, 18, 9, 22, 17, 24, 8, 12, 27]
seq = filter(lambda x: x % 3 == 0, list)
for val in seq:
        print(val)              # 18, 9, 24, 12, 27

seq = filter(lambda x: x < 0, range(-5,5))
for val in seq:
        print(val)              # -5, -4, -3, -2, -1
```

map function

- *resultIter = map(function, iterable, [iterables])*
- Return an iterator that applies *function* to every item of *iterable*.
- If additional *iterables* arguments are passed, *function* must take that many arguments and is applied to the items from all *iterables* corresponding
  - If the number of items in all *iterables* is not even, the map function will only iterate through the common items, in python 3 (the number of iterations is defined by minimal sequence length)
    - In python 2, the map function iterates through all items and sends None for absent correspondings (the number of iterations is defined by maximal sequence length)

map function – cont'd

For Example:

```
list = [2, 18, 9, 22, 17, 24, 8, 12, 27]
map(lambda x: x * 2 + 10, list)   # 14, 46, 28, 54, 44, 58, 26, 34, 64

map(lambda w: len(w), 'A lot of cats and dogs here'.split())
                                          #1, 3, 2, 4, 3, 4, 4

map(pow, [2, 3, 4], [10, 5, 1, 3])     # 1024, 243, 4
```

Equality (==) and is

- *is* operator checks whether 2 arguments refer to the same object
- == operator checks whether 2 arguments have the same value

Equality (==) and is - cont'd

- Numbers:

```
v1 = 10; v2 = 10
v1 == v2          # compare values, return True
v1 is v2          # probably reference to the same object, so True
print(id(v1), id(v2))     # 30779644, 30779644

v3 = 10
v4 = 11
v3 += 1
v1 == v2          # compare values, return True
v1 is v2          # probably reference to the same object, so True
print(id(v3), id(v4))     # 30779632, 30779632
```

Equality (==) and is - cont'd

- This kind of memory sharing we can in the following types as well:
  - string
  - bool
  - None

```
v1 = 'aaa'; v2 = 'aaa'
v1 == v2          # compare values, return True
v1 is v2          # probably reference to the same object, so True

x = True
y = True
x is y            # probably reference to the same object, so True
```

is and id keywords – cont'd

- unlike the immutable, python will never referee two different mutable objects to the same memory

For example:

```
v1 = [1,2,3]; v2 = [1,2,3]
v1 == v2      # compare values, return True
v1 is v2      # reference to different object, so False
v3 = v1
v1 == v3      # compare values, return True
v1 is v3      # reference to the same object, so True
```

Decorators

- Decorator is a feature that extends the functionality of functions without modify it
- Built-in python decorators are @staticmethod and @classmethod
- @staticmethod
  - The *staticmethod* decorator modifies a method so that it does not use the self variable. Static method do not have access to any attribute of a specific instance of the class.
- @classmethod
  - The *classmethod* decorator decorates methods as static methods, like *staticmethod*. Not like staticmethod, *the classmethod* receives the class object as the first parameter.

Built in decorators - Example

```
class Date(object):
    def __init__(self, day=1, month=1, year=1970):
        self.day = day
        self.month = month
        self.year = year

    @classmethod
    def from_string(cls, date_as_string):
        day, month, year = map(lambda val: int(val), date_as_string.split('-'))
        date1 = cls(day, month, year)
        return date1

    @staticmethod
    def is_date_valid(date_as_string):
        day, month, year = map(lambda val: int(val), date_as_string.split('-'))
        return day <= 31 and month <= 12 and year <= 3999
```

Composition of Decorators

- Define functions inside other functions

```
def printHello(name):
    def message():
        return "Hello "

    result = message() + name
    return result

print(printHello ("David"))

# Outputs: Hello David
```

Composition of Decorators – cont'd

- Functions can be passed as parameters to other functions
- Functions can return other functions

```
def message(name):
  return "Hello " + name

def printMessage(func):
     print(func("David"))

def byeMessage():
     def goodbyeMessage(name):
          return "Goodbye" + name
    return goodbyeMessage

print(printMessage (message)) # Outputs: Hello David
byeFunc = byeMessage()
byeFunc("David")              # Outputs: Goodbye David
```

Composition of Decorators – cont'd

- Function decorators are simply wrappers to existing functions.
- In this example let's consider a function that substitute space sequences by single space in output string of another function.

```
def get_text(name):
  return "long    sentence     for  {0}   ".format(name)

def squeezeSpace  (func):
  def  squeezer(name):
     str=func(name)
     return re.sub(r' +', ' ', str)
  return squeezer

getText = squeezeSpace(get_text)
print(getText ("John"))   # prints "long sentence for John"
```

Composition of Decorators – cont'd

- Python's Decorator Syntax – the final example

```
import re
def squeezeSpace (func):
  def squeezer(name):
    str=func(name)
    return re.sub(r' +', ' ', str)
  return squeezer

@ squeezeSpace
def get_text(name):
  return "long    sentence    for {0}   ".format(name)

print(get_text("John"))  # prints "long sentence for John"
```

Passing arguments to decorators

- Suppose we want different functions to squeeze custom character

```
def squeezeChar(ch):
  def squeezeRepeat(func):
    def squeezer(name):
      str=func(name)
      return re.sub("{}+".format(ch), ch, str)
    return squeezer
  return squeezeRepeat

@squeezeChar (" ")
def get_text(name):
  return "long    sentence    for {0}   ".format(name)

@squeezeChar ("\n")
def parse_text(text):
      .....

print(get_text("John"))          # prints "long sentence for John"
```

Special method names

- A class can implement certain operations that are invoked by special syntax (such as arithmetic operations or iterating) by defining methods with special names.
- Using special methods, your classes can act like sets, like dictionaries, like functions, like iterators, or even like numbers

Overload operators - __str__

- *__str__* returns the string representation of an object
- __str__ function can be placed in classes and it is used implicitly in string context like str() or print

```
class Fraction:
    def __init__(self, numerator, denominator):
        self.numerator = numerator
        self.denominator = denominator

    def __str__(self):
        return "{0}/{}".format(self.numerator, self.denominator)

f = Fraction(1,2)
print (f)              # print 1/2
```

Overload operators - Comparing objects

- object.__lt__(self, other) called for x<y
- object.__le__(self, other) called for x<=y
- object.__eq__(self, other) called for x==y
- object.__ne__(self, other) called for x!=y
- object.__gt__(self, other) called for x>y
- object.__ge__(self, other) called for x>=y

```
class Fraction:
        # ....
        def __lt__(self, other):
                return (self.numerator * other.denominator) < \
                 (other.numerator* self.denominator)
        def __ge__(self, other):
                return not (self < other)
```

Overload operators - Comparing objects – cont'd

```
f1= Fraction(1,2)
f2 = Fraction(1,3)

print(f2 < f1)          # print True
print(f1 >= f2)         # print True
```

Overload operators – more examples

- There are much more *operator overloading* functions like:
  - __iter__ for iterating through – will be discussed later
  - __hash__ called on items insertion to dictionary
  - __len__ - to support the *len* built-in function
  - __contains__ - to support *in* operator
  - __nonzero__ called to implement truth value while testing the built-in operation bool() or boolean context
  - __add__, __sub__, __mul__, __floordiv__, __mod__ for arithmetic operations
  - __enter__ and __exit__ - to implement context manager
  - And much more…

Iterators

- *Iterator* is an objects that can be used to go through all its items (with a *for* loop for example)
- For example:

```
for i in [1, 2, 3, 4]:    or    for i in "python":    or    for i in open("file.txt"):
                                    print( i)
```

- There are many functions which consume these *iterables:*
  - for loop
  - sum, min, max built-in functions
  - filter and map built-in functions
  - Comprehension
  - Type conversions to python sequence
  - etc

The Iterator Protocol

- In order to become an *iterable object* our class should implement the following protocol:
  - It should implement *__iter__* function that returns an iterator
  - The iterator should implement *__next__* function that gives us the next element each time we call it.
    - *__next__* method of python 3, implemented as *next* method in python 2
- The built-in function *iter* takes an iterable object and returns an iterator. Most *iter* function invocations are implicit, by *iterator consumers*

The Iterator Protocol – cont'd

```
it = iter([1, 2, 3])
print(it)                    # prints  <listiterator object at 0x1004ca850>
print( it.__next__())        # prints  1
print( it.__next__())        # prints  2
print( it.__next__())        # prints  3
print( it.__next__())
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
StopIteration
```

Implement Iterator – Version 1

```
class Yrange:
   def __init__(self, n):
      self.i = 0
      self.n = n

   def __iter__(self):
      return self

   def __next__(self):
      if self.i < self.n:
         self.i += 1
         return self.i-1
      else:
         raise StopIteration()
```

Implement Iterator – Version 2

```
class Zrange:
   def __init__(self, n):
      self.n = n

   def __iter__(self):
      return Zrange_iter(self.n)
```

```
class Zrange_iter:
   def __init__(self, n):
      self.i = 0
      self.n = n

   def __next__(self):
      if self.i < self.n:
         self.i += 1
         return self.i-1
      else:
         raise StopIteration()
```

Iterators consumers

```
y = Yrange(5)
print(list(y))                          #prints [0, 1, 2, 3, 4]
print(list(y))                                  #prints []

z = Zrange(5)
print(list(z))                                  # prints [0, 1, 2, 3, 4]
print(list(z))                                  # prints [0, 1, 2, 3, 4]
print(sum(z))                                   # prints 10
l1 = [x*10 for x in z]                          # l1 = [0, 10, 20, 30, 40]
l2 = filter(lambda n : n%2, z)                  # l2 = [1, 3]
```

Generators

- Generators are functions allow as to declare a function that behaves like an iterator, i.e. it can be used in a for loop
- Generators simplifies creation of iterators. A generator is a function that produces a sequence of results instead of a single value.
- Generators are functions that have a ***yield*** statement

Generators – example

For example:

```
def gen_range(n):
    i = 0
    while i < n:
        yield i
        i += 1

resIter = gen_range(5)
for n in resItem:
        print(n)
```

Generators consumers

```
def gen(n):
    for i in range(n+1):
        yield i

print(sum(gen(10)))                    # 55

for el in gen(4):                      # 0 1 2 3 4
    print (el)

first7Values = gen(7)
lst = [x*x for x in first7Values]
print (lst)                            # [0 1 4 9 16 25 36 49]
```

Python Garbage Collection

- Python's memory allocation and deallocation method is automatic.
- Python uses two strategies for memory allocation - **reference counting** and **garbage collection**.
- Prior to Python version 2.0, the Python interpreter only used reference counting for memory management.

Reference counting

- Reference counting works by counting the number of times an object is referenced by other objects in the system.
- When references to an object are removed, the reference count for an object is decremented.
- When the reference count becomes zero the object is deallocated, including invoking __del__ method on objects
- Reference counting is extremely efficient but it does not always works
  - One of the biggest problems in reference counting is reference cycles.

Reference counting – reference cycles

- A reference cycles prevents from objects' reference count to drops down to zero, yet there is no way to reach the objects
- For example:

```
def make_cycle():
    l = [ ]
    l.append(l)
make_cycle()
```

AGC - Automatic Garbage Collection

- The main role of AGC in python is to detect reference cycles for python sequences and user objects and to deallocate them
- The automatic garbage collection has been included in Python since version 2.0.
- A reference-counting scheme collects objects as soon as they become unreachable
- The time of collection of objects with circular references is unknown.
- Do not depend on immediate finalization of objects when they become unreachable

Python gc - Manual Garbage Collection

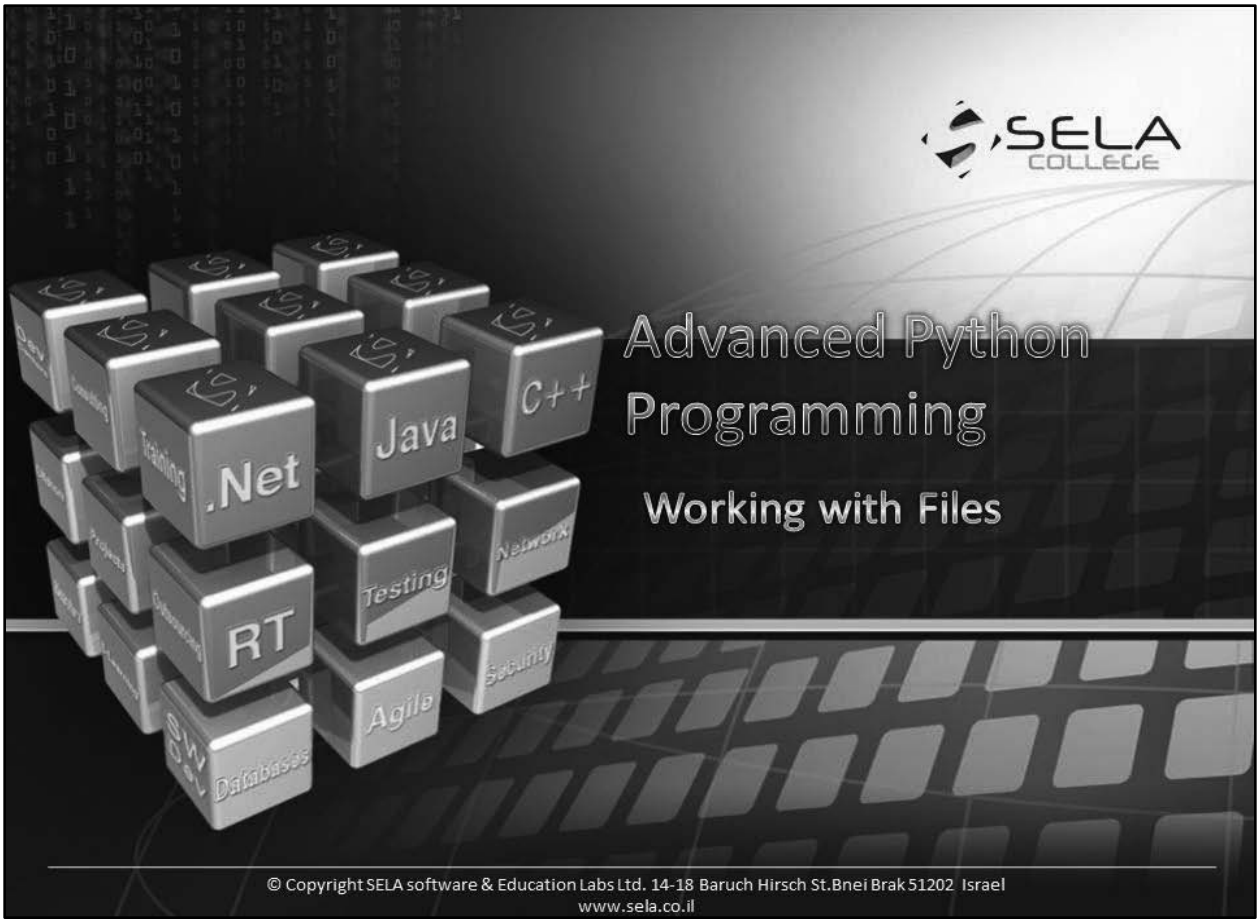
- For some programs, especially long running server applications or embedded applications, automatic garbage collection may not be sufficient
- The garbage collection can be invoked manually with gc.collect() function, which returns the number of objects it has collected and deallocated

gc example

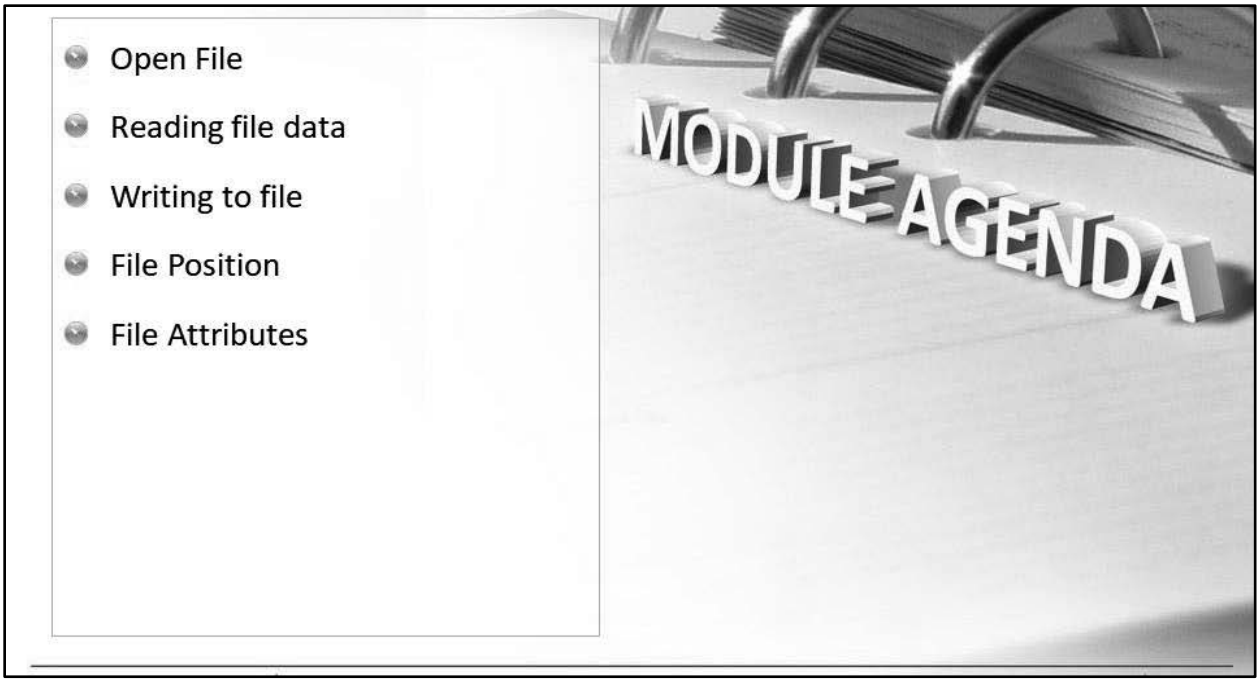
```
import gc

def make_cycle():
    l = [ ]
    l.append(l )

def main():
    collected = gc.collect()
    print("Garbage collector: collected {} objects.".format (collected)
    print ("Creating cycles...")
    for i in range(10):
        make_cycle()
    collected = gc.collect()
    print ("Garbage collector: collected {} objects.".format (collected))

main()
```

```
Garbage collector: collected 0 objects.
Creating cycles...
Garbage collector: collected 10 objects.
```

Advanced Python Programming



Agenda



Open Files

- fileObject = open(filename, mode = 'r')
- *modes*:
  - 'r' - Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
  - 'w' - Opens a file for writing. If file already exists it will be truncated, if not it will be created.
  - 'a' - Opens a file for appending. The file pointer is at the end of the file if the file exists. If the file does not exist, it creates a new file for writing.
  - 'r+' - Opens a file for both reading and writing. The file pointer will be at the beginning of the file
  - 'w+' - Opens a file for both writing and reading. If file already exists it will be truncated, if not it will be created.

Close Files

- *fileObject*.close() - to close it and free up any system resources taken up by the file
- *fileObject*.closed – returns whether the file is already closed (True/False)

- It is good practice to use the **with** keyword when dealing with file objects (context manger)
- This has the advantage that the file is properly closed after its suite finishes, even if an exception is raised on the way.
- It is also much shorter than writing equivalent try-finally blocks

```
with open('output.txt', 'w') as f:
        f.write('Hi there!')
```

Reading data

- *fileObject*.**readline()**
  - reads a single line from the file a newline character (\n) is left at the end of the string
  - if f.readline() returns an empty string, the end of the file has been reached

- *fileObject*.**readlines( )** – return a list containing the file content lines

```
file = open('somefile.txt',  'r')
print (file.readlines())          #['first line\n', 'second line\n']
```

Reading data – cont'd

- It is possible to read file data with **read** function
- *fileObject*.read(size) - reads some quantity of data and returns it as a string:
  - s = f.read(size)
- *size* is an optional numeric argument. When *size* is omitted or negative, the entire contents of the file will be read and returned:
  - wholeData = *fileObject*.read() - reads the entire contents of the file

File iterator

- For reading lines from a file, you can loop over the file object. This is memory efficient, fast, and leads to simple code:

```
for line in fileObject:
    print (line)
```

Writing to file

- *fileObject*.write(string) - writes the contents of *string* to the file

```
with open('out', 'w') as f:
        f.write('This is a test\n')
```

File Position

- tell() - returns current position in the file, in bytes from the beginning of the file.
- The *seek(offset, from=0)* method changes the current file position.
  - The *offset* argument indicates the number of bytes to be moved.
  - The *from* argument specifies the reference position from where the bytes are to be moved:
    - io.SEEK_SET (0) – from the beginning of the file
    - io.SEEK_CUR (1) – from current file position
    - io.SEEK_END (2) – from the end of the file
  - offset must be zero for SEEK_CUR and SEEK_END in python 3.x

File object attributes

- file.closed - Returns true if file is closed, false otherwise.
- file.mode - Returns access mode with which file was opened.
- file.name - Returns name of the file.

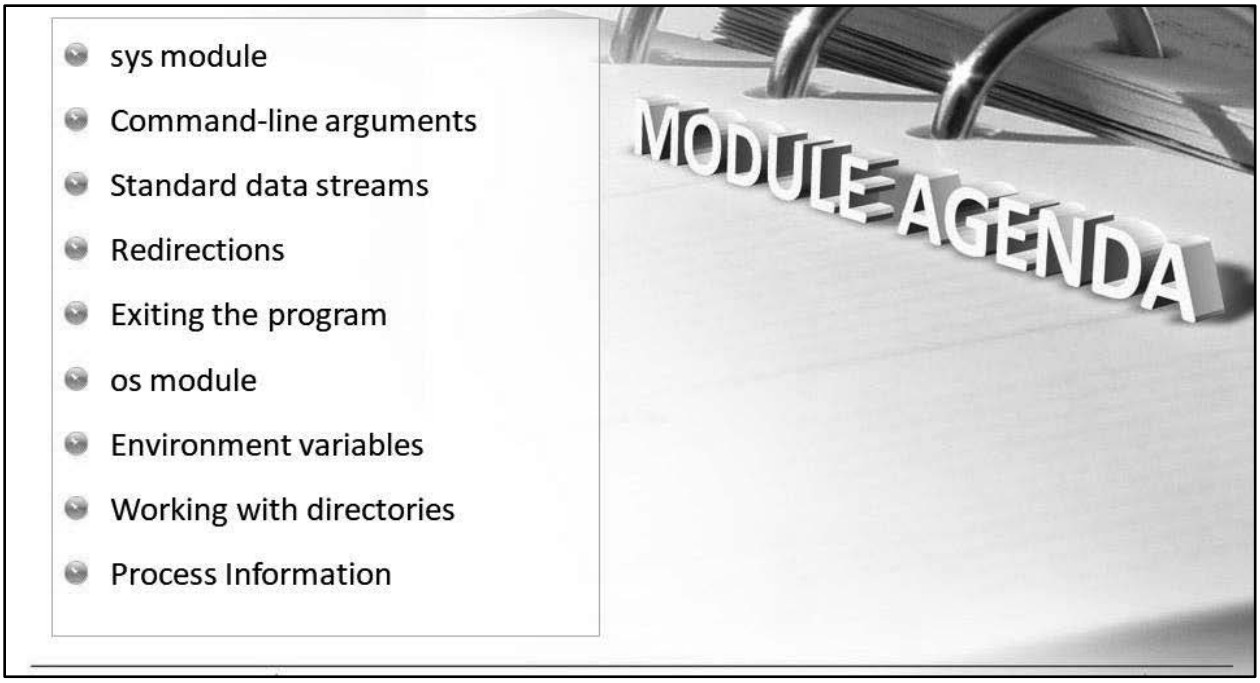
```
fo = open("foo.txt", "w")
print ("Name of the file: ", fo.name )
print ("Closed or not : ", fo.closed )
print ("Opening mode : ", fo.mode)
```

The output:

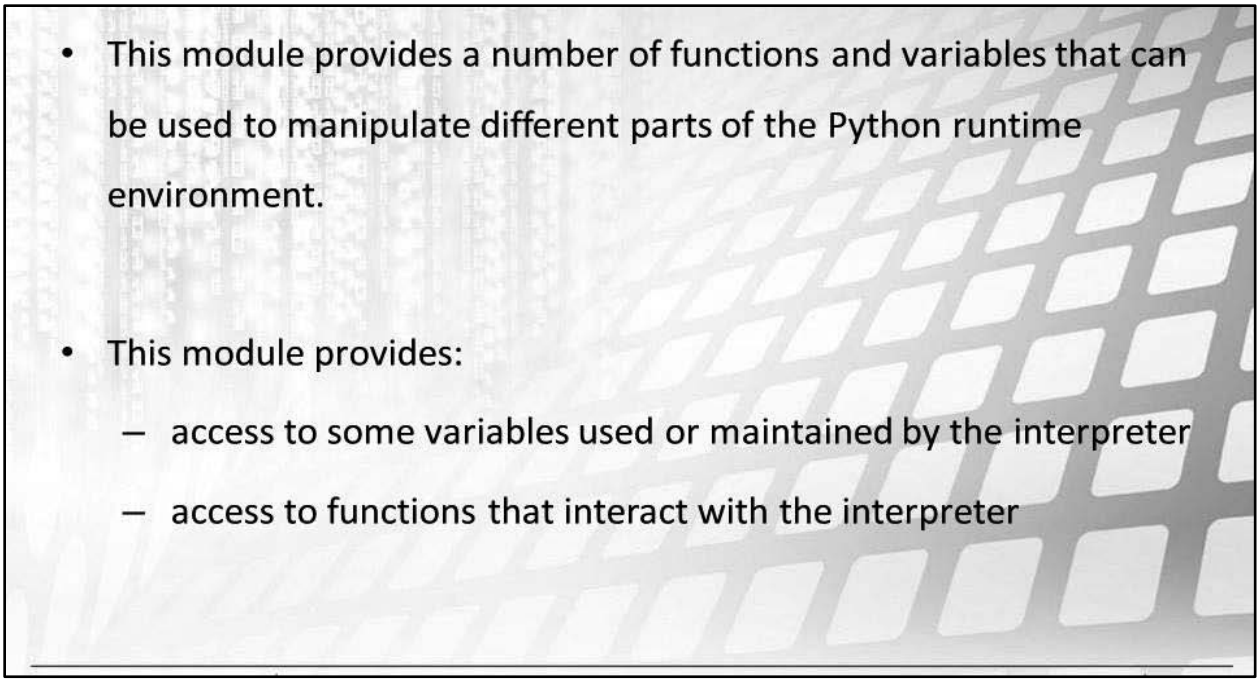
```
"foo.txt“
False
w
```

Advanced Python Programming

Agenda



sys module



sys module – version and platform

- sys.version
  - The output: 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit (Intel)]
- sys.platform

System	platform value
Linux (2.x *and* 3.x)	linux2
Windows	win32
Windows/Cygwin	cygwin
etc	

Command-line arguments

- Arguments passed to a script called Command Line Arguments
- Python script can access those command line arguments through *sys.argv* list.
- The first item of this list is a path of the script itself

```
for i, arg in enumerate(sys.argv):
    if i == 0:
        print("Script name: {}".format(sys.argv[0]))
    else:
        print("{}. argument is: {}".format(i,sys.argv[i]))
```

Standard data streams

- Almost every programmer familiar with standard streams:
  - standard input - as default, connected to the keyboard
  - standard output - as default, connected to the terminal (or working window)
  - standard error - as default, connected to the terminal (or working window)
- These data streams can be accessed from Python via the objects of the sys module: sys.stdin, sys.stdout and sys.stderr.

Standard data streams – cont'd

```
sys.stdout.write("some string")
s="some string"
sys.stdout.write(s)

line = sys.stdin.readline()[:-1] #removes the \n
            from the end of the line
sys.stdout.write(line)
```

Redirections

- The standard output, error and input can be redirected e.g. into a file, so that we can process this file later with another program.
- We can redirect both stderr and stdout into the same file or into separate files

```
import sys
save_stdout = sys.stdout
fd = open("test.txt","w")
sys.stdout = fd
print("This line goes to test.txt")
#... Do things...
sys.stdout = save_stdout
fd.close()              # return to default:
```

Redirections – cont'd

```
import sys

fd = open("in", "r")
sys.stdin = fd

x = input()       #reads from the file
print(x)
```

os module

- The OS module provides a portable way of using operating system dependent functionality.
- The functions that the OS module provides allows you to interface with the underlying operating system that Python is running on – be that Windows, Mac or Linux.

Environment variables

- os.environ - A mapping object representing the string environment.
- os.getenv(varname, value=None) - Return the value of the environment variable varname if it exists, or value otherwise.

  For example:

  ```
  os.environ['HOME']
  os. getenv("HOME")
  ```

- Both returns a PATH environment variable value.
- PATH value specifies the directories in which executable programs are located on the machine that can be started without knowing and typing the whole path to the file on the command line

Environment variables – cont'd

- os.putenv(*varname*, *value*) - Set the environment variable named *varname* with a *value*.
  - Such a changes to the environment affects the sub-processes, created after the change

os.path — Common pathname manipulations

- os.path.dirname(*path*) - return the directory name of pathname *path*
- os.path.exists(*path*) - return True if *path* refers to an existing path
- os.path.isfile(*path* - return True if *path* is an existing regular file.
- os.path.isdir(*path*) - return True if *path* is an existing directory.
- os.path.islink(*path*) - return True if *path* refers to a directory entry that is a symbolic link
- os.path.join(*path*, **paths*) - join one or more path components intelligently
- etc

Working with directories

- **os.getcwd()** – returns current working directory
- **os.chdir(path)** – change current directory

- Create a directory named path **os.mkdir(path)**
- Recursive directory creation function. **os.makedirs(path)**

- Return a list of the entries in the directory given by path. **os.listdir(path)**

- Remove (delete) the file path. **os.remove(path)**
- Remove (delete) the directory path. **os.rmdir(path)**

Advanced Python Programming

Exceptions Intro

- An exception is an error that happens during the execution of a program. When that error occurs, Python generates an exception that can be handled
- Unhandled exceptions cause your program to crash.
- Exceptions come in different types. Python generates an exception with type, suitable to an error
- The words "try", "except" and "finally" are Python keywords that used to catch exceptions.
- Exceptions can be raised (thrown) using "raise" statement

Exceptions Intro – cont'd

For Example:

```
try:
        print(1/0)
except ZeroDivisionError:
        print ("You can't divide by zero")
```

Some Built-in Exception Errors

- Below is some common exceptions errors in Python:
  - **Exception** Base class for all exceptions
  - **IOError** If the file cannot be opened.
  - **ImportError** If python cannot find the module
  - **ValueError** Raised when built-in function receives an inappropriate value
  - **KeyError** Raised when the specified key is not found in the dictionary.
  - **NameError** Raised when an identifier is not found in the local or global namespace.
  - **EOFError** Raised when one of the built-in functions (input() or raw_input()) hits an end-of-file condition (EOF) without reading any data
  - **ArithmeticError** The base exceptions for various arithmetic errors: OverflowError, ZeroDivisionError, FloatingPointError
  - **SyntaxError** Raised when the syntax is incorrect

Exception handling example

Version 1:

```
s = input("Enter a number between 1 - 10")
number = int(s)
print ("your number is – {}".format(number))
```

- If the input is not numeric, "67a" for example, the code raises exception:

  ValueError: invalid literal for int() with base 10: '67a'

Exception handling example – cont'd

Version 2:

```
number = 5
try:
    s = raw_input("Enter a number between 1 – 10, 5 is default")
    number = int(s)
except ValueError:
    print ("your value is incorrect, default value is set“)
print ("your number is – {}".format(number))
```

- The program will continue running whether the input is correct or not.

Argument of an Exception

- An exception can have an argument, that holds an additional information about the problem and cause of the exception

```
try:
        print(val + 10)
except NameError as ne:
    print(ne)

File "<stdin>", line 1, in <module>
NameError: name 'val' is not defined
```

Multiple excepts

try statement supports multiple excepts

```
try:
    ....
except ValueError as ve:
    ...
except ArithmeticError as ae:
    ...
except Exception:
    ...
```

Try ... finally example

```
fo = open("some file","r")
try:
        # work with file
finally:
        fo.close()
```

Raising Exceptions

- The *raise* statement allows the programmer to force a specified exception to occur
- *raise* can come with a parameter (or several parameters). The parameter is a message or any additional information about the error
- *raise* usually placed in infrastructure methods, classes and modules.

```
def checkGradeValidation(gradeValue):
        if type(gradeValue)  != int:
                raise TypeError("grade value must be integer")
        if gradeValue < 0 or gradeValue>100:
                raise ValueError("grade value must be between 0 – 100)
        .....
```

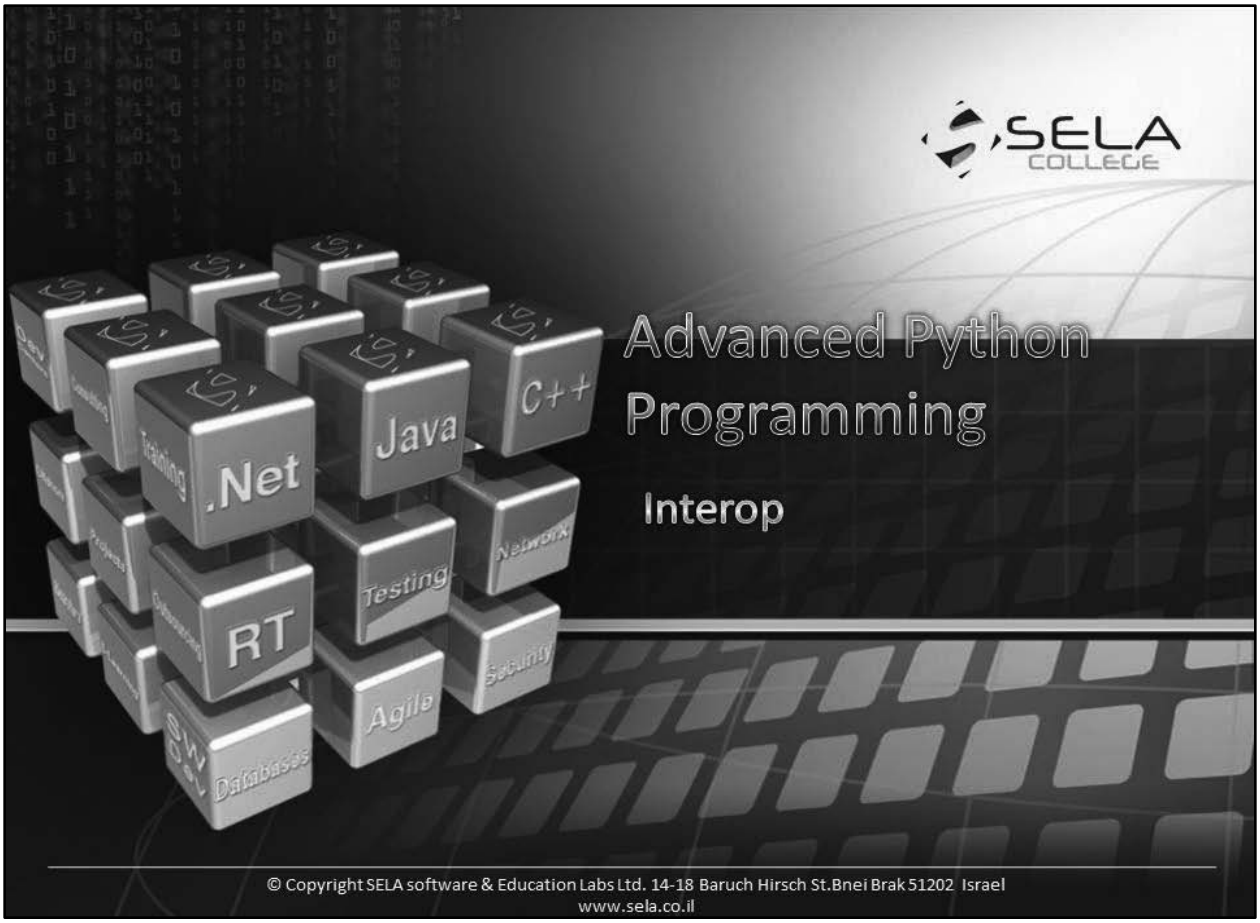
User-Defined Exceptions

- Python has many built-in exceptions
- Sometimes we may need to create a custom exceptions that serves our purpose.
- Python, supports creation of a custom exception by defining a new class that derives from *Exception* built-in class
- Usually custom exceptions add some additional information

User-Defined Exceptions

```
class ValidationError(Exception):
    def __init__(self, message, errors):
        Exception.__init__(self, message) # base class __init__
        self.errors = errors

    def __str__(self):
        return "{}, error is {}".format(Exception.__str__(self),
                                        self.error)
```

Advanced Python Programming



Python C interoperability

- There a various tools which make it easier to bridge the gap between Python and C/C++
- **Cython** - programming language, a superset of Python with a foreign function interface for invoking C/C++ routines. It is actually a Python and C source code translator that integrates on a low level.
- **ctypes** is a Python module allowing to create and manipulate C data types in Python. These can then be passed to C-functions loaded from dynamic link libraries.
- **elmer** - compile and run python code from C, as if it was written in C
- **weave** - include C code lines in Python program
- etc

ctypes introduction

- ctypes is a foreign function library for Python.
- It provides C compatible data types and allows calling functions in DLLs or shared libraries.
- ctypes exports the *cdll* class for loading dynamic link libraries.
- *cdll* loads libraries which export functions using the standard C decl calling convention
- The LoadLibrary() method used to load the dll
- Lets start with simple example

C Test.dll Example

- First, lets create C simple dll named Test
  - Win32->Win32 project
  - Pick dll type, choose Empty project
- Lets write some code

```
#include <stdio.h>

extern "C" {
    __declspec(dllexport) void Print() {
        printf("declspecl say hello"); }

    __declspec(dllexport) int Add(int a, int b) {
        return a+b; }
}
```

Python code Example

- First step for interoperability is to load dll library.
- Python loads C dlls with LoadLibrary function placed in cdll
- *LoadLibrary(dll path) -> loaded dll object*
  - dll path doen't have to have .dll exctension
- Loaded dll object now has access to dll's functions

```
import ctypes

loadedDll = ctypes.cdll.LoadLibrary(r"C:\...\TestDll")
res = loadedDll.Add(2,3)
print(res)
```

ctype types

- int is the default parameter type or return value type and the only type python can work with without casting

- Lets see C dll function simple example:

```
__declspec(dllexport) double doubleFunc(double d)
{
        return ++d;
}

import ctypes
d = ctypes.cdll.LoadLibrary(r"C:\...\TestDll")
res = d.doubleFunc(1.23)
print (res)
```

ctype types – cont'd

- We get the fallowing result when trying to run the python program:

```
Traceback (most recent call last):
  File "C:\Python27\ctypes.py", line 7, in <module>
    res = d.doubleFunc(1.23)
ctypes.ArgumentError: argument 1: <type 'exceptions.TypeError'>:
Don't know how
to convert parameter 1
```

- We need a types conversion table

Types conversion table

ctypes type	C type	Python type
c_char	char	1-character string
c_wchar	wchar_t	1-character unicode string
c_byte	char	int/long
c_ubyte	unsigned char	int/long
c_short	short	int/long
c_ushort	unsigned short	int/long
c_int	int	int/long
c_uint	unsigned int	int/long
c_long	long	int/long
c_ulong	unsigned long	int/long
c_longlong	__int64 or long long	int/long
c_ulonglong	unsigned __int64 or unsigned long long	int/long

Types conversion table – cont'd

ctypes type	C type	Python type
c_float	float	float
c_double	double	float
c_longdouble	long double	float
c_char_p	char * (NUL terminated)	string or None
c_void_p	void *	int/long or None

- An example of python code:

```
import ctypes
val = ctypes.c_double(11.22)
print (val)              # c_double(11.22)
print (val.value)        # 11.22
```

TestDll.dll Example

```
#include <stdio.h>
#include <stdlib.h>

extern "C " {

__declspec(dllexport)  double doubleFunc(double  d) {
        return ++d;
}

__declspec(dllexport)  char* strFunc(char* str) {
        return str;
}
```

TestDll.dll Example cont'd

```
__declspec(dllexport)  void swap(int* p1, int* p2) {
    int temp=*p1;
    *p1=*p2;
    *p2=temp;
}
__declspec(dllexport) float* pointerFunc(float val)  {
    float * ptr = (float*)malloc(sizeof(float));
    *ptr = -val;
    return ptr;
}
__declspec(dllexport) Point* structFunc(Point p) {
    Point* cp = (Point*)malloc(sizeof(Point));
    cp->x=p.x;
    cp->y=p.y;
    return cp;
}
```

```
typedef struct
{
        int x;
        int y;
} Point;
```

Python Code

- When dll is loaded python has access to its functions
- functions within the dll has *restype* property that helps define return value type

```
import ctypes

d = ctypes.cdll.LoadLibrary(r"C:\...\TestDll")

par = ctypes.c_double(12.3)
d.doubleFunc.restype = ctypes.c_double
res = d. doubleFunc(par)
print (res)       #13.3
```

Python Code – cont'd

```
par = ctypes.c_char_p('hi all')
d.strFunc.restype = ctypes.c_char_p
print (d.strFunc(par))  # hi all
```

- ctypes has *byref* function for integration with C functions that receives arguments by reference
- *byref* works with ctypes types only .

```
a = ctypes.c_int(2); b = ctypes.c_int(5)
print ("before swap a = {}, b = {}".format( (a.value, b.value)) # 2, 5
res = d.swap(ctypes.byref(a), ctypes.byref(b))
print ("after swap a = {}, b = {}".format( (a.value, b.value))    # 5,2
```

Python Code – cont'd

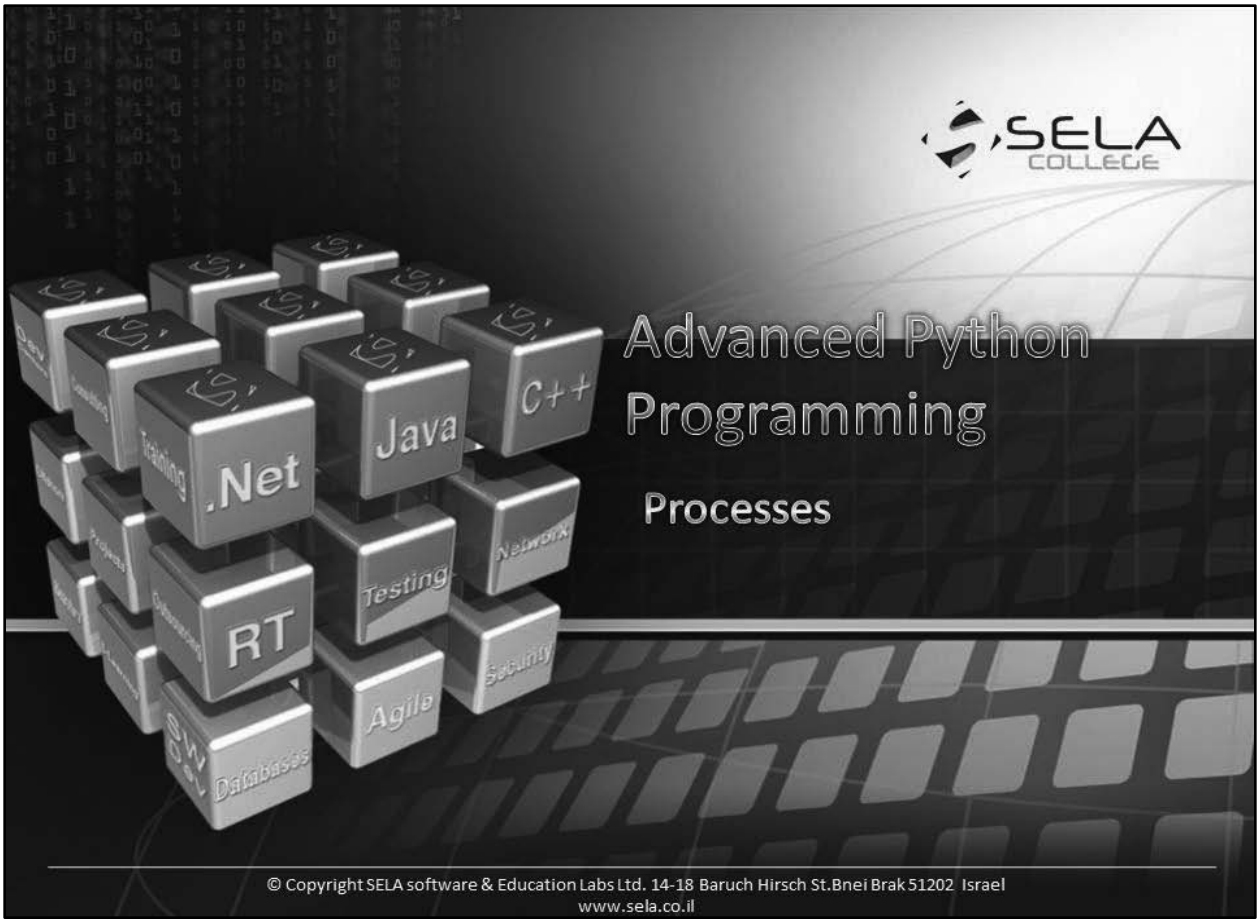
- ctypes module also has POINTER type for integrating C pointers

```
d. pointerFunc.restype = ctypes.POINTER(ctypes.c_float)
res=d. pointerFunc(ctypes.c_float(15))

print (res)                    #<__main__.LP_c_float object at
                                                    0x01E98260>
print (res.contents)           #c_float(-15.0)
print (res.contents.value)     # 15.0
```

Python Code – cont'd

```
class PointClass(ctypes.Structure):
 _fields_ = [
          ("x", ctypes.c_int),
          ("y", ctypes.c_int)
        ]
d. structFunc.restype  = ctypes.POINTER(PointClass)
c = PointClass (100,200)
res = d. structFunc(c)

print (res)                    #<__main__.LP_PointClass object at 0x01E982B0>
print (res.contents)           #<__main__. PointClass object at 0x01E98210>
print (res.contents.x, res.contents.y)          # 100 200
```

Advanced Python Programming



multiprocessing

- *multiprocessing* module is responsible for processes creation and management
- *multiprocessing* API is similar to *threading* API and it extends the *threading* module functionality
- Python processes avoid the Global Interpreter Lock (GIL) and take full advantages of multiple processors on a machine
- *multiprocessing* module includes a lot of useful classes for processes creation, synchronization and IPC

The multiprocessing.Process class

- *We can use* the **Process** class to create a process object

Process(group=None, target=None, name=None, args=())

  - *target* is the callable object to be invoked by the Process
  - *name* is the process name
  - *args* is the argument tuple for the target invocation.
  - *group* – should be always be None

Process start and join functions

- *start*() - Starts the process's activity.
- *join*([*timeout*])
  - Block the calling method until the process whose join() method is called terminates or until the optional timeout occurs.
  - If timeout parameter specified the calling method will be blocked up to *timeout* seconds. The *exitcode* will stays None until the process is actually finishes (Will be discussed later)

The Process example

```
import multiprocessing  as MP
def f(val):
    cp = current_process()
    print("in child process name={}, pid={}, val = {}".format(cp.name,
    cp.pid, val)

if __name__ == '__main__':
    p = Process(target=f, args=('paramVal',))
    p.start()
```

The Process example – cont'd

- In some platform, like windows, the child process goes through the main space before executing the target function. This is why the creation of child process must be under the: *if __name__ == '__main__'*
- Single-item tuples require a trailing comma:

  tpl = (1,)

join with timeout example

```
import multiprocessing as MP
import time

def func():
    for n in range(100000000):
        pass

if __name__ == "__main__":
    p = MP.Process(target=func)
    p.start()

    p.join(1)
    while p.exitcode is None:
        print("in loop")
        p.join(1)
        time.sleep(1)
```

The output:

```
In loop
In loop
In loop
In loop
```

Exchanging data between processes

- When it comes to communicating between processes, the multiprocessing modules has two primary methods:
  - Queues
  - Pipes

Exchanging data between processes – Queue

- Multiprocessing Queue main functions:
  - put
  - put_nowait
  - get
  - get_nowait
  - empty / full

Exchanging data between processes – Queue

```
from multiprocessing import Process, Queue
def f(q):
    q.put(["one", "two",3])

if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print(q.get())    # prints (["one", "two",3]
    p.join()
    q.close()
```

Exchanging data between processes – Pipe

- Pipe() returns a pair of connection objects connected by a pipe which by default is duplex (two-way)
  - Each connection object has send() and recv() methods
- A *Pipe* can only have two endpoints
- A *Pipe* is much faster

Exchanging data between processes – Pipe

```
from multiprocessing import Process, Pipe

def f(conn):
    conn.send(['hello', 'world'])
    conn.close()

if __name__ == '__main__':
    parent_conn, child_conn = Pipe()
    p = Process(target=f, args=(child_conn,))
    p.start()
    print(parent_conn.recv())   # ['hello', 'world']
    p.join()
```

Synchronization between processes

- Take a look at the following program:

```
import multiprocessing as MP
import sys

def loop():
    for i in range(400):
        sys.stdout.write(str(i)+" ")
        sys.stdout.flush()
        sys.stdout.write("in process ")
        sys.stdout.flush()
        sys.stdout.write(MP.current_process().name + "\n")
        sys.stdout.flush()
```

Synchronization between processes – cont'd

```
if __name__ == "__main__":
    MP.Process(target=loop, name="child1").start()
    MP.Process(target=loop, name="child2").start()

    for i in range(400):
        sys.stdout.write(str(i)+" ")
        sys.stdout.flush()
        sys.stdout.write("in process ")
        sys.stdout.flush()
        sys.stdout.write(MP.current_process().name + "\n")
        sys.stdout.flush()
```

Synchronization between processes-cont'd

- multiprocessing module has 3 classes for process synchronization:
  - *Lock* - non-recursive lock object
  - *Rlock* - recursive lock object
  - *Semaphore* – created with internal *counter* and can be acquired *counter* times before released
- They all support context manages and can be used with *with* statement
- They all have the following function:
  - acquire(blocking=True, timeout=-1)
    - acquire returns True if the locking were successful and False, otherwise
  - release()

Multiprocessing Lock example

```
import multiprocessing as MP
import sys

def loop(lock):
    for i in range(400):
        with lock:
            sys.stdout.write(str(i)+" ")
            sys.stdout.flush()
            sys.stdout.write("in process ")
            sys.stdout.flush()
            sys.stdout.write(MP.current_process().name + "\n")
            sys.stdout.flush()
```

Multiprocessing Lock example – cont'd

```
if __name__ == "__main__":
    lock = MP.Lock()
    MP.Process(target=loop, name="child1", args=(lock,)).start()
    MP.Process(target=loop, name="child2", args=(lock,)).start()

    for i in range(400):
        sys.stdout.write(str(i)+" ")
        sys.stdout.flush()
        sys.stdout.write("in process ")
        sys.stdout.flush()
        sys.stdout.write(MP.current_process().name + "\n")
        sys.stdout.flush()
```

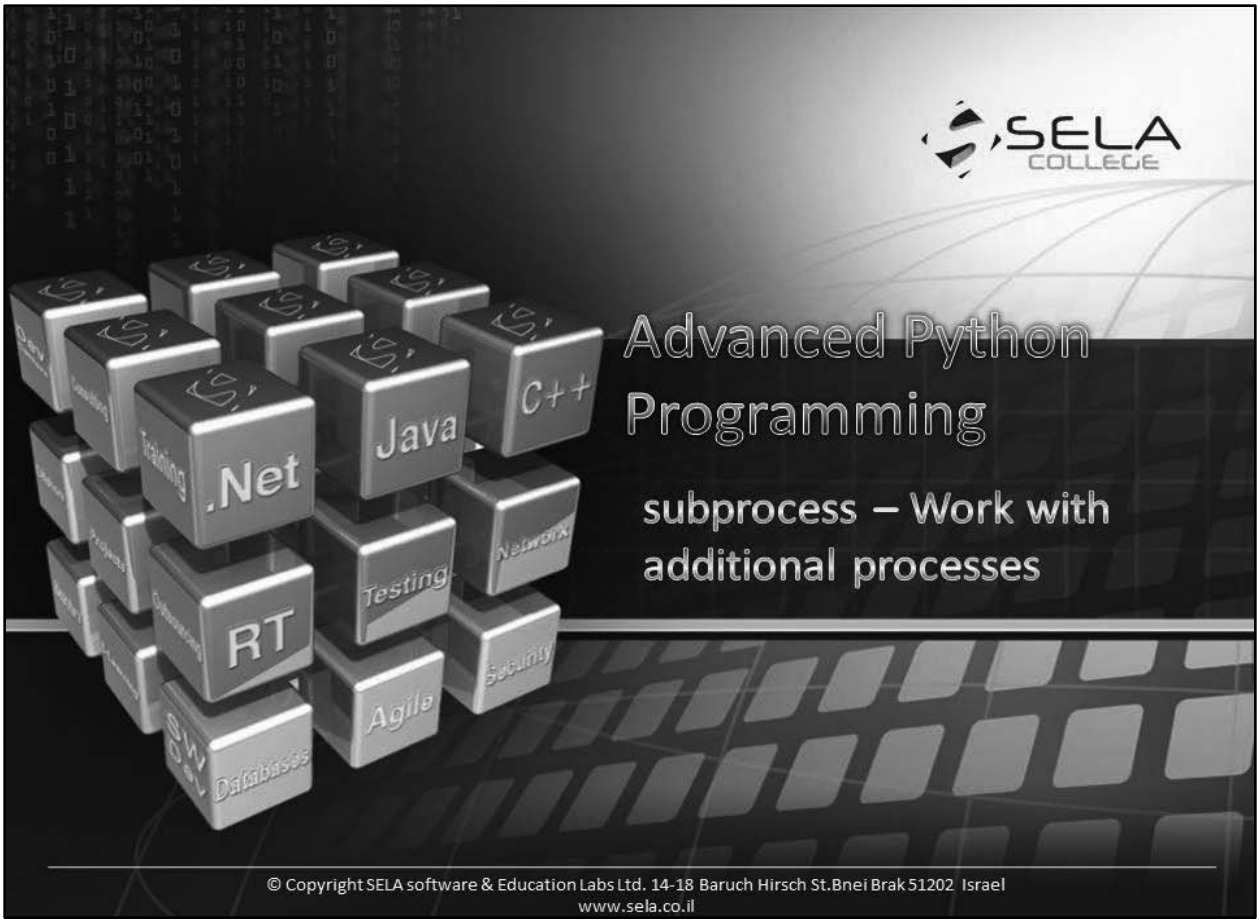
Multiprocessing Pool

- Multiprocessing module contains (among others) a Pool class that can be used for parallelize executing a function across multiple inputs.
- Using a *Pool* can be a convenient approach for simple parallel processing tasks. Some of *Pool* tasks are:
  - pool.map
  - pool.map_unordered
  - pool.imap
  - pool.map_async
  - pool.apply
  - etc

Multiprocessing Pool – example

```
from multiprocessing import Pool

def increment(number):
   return number + 1

if __name__ == '__main__':
   numbers = [1,2,3,4,5,6,7,8,9,10]
   pool = Pool(processes=3)
   incrementedList = pool.map(increment, numbers)
   print(incrementedList)        # [2,3,4,5,6,7,8,9,10,11]
```

Advanced Python Programming



The subprocess module

- The *subprocess* module provides a consistent interface to creating and working with additional processes.
- The *subprocess* module has a wide functionality for running additional processes, controlling their IO, including shell feathers like wildcards, pipes, redirections
- The *subprocess* module is an updated module for old and deprecated functionality with the same purpose like: os.system(), os.spawn*(), os.popen*() …

The call function

- subprocess.call(*args, stdin=None, stdout=None, stderr=None, shell=False, cwd=None, timeout=None*)
- Run the command described by *args*. Wait for command to complete, then return the returncode

The call function example

```
import subprocess

subprocess.call(["dir", "*.py"], shell=True)
subprocess.call([r"C:\Python27\test.py"], shell=True)
subprocess.call(["calc.exe"])

f = open(r"C:/Test/out.txt", "w")
subprocess.call(["dir", "/p"], shell=True, stdout=f)

print( "end")
```

The check_output function

- subprocess.check_output(*args*, *stdin=None*, *stderr=None*, *shell =False*, *cwd=None*, *timeout=None*)
- Run command with arguments and return its output.
- If the return code was non-zero it raises a CalledProcessError.

The check_output function example

```
def getCommandOutput(cmd):
    try:
        output = subprocess.check_output(cmd,
cmd="C:\someDir")
        return True, output
    except subprocess.CalledProcessError:
        return False, None

    isOk, output = getCommandOutput("dir")
```

Popen class

- The Popen class offers a lot of flexibility to handle the common and the less common cases not covered by the convenience functions.
- To support a wide variety of use cases, the Popen accept a large number of optional arguments (in most cases, most of the arguments can stay with their default value)
- Popen doen't block the calling function

Popen class – cont'd

- The most common Popen constructor arguments (like in *call* and *check_output* functions) are:
  - args
  - shell
  - stdin, stdout, stderr:
    - Existing file descriptor
    - PIPE - indicates that a new pipe to the child will be created

Popen example

```
import subprocess as SP
r = SP.Popen(["dir"], shell=True)
print(r.returncode)          # probably None


import subprocess as SP
r = SP.Popen(["dir"], shell=True)
print(r.returncode)                    # probably None
import time
time.sleep(3)
print(r.returncode)                    # probably 0
```

Popen with PIPE example

```
import subprocess as SP
r = SP.Popen(["dir"], stdout=SP.PIPE, stderr=SP.PIPE, shell=True)
output, error = r.communicate()       # blocking operation
print(output)                         # list of files info
print(error )                         # empty string

r = SP.Popen(["nodir"], stdout=SP.PIPE, stderr=SP.PIPE, shell=True)
output, error = r.communicate()
print(output)                         # empty string
print(error)                          # 'nodir' is not recognized as
an internal or external command...
```

Advanced Python Programming

What Is A Socket?

- **A socket is an endpoint for communicating processes** across a network. The socket mechanism is versatile, supporting both connection oriented communications (stream sockets) providing point to point communications, and connectionless communications (datagram sockets) providing the ability to broadcast information.
- An applications plugs in to the network, sending and receiving data, through a socket.

- The socket mechanism allows the programmer full control over virtually every aspect of the programming. Which protocol will transport the data, the domain, etc.

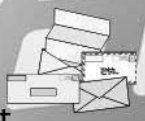
Protocols types - TCP

- Connection Oriented Communications – TCP:
  - The Transmission Control Protocol, better known as TCP, is one of the widely used protocols.
  - Applications using this protocol will connect to each and then pass information, just like when you use the phone.

  - The protocol make the data to be received safely:
    - in order,
    - without errors
    - etc.

Protocols types - UDP

- Connectionless Communications And UDP
  - Applications need not be "connected" in order to communicate. Just as mail can be sent to your home address, data can be sent, in messages called datagrams.
  - The User Datagram Protocol, better known as UDP, is the messenger protocol for this type of communication. This protocol allows directed datagrams and broadcasting.
  - Connectionless communications is generally considered to be less reliable.
    - An application might broadcast on a network that is temporarily down or send a message to a host that is not running.
    - There is no way for the sender to verify that the data has been received on the other side, in order and without errors.



Sockets protocols - Stream Sockets

- Sockets come in two main flavors, one of them being the stream socket.
- Stream socket is full duplex byte stream implementing the connection oriented model.
  - Server sets up a socket with a well known address.
  - Clients connect to the server and data is passed on this open connection.
  - When the client and or server are done the connection is terminated.
- The socket mechanism provides an API for each step on the client and the server. The minimal steps required are:
  - Server side:
    - socket()
    - bind()
    - listen()
    - accept() (possibly repeating the accept())
  - Client side:
    - socket()
    - connect()

Creating sockets

- Sockets are used nearly everywhere. Sockets mechanism in python placed in *socket* module.
- The first step is to create an endpoint, a socket, with socket function:

socket(family=AF_INET, type= SOCK_STREAM, proto=0) Create a new socket using the given address family, socket type and protocol number.

  - The address (and protocol) family can be:
    - socket.AF_INET – for internet sockets ipv4
    - socket.AF_INET6 – for internet sockets ipv6
    - socket.AF_UNIX – unix domain sockets (UNIX IPC), not supported on all platforms

Creating sockets – cont'd

- **Socket type**:
  - socket.SOCK_STREAM
  - socket.SOCK_DGRAM
  - socket.SOCK_RAW
  - socket.SOCK_RDM
  - socket.SOCK_SEQPACKET
- Only sock_stream and sock_dgram appear to be generally useful
- The **proto** argument - this parameter specifies a particular protocol to be used with the socket
  - Normally only a single protocol exists to support socket type within a given protocol family. In this case, *proto* stays default (zero)
    - TCP for stream sockets and UDP for datagram sockets
  - However, it is possible that many protocols may exist, in which case a particular protocol (protocol number) must be specified

Binding The Socket

- The server now has to "bind" the socket
- *socket.bind(address)* - bind the socket to *address*, so that clients can connect to it
- The socket must not already be bound.
- The format of *address* depends on the address family:
  - In INET and INET6 address family, the *address* should be a tuple of ip and port
  - In UNIX address family, the *address* should be a file system path
- For Example:
  - s.bind((socket.gethostname(), 8765))
  - s.bind(('localhost', 8765))

Binding Internet Socket

- The IP
  - Server application can use socket.gethostname() so that the socket would be visible to the outside world.
  - Using 'localhost' or '127.0.0.1' the socket will be only visible within the same machine.
  - Empty sting ('') specifies that the socket is reachable by any address the machine happens to have.
- The port
  - low number ports are usually reserved for "well known" services (HTTP, ftp, telnet, etc).
  - The non-privileged ports value should be at least 4 digits number

Listen for connections on a socket

- After binding a socket to an address the server must create a queue for clients wanting to connect to the server. A connect request from a client is queued until the server accepts the connection.
- *socket.listen(backlog)*
  - The *backlog* argument specifies the maximum number of queued connections
  - The maximum value is system-dependent (usually 5), the minimum value is forced to 0.

Accepting Connections

- The server, having created, bound socket and created a queue for the clients, can now accept requests from clients
- *socket.accept()*
  - The return value is a pair (conn, address) where
    - *conn* is a *new* socket object usable to send and receive data
    - *address* is the address bound to the client
- accept blocks the calling process until a connection is present.

Closing Down The Connection

- The socket can be closed using
  - close
  - shutdown
- socket.close() - close the socket.
  - All future operations on the socket object will fail.
  - This function actually destroy the socket
- *socket.shutdown(how)* - shut down one or both halves of the connection.
  - If *how* is SHUT_RD, further receives are disallowed.
  - If *how* is SHUT_WR, further sends are disallowed.
  - If *how* is SHUT_RDWR, further sends and receives are disallowed. The socket will still be able to receive pending data that already sent

The client

- There is no difference between the client and the server as far as creating the socket is concerned.
- *socket.connect(addr)* – connect to remote socket at address - *addr*

Socket Example – server side

```
Import socket

HOST = '';  PORT = 50007
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(1)
conn, addr = s.accept()
print ('Connected by {}'.format(addr)
while 1:
   data = conn.recv(1024)
   if not data:
          break
   conn.sendall(data)
conn.close()
```

```
import socket

HOSTNAME = 'mydomain.com'
HOST = socket.gethostbyname(HOSTNAME)
PORT = 50007
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))
s.sendall('Hello, world')
data = s.recv(1024)
s.close()
print ('Received: {}'.format( data)
```

Receive data from the socket

- *socket.recv(bufsize)* - receive data from the socket.
- The return value is a string representing the data received. The maximum amount of data to be received at once is specified by *bufsize*.
- The best *bufsize* match with hardware and network realities should be a relatively small power of 2, for example - 1024.

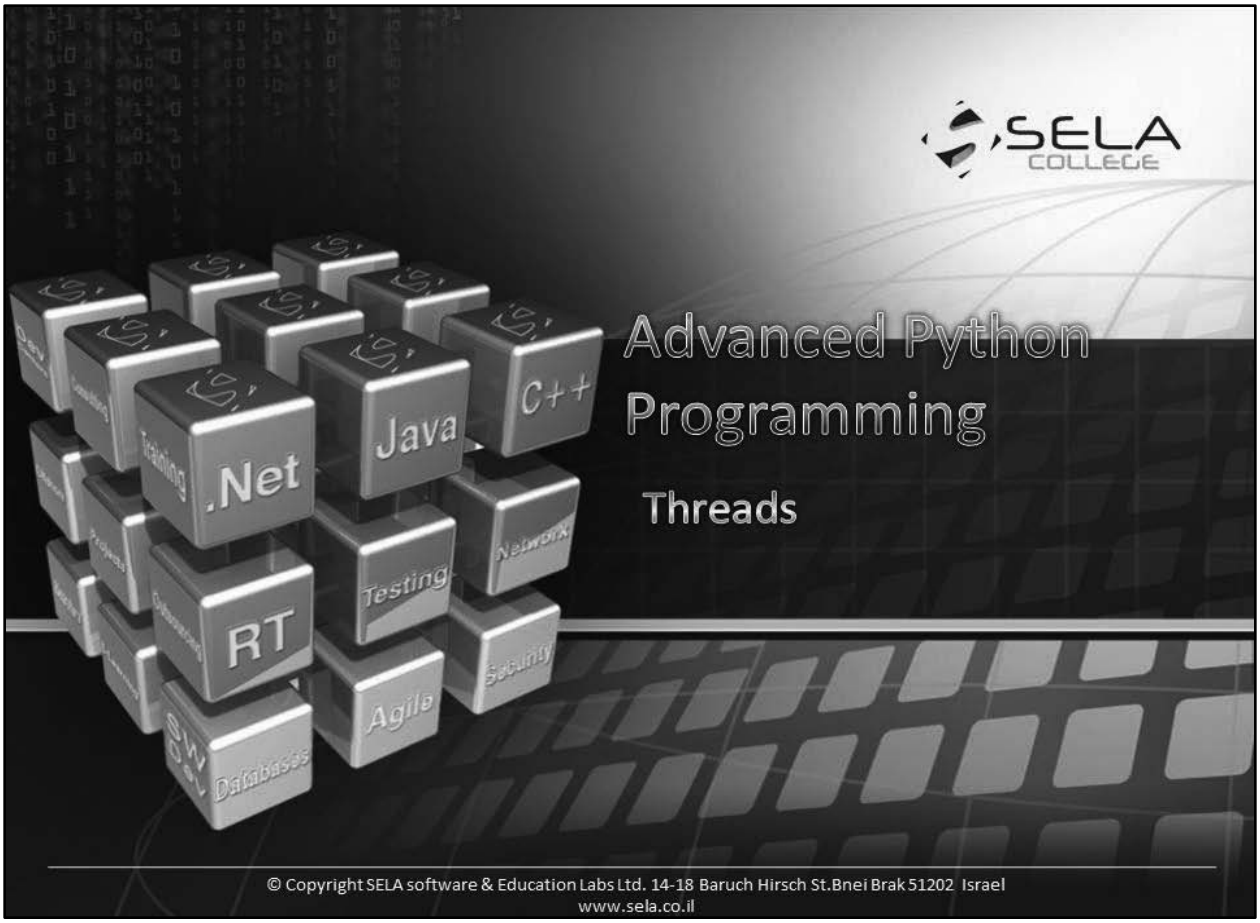
Send data to the socket

- socket.send(*string*) - send data to the socket.
  - Returns the number of bytes sent.
- *socket.sendall(string)* - send data to the socket. Unlike send(), this method continues to send data from *string* until either all data has been sent or an error occurs.
  - None is returned on success.
  - An exception is raised on error
  - In case of error, there is no way to determine how much data, if any, was successfully sent.

Getting host by name

- *socket.gethostbyname(hostname)* - translate a host name to IPv4 address format.
- The IPv4 address is returned as a string, such as '100.50.200.5'.
- If the host name is an IPv4 address itself it is returned unchanged.
- gethostbyname() does not support IPv6 name resolution. Use *getaddrinfo*() instead for

Advanced Python Programming



What is Thread

- A Thread or a Thread of Execution is defined in computer science as the smallest unit that can be scheduled in an operating system
- Threads are contained in processes. More than one thread can exist within the same process.
- These threads share the memory and the state of the process.

threading Module

- *Threading* module is a simple way to create threads. Threading API is very similar to multiprocessing API
- Using threads allows a program to run multiple operations concurrently in the same process space.
- To create a new thread in out program we should use the *Tread* class of *Threading* module

Thread class

Thread(group=None, target=None, name=None, args=())

- *target* is the callable object to be invoked by the Process
- *name* is the process name
- *args* is the argument tuple for the target invocation.
- *group* – should be always be None

- Thread has *start* and *join* functions, exactly like Process does

threading Module example

```
import time
import threading

globalNum = 10
def func():
    global globalNum
    globalNum = 11

thread1 = threading.Thread(target=func)
thread1.start()
thread1.join()
print(globalNum)              # 11
```

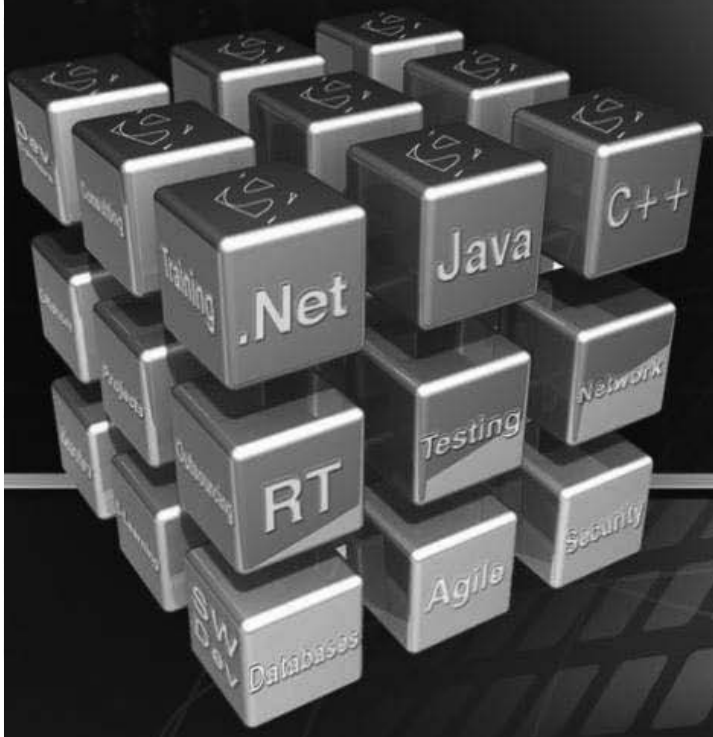
Threading module Synchronization

- threading module has 3 classes for threads synchronization, like multiprocessing module
  - *Lock* - non-recursive lock object
  - *Rlock* - recursive lock object
  - *Semaphore* – created with internal *counter* and can be acquired *counter* times before released

```
lock = threading.Lock()
with lock:
        # critical section code
```

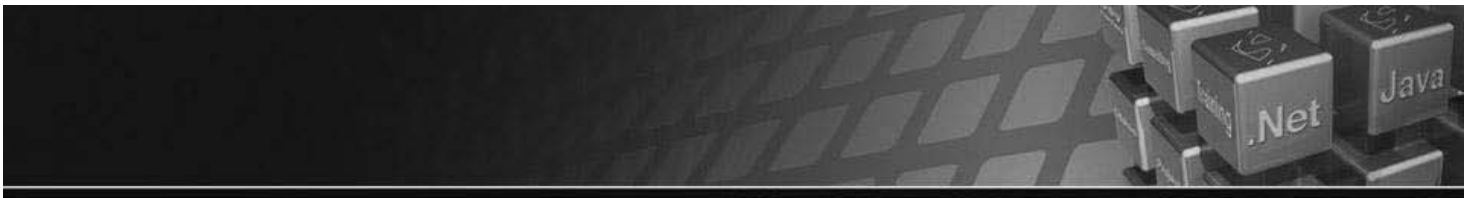
The Global Interpreter Lock

- In CPython, the Global Interpreter Lock (GIL), is a mutex that protects access to Python objects, preventing multiple threads from executing Python bytecodes at once.
- The GIL is controversial because it prevents multithreaded CPython programs from taking full advantage of multiprocessor systems
- There are some GIL free operations, such as I/O and image processing. They happen outside the GIL.
- The multithreaded programs that spend a lot of time inside the GIL, interpreting CPython bytecode, that the GIL becomes a bottleneck



Advanced Python Programming

Introduction to PyQt5



MODULE AGENDA

- About PyQt5
- Basic widgets
- Creating Gui application
- Separation application to classes
- Using Qt Designer

About PyQt5

- Qt library is one of the most powerful GUI libraries. It is available in python as well since python 2.x
- The latest Qt version for python is PyQt5 and it is not backward compatible with PyQt4
- PyQt is a multiplatform toolkit which runs on operating systems like Unix, Linux, Windows and Mac OS
- PyQt module must be installed before it can be used. We can any installation tool, including pip:

  pip install pyqt5

  pip install pyqt5-tools
- PyQt4 is a high level toolkit, written in C++. This is why our GUI programs will be relatively short and simple

- PyQt5 contains several modules:
  - QtWidgets - for UI components (Widgets)
  - QtCore
  - QtGui - for windowing system integration: fonts, text, icons, etc

PyQt5 basic functionality

- Every PyQt5 application must create an application object, this is a requirement of Qt:
  - Many parts of Qt don't work until
  - Application object receives command line arguments passed (sys.argv)

    PyQt5.QtWidgets.QApplication(sys.argv)

- Now we create a main form:

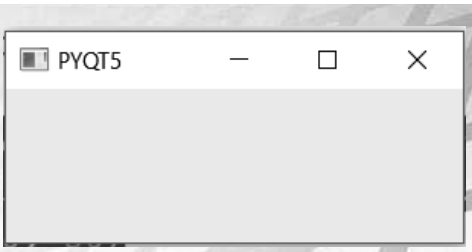
    PyQt5.QtWidgets.Qwidget

- Finally, we enter the mainloop of the application. The event handling starts from this point. The mainloop ends when the main widget is destroyed:

    sys.exit(app.exec_())

PyQt5 first programm

```
from PyQt5 import QtWidgets
import sys

if __name__ ==  "__main__":

    app = QtWidgets.QApplication(sys.argv)
    form = QtWidgets.QWidget()
    form.setWindowTitle("PYQT5")
    form.resize(300,100)
    form.show()
    sys.exit(app.exec_())
```



PyQt5 first program - cont'd

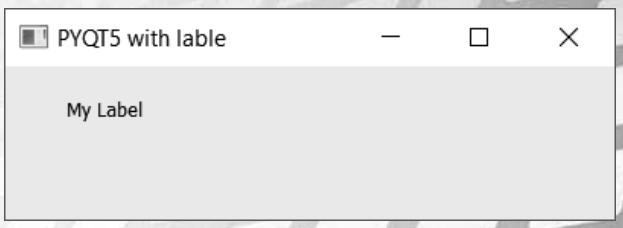
- The show() method displays the main widget on the screen.
- The main widget is first created in memory and then can be shown on the screen using show() method

Logic and implementation

- PyQt applications are usually separate their logic and implementation with object oriented programming styles, by creating a class that responsible for widget initialization and management
- Such a class should derive from *QWidget* class
- It is also recommended to have an inner function, responsible for initialization of all UI components
- The *QT organizer* converts GUI to python code the save way

PyQt5 second program

```
from PyQt5 import QtWidgets
import sys

class WidgetForm(QtWidgets.QWidget):
    def __init__(self):
        super().__init__()
        self._setupUi()

    def _setupUi(self):
        self.setWindowTitle("PYQT5 with lable")
        self.resize(400,100)
        lbl= QtWidgets.QLabel("My Label", self)
        lbl.move(40,20)

if __name__ == "__main__":
    app = QtWidgets.QApplication(sys.argv)
    form = WidgetForm()
    form.show()
    sys.exit(app.exec_())
```

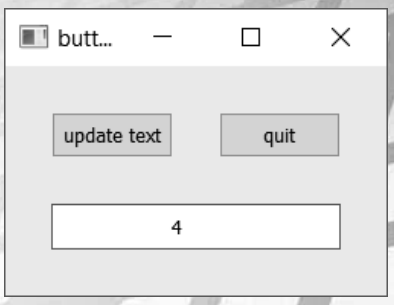


Widgets – Button and TextBox

- QtWidgets module of PyQt5 manages a lot of UI components, such a QLable, QPushButton, QMessageBox, QCheckBox, QRadioButton, QLineEdit, QHBoxLayout, QVBoxLayout and more
- Each control has a set of attributes and slots (events)
- For Example:
  - QPushButton created by with the text to display and the widget to be added to as parameters. Its attributes: resize, move, setGeometry ,setToolTip, setFont, adjustSize, setStyleSheet, etc and its main Slot is *clicked*
  - QLineEdit attributes: move, resize, setText() to set the textbox value, text() to get the value, etc and its main Slot is *textChanged*

Button and TextBox Example

```
class WidgetForm(QtWidgets.QWidget):
    def __init__(self):
        super().__init__()
        self._setupUi()
        self._cnt = 1

    def _setupUi(self):
        self.resize(250,150)
        btn1 = QtWidgets.QPushButton("update text", self)
        btn2 = QtWidgets.QPushButton("quit", self)
        btn1.setGeometry(30,30,80,30)                    # #left, top, width, height
        btn2.setGeometry(140,30,80,30)
        btn1.setToolTip("Click to see the count")
        btn2.setToolTip("Click to exit")
        self._txt = QtWidgets.QLineEdit(self);   self._txt.setGeometry(30,90,190,30)
        btn1.clicked.connect(self._updateText)
        btn2.clicked.connect(QtWidgets.QApplication.instance().quit)

    def _updateText(self):
        self._txt.setText("{:^40}".format(str(self._cnt)))
        self._cnt += 1
```



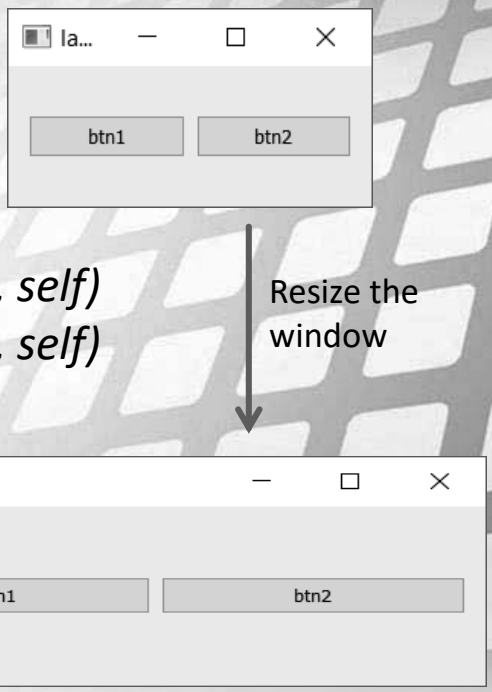
Layout Managment

- When we use the absolute position and size for our widgets they will not change if we resize a window
- Even a small change in layout can cause the complete GUI redesign
- Layouts is a better way to place and organize the widgets on the application window
- There are different layouts in PyQt5
  - HBoxLayout – the horizontal layout
  - VBoxLayout – the vertical layout
  - GridLayout

  PyQt Layouts Attributes are: addWidget , addLayout, setAlignment setSpacing, etc

QHBoxLayout

```
class WidgetForm(QtWidgets.QWidget):
    ……
    def _setupUi(self):
        self.setWindowTitle("layout")
        self.resize(300,200)
        btn1 = QtWidgets.QPushButton("btn1", self)
        btn2 = QtWidgets.QPushButton("btn2", self)
        hlo = QtWidgets.QHBoxLayout()
        hlo.addWidget(btn1)
        hlo.addWidget(btn2)
        self.setLayout(hlo)
```



QVBoxLayout and QHBoxLayout

```
def _setupUi(self):
    btn1 = QtWidgets.QPushButton("btn1", self)
    btn2 = QtWidgets.QPushButton("btn2", self)
    hlo1 = QtWidgets.QHBoxLayout()
    hlo1.addWidget(btn1)
    hlo1.addWidget(btn2)

    btn3 = QtWidgets.QPushButton("btn3", self)
    btn4 = QtWidgets.QPushButton("btn4", self)
    hlo2 = QtWidgets.QHBoxLayout()
    hlo2.addWidget(btn3)
    hlo2.addWidget(btn4)

    vlo = QtWidgets.QVBoxLayout()
    vlo.addLayout(hlo1)
    vlo.addLayout(hlo2)
    vlo.setSpacing(40)
    vlo.setAlignment(Qt.AlignTop)
    self.setLayout(vlo)
```

QGridLayout

- QGridLayout is the most universal layout class.
- It is dynamically divides the space into rows and columns.
- Widgets are added to the grid by specifying the number of row and column.
  - We can specify the spanning factor for rows and columns
  - The grid will expand itself to fit the numbers

addWidget(widget, row, column, [rowSpan, columnSpan])

QGridLayout example


```
def _setupUi(self):
    grid = QtWidgets.QGridLayout()
    grid.addWidget(QtWidgets.QLabel("First Name: "),0,0)
    grid.addWidget(QtWidgets.QLineEdit(),0,1)

    grid.addWidget(QtWidgets.QLabel("Last Name: "),1,0)
    grid.addWidget(QtWidgets.QLineEdit(),1,1)

    grid.addWidget(QtWidgets.QPushButton("Enter"), 2, 0, 1, 2)
    self.setLayout(grid)
    self.setLayout(grid)
```

Widgets Style

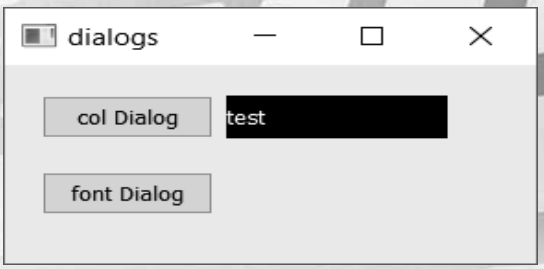
- Python widgets created with default colors and style
- They can be changed using the *setFont* and *setStyleSheet* functions:

```
wig1.setFont(QFont('Arial', 15)
wig2.setStyleSheet("""color:rgb(255,0,255);
        background-color:rgb(200,200,200);""")
```

- *setStyleSheet* can set style attributes like: color, background-color, background-image, border-style, border-width, font, border-radius, border-color, font-size, padding and more
- There are pyqt dialogs that cat help us choose, like *QFontDialog* and *QColorDialog*

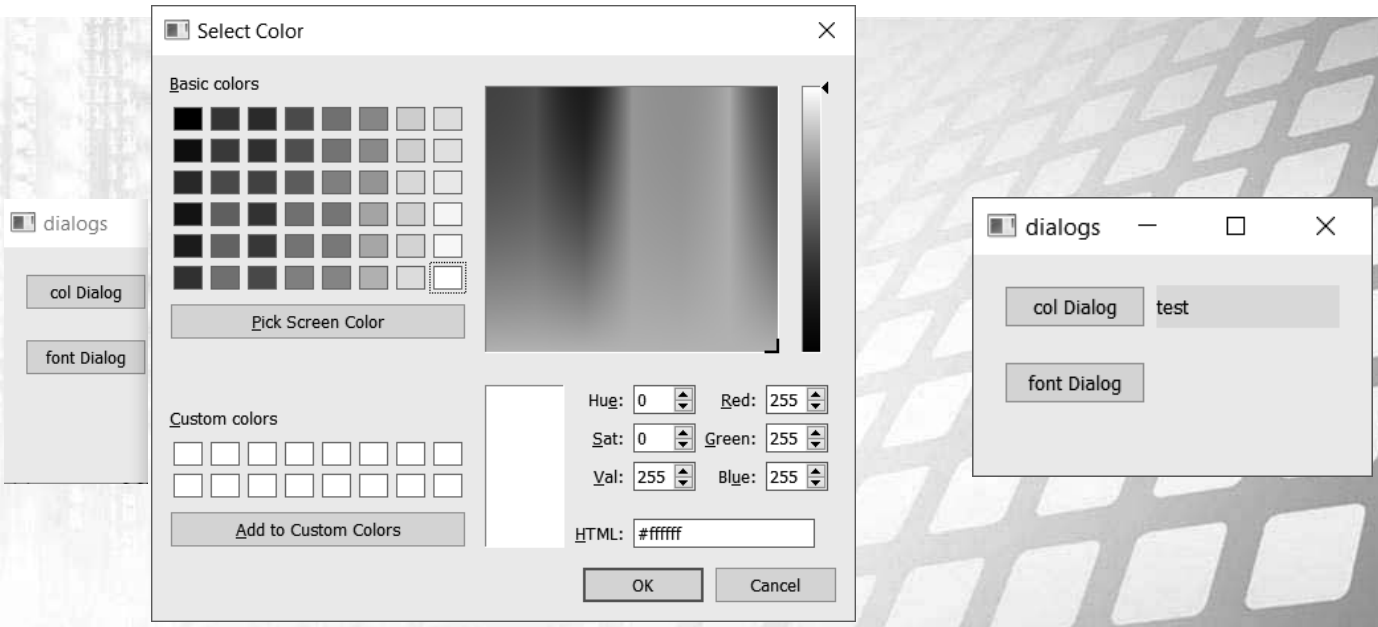
Dialogs and style

```
def initUI(self):
    btn = QtWidgets.QPushButton('col Dialog', self)
    btn.move(20, 20)
    btn.clicked.connect(self.showColorDialog)

    btn1 = QtWidgets.QPushButton('font Dialog', self)
    btn1.clicked.connect(self.showFontDialog)
    btn1.move(20, 70)

    self.lbl = QtWidgets.QLabel("test",self)
    self.lbl.setStyleSheet("""background-color : rgb(0,0,0);
    color:rgb(255,255,255)""")
    self.lbl.setGeometry(120, 20, 120, 28)

    self.setGeometry(300, 300, 450, 180)
    self.setWindowTitle('dialogs')
    self.show()
```

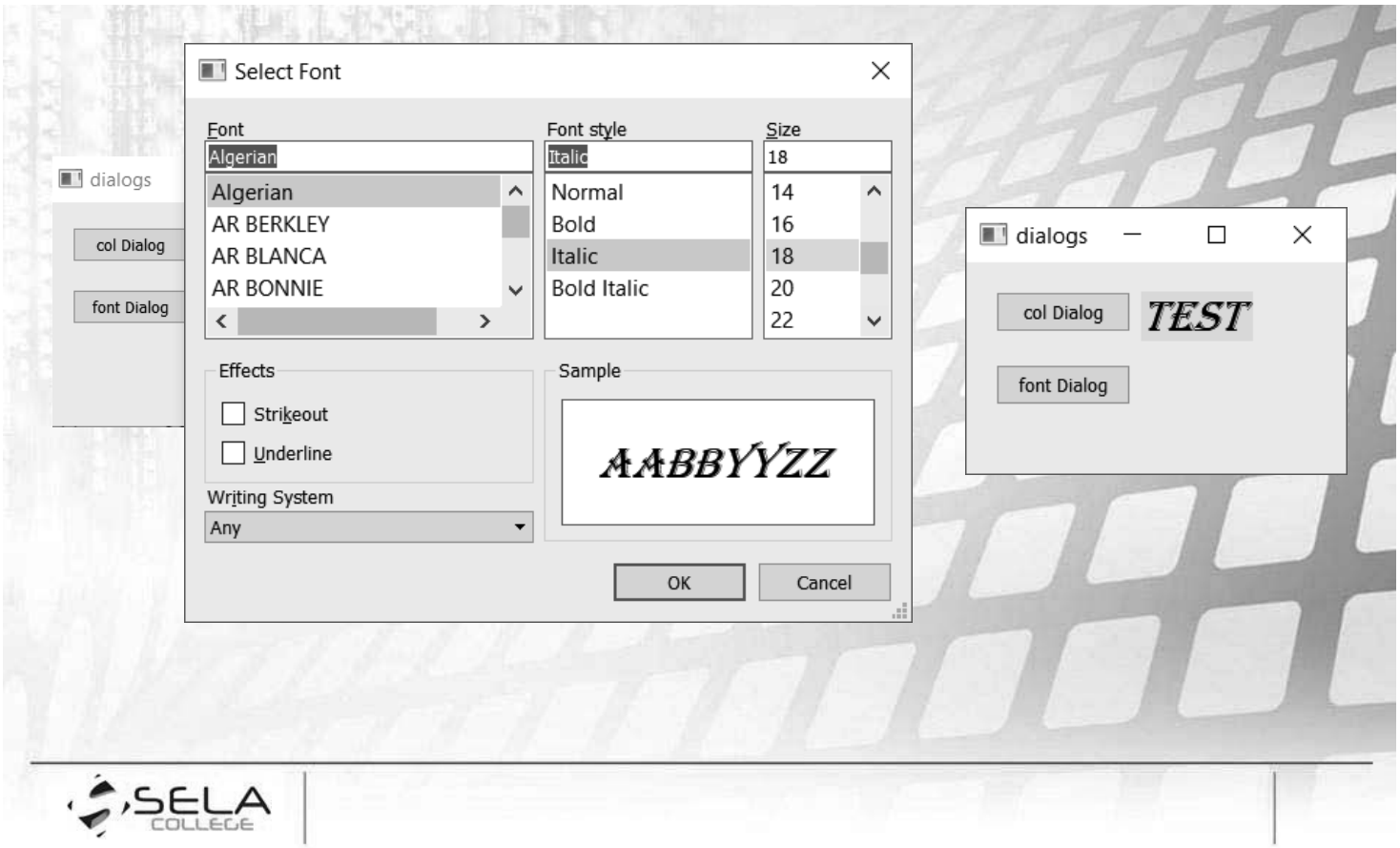


```
def showFontDialog(self):
    font, ok = QtWidgets.QFontDialog.getFont()
    if ok:
        self.lbl.setFont(font)
        self.lbl.adjustSize()


def showColorDialog(self):
    col = QtWidgets.QColorDialog.getColor()
    if col.isValid():
        self.lbl.setStyleSheet("background-color: {}
".format(col.name()))
```


Color dialog



Font dialog

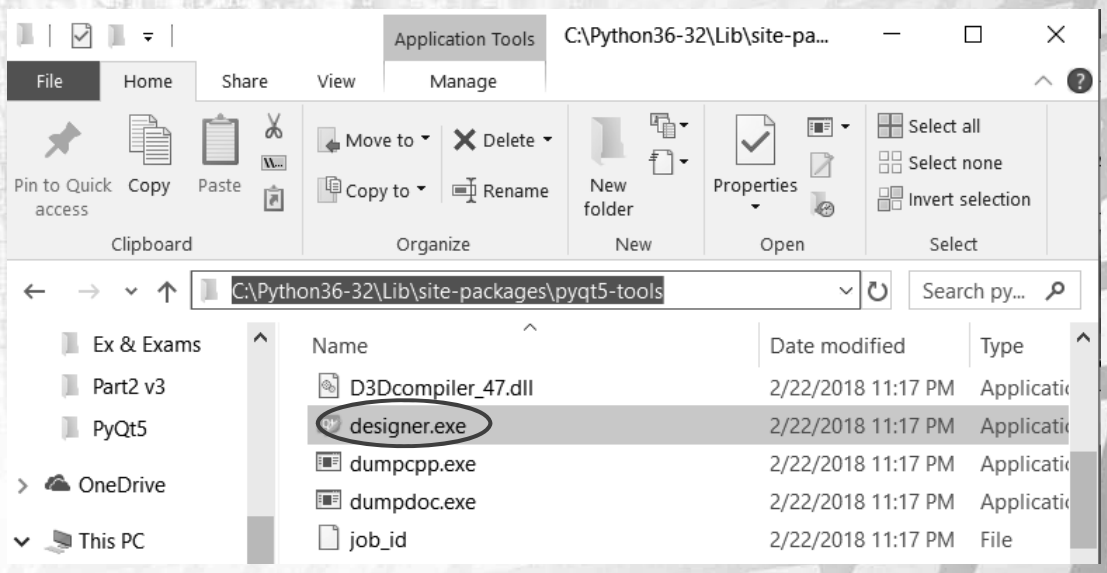


The Qt Designer

- We can use the *Qt Designer* to design our widget form
- Using the *Qt Designer* we can simply drag and drop widgets
- The *Qt Designer* is usually located under: …PythonXX\Lib\site-packages\pyqt5-tools\designer.exe
- Using *Qt Designer* we can create the GUI design, without the code support
- To add the code support we should first save the *Qt Designer* application and then convert it to base python GUI application using the *pyuic5* (python GUI converter) tool

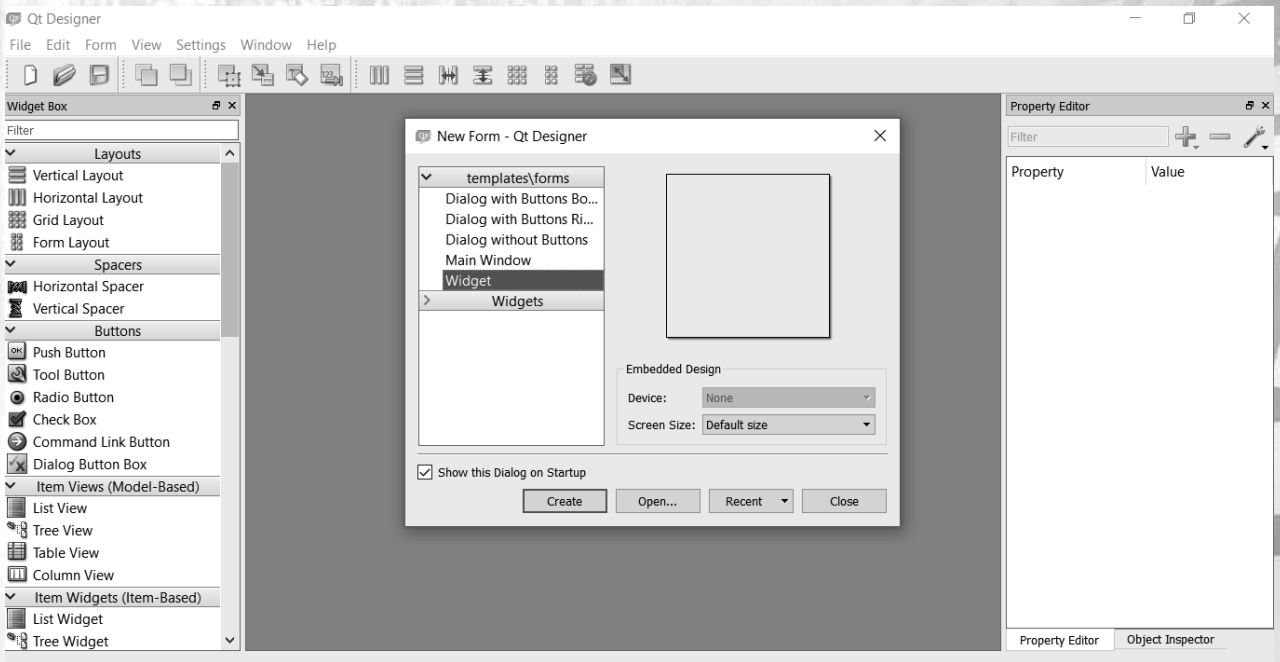
Create Qt Designer application

- The first step is to run the designer.exe



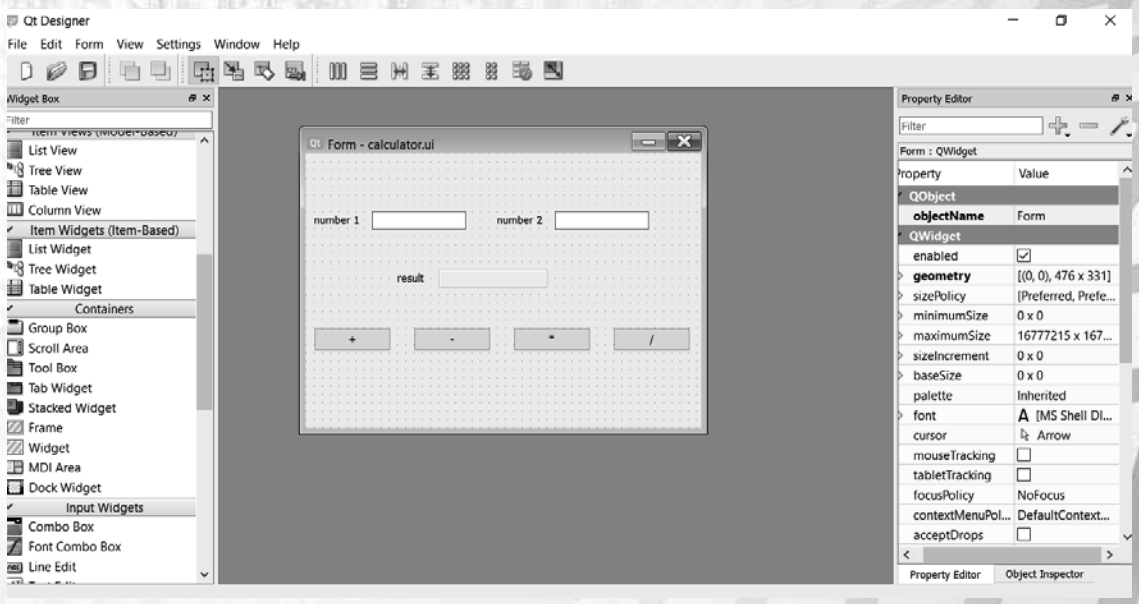
Create Qt Designer application – cont'd

- Pick the type for your application and the location



Create Qt Designer application – cont'd

- Now we can design the widget by drag and drop graphical items on it



- When the design is ready we must save it as *SomeName.ui*

The pyuic5

- pyuic5 (python GUI converter) tool used to convert python-UI design to python code
- Pyuic5 tool usually located in …PythonXX\Scripts\pyuic5.exe
- The easiest way to run the tool is from command line app (shell)
- The syntax on Windows machines is:

  pyuic5.exe SomeName.ui -o SomeDest_ui.py
- The syntax on Unix/Linux machines is:

  pyuic5.exe SomeName.ui > SomeDest_ui.py

- Now the *SomeDest_ui.py* python code file is created. This is a very base version of code, basically responsible for initialization part only.

Important Notes

- Important Notes about the converted code application:
  - Create class derived from *object* and it should be changed to *QtWidgets.QWidget*

    class Ui_Form(object) => class Ui_Form(*QtWidgets.QWidget*)
  - The class has no __init__ function implementation. The function should be added and it should call the ui initialization function in it
  - All the event code support should be added manually as well
  - The code has no main part – it should be added manually
  - All the changes and extensions in the code file must be created only in the final code version. **Each time we change the design and convert it – the conversion is overrides the existed file**.