Python for Data Science

Note that we will focus on particular aspects of Python that would be important for someone who wants to

- ✓ load in some data sets,
- ✓ perform some computations on them,
- ✓ and plot some of the results.

Therefore, we will mostly be talking about Python's built-in data structures and libraries from the perspective of processing and manipulating structured and unstructured data.

Why Python?

The practice of data science involves many interrelated but different activities, including

- accessing data,
- manipulating data,
- computing statistical summaries or business metrics,
- plotting/graphing/visualizing data,
- building predictive and explanatory models,
- evaluating those models, and finally,
- integrating models into production systems

One option for the data scientist is to learn several different software packages that each specialize in one of these things, or to use a general-purpose, high-level programming language that provides libraries to do **all** these things.

Why Python?

Python is **an excellent choice** for this. It has a diverse range of open source libraries for just about everything the data scientist will do. Some of its highlights include:

- **Cross-platform** – high performance python interpreters exist for running your code on almost any operating system (Windows, Mac or Linux) or architecture.
- **Free** – Python and most of its libraries are both open source and free.
- **Simple** – It has efficient high-level data structures and a simple but effective approach to object-oriented programming.
- **Elegant syntax** - which, together with its interpreted nature, makes it an ideal language for scripting and rapid application development in many areas on most platforms

Python: Writing Pythonic Code

- ✓ You will often read questions on StackOverflow like, 'What is a more Pythonic way of doing X.'
- ✓ To know what that means, read **The Zen of Python.** Simply run `import this` on any Python interface.
- ✓ It is a description of its design principles, and code written using these principles is called 'Pythonic.'
- ✓ While there are typically multiple ways to crack a given problem, we will generally favor Pythonic solutions over shabby ones.

Python: Package Managers

Do read about **pip** and **conda** – both of which will act as your package/library managers.
To install libraries that aren't part of the Anaconda distribution, you will be using commands such as

- `pip install ggplot`
- `conda install ggplot`

What do you mean Python Basics?

Python Basic programming topics

- ✓ Basic Rules
- ✓ Declaring & Printing variables
- ✓ Objects, Methods, Attributes and Functions
- ✓ Using built-in functions
- ✓ Modules (Libraries)
- ✓ Data Types
- ✓ Basic Operators: Arithmetic using binary operators
- ✓ Dealing with Strings
- ✓ Control flow statements
- ✓ Control Flow with if, elif, else
- ✓ Loops
- ✓ Data Structures
- ✓ Working with Collections - List, Tuple, Set & Dictionary
- ✓ Functions – User defined functions
- ✓ Lambda functions
- ✓ Classes

Python: Basic Rules

Comments:
Any text preceded by the hash mark (pound sign) # is ignored by the Python interpreter.

Whitespace Formatting (Indentation):
Many languages like R, C++, Java, and Perl use curly braces to delimit blocks of code. Python uses whitespace indentation to make code more readable and consistent. A colon denotes the start of an indented code block after which all of the code must be indented by the same amount

One major reason th at whitespace matters is th at it results in most Python code looking cosmetically similar, which means less cogniti ve dissonance when you read a piece of code that you didn't wri te yourself.

Python uses indentation for blocks, instead of curly braces. Both tabs and spaces are supported, but the standard indentation requiresstandard Python code to use four spaces. For example

Declaring & Printing variables

Variable are dynamically typed, so no need to mention the variable types. Python interpreter can automa tically infer the type when the variables are initialized. The simplest directive in Python is the "print" directive- it simply prints outa line

```
In [1]: var1 = 2
        var2 = 5.0

In [2]: var1
Out[2]: 2

In [3]: type( var1 )
Out[3]: int

In [4]: type( var2 )
Out[4]: float
```

```
In [5]: print( var1 )
        2

In [6]: mystring = 'This is python'
        print( mystring )
        This is python

In [7]: print( var1, var2, mystring )
        2 5.0 This is python
```

There is difference between Python 2 and 3 for the print statement. In Python 2, the "print" statement is not a function, and therefore it is invoked without parentheses. However, in Python 3, it is a function, and must be invoked with parentheses.

Objects, Methods, Attributes and Functions

Everynumber, string,datastructure, function, class,module,and soonexistsinthe Pythoninterpreteris referredto as a **Python object**.

Each object has an associated

- **type** (int,float,list,dict,str andsoon…)
- attachedfunctions, known as **methods**,
  - these haveaccess to theobject's internal data.
    - They canbecalledusingthesyntax:*obj.<method>(parameters)*
- **attributes** which can beaccessed with the syntax:*obj.attribute*

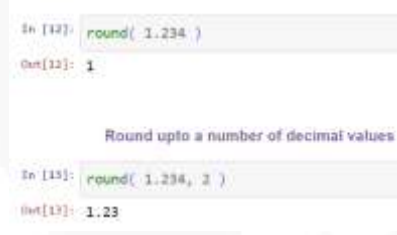
"Functions are called using parentheses and passingzeroor more arguments, optionallyassigningthereturned valueto a variable: *result=f(x,y,z)*"

Let's discuss classes & objects later oncewe havedonesome basictopics inpython

Built in functions

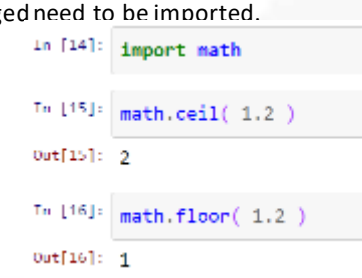
Functions comes with pythonbase version, called built in functions.

Example:round()



To invokesome functions that packaged need to be imported.

For example import a math function



Modules (Libraries) - Packages

Certain functions in Python are not loaded by default.

These include both features included as part of the language as well as third-party features that you download explicitly. In order to use these features, you'll need to **import the modules** that contain them.

>In Python a module is simply a .py file containing function and variable definitions. You

can import the module itself as: `import pandas`

But after this you'll have to always access its functions by prefixing them with the module name,

For example : `pandas.Series()`
Alternatively, we can provide an alias: `import pandas as pd`

This will save us some typing as we can then write `pd.Series()` to refer to the same thing.
Another option is to import frequently used functions explicitly and use them without any prefixes. For example,

```
from pandas import Series
```

Tip: Importing everything from a module is possible, but is considered bad practice as it might interfere with variable names and function definitions in your working environment.
So **avoid** doing things like: `from pandas import *`

Modules (Libraries) - Packages

Exploring built-in modules:

- ✓ Two very important functions come in handy when exploring modules in Python - the dir and help functions.
- ✓ We can look for which functions are implemented in each module by using the dir function
- ✓ When we find the function in the module we want to use, we can read about it more using the help function, inside the Python interpreter

Writing modules

- ✓ Writing Python modules is very simple. To create a module of your own, simply create a new .py file with the module name, and then import it using the Python file name (without the .py extension) using the import command.

Writing packages

- ✓ Packages are namespaces which contain multiple packages and modules themselves. They are simply directories, but with a twist.
- ✓ Each package in Python is a directory which **MUST** contain a special file called __init__.py. This file can be empty, and it indicates that the directory it contains is a Python package, so it can be imported the same way a module can be imported.
- ✓ If we create a directory called foo, which marks the package name, we can then create a module inside that package called bar. We also must not forget to add the __init__.py file inside the foo directory.

Data Types

Python supports two types of numbers –integers and floating point numbers. (It also supports complex numbers, which will not be explained in this tutorial).

Python has a small set of built-in types for handling numerical data, strings, boolean (True or False) values, and dates and time. These include

- *None* –The Python Null Value
- *str, unicode* –for strings
- *int* –signed integer whose maximum value is platform dependent.
- *long* –large ints are automatically converted to long
- *float* –64–bit (double precision) floating point numbers
- *bool* –a True or False value

You could call the function *type* on an object to check if it is an int or float or string etc.
Type Conversion can be achieved by using functions like `int()`, `float()`, `str()` on objects of other types.

Basic Operators: Arithmetic using binary Operators

Just as any other programming languages, the addition, subtraction, multiplication, and division operators can be used with numbers. Most of the binary math operations and comparisons are as you might expect:
1 + 23; 5 – 7; 'This' + 'That'

Operation	Description
a + b	Add a and b
a – b	Subtract b from a
a * b	Multiply a by b
a / b	Divide a by b
a // b	Floor–divide a by b, dropping any fractional remainder
a ** b	Raise a to the b power
a & b	True if both a and b are True. For integers, take the bitwise AND.
a | b	True if either a or b is True. For integers, take the bitwise OR.
a ^ b	For booleans, True if a or b is True, but not both. For integers, take the bitwise EXCLUSIVE-OR.
a == b	True if a equals b
a != b	True if a is not equal to b
a <= b	True if a is less than (less than or equal) to b
a < b a > b	True if a is greater than (greater than or equal) to b
a >= b a is b	True if a and b reference same Python object
a is not b	True if a and b reference different Python objects

Arithmetic using Binary Operators

NOTE that Python 2.7 uses **integer division by default**, so that 5 / 2 equals 2. Almost always this is not what we want, so we have two options:

- Start your files with *from __future__ import division*
- Explicitly convert your denominator to a float as *5/float(2)*

However, if for some reason you still want integer division, use the **//** operator.

Arithmetic using Binary Operators

Strings

"Many people use Python for its powerful and flexible built-in string processing capabilities. You can write string literal using either single quotes or double quotes, but multiline strings are defined with triple quotes. The difference between the two is that using double quotes makes it easy to include apostrophes

```
a = 'one way of writing a string' b = "another way"
c = """This is a multiline string"""
```

Strings are

- **sequences** of characters, and so can be treated like other Python sequences (for iteration)
- **immutable,** you cannot modify them in place without creating a new string
- can contain escape characters like \n or \t
  - there's a workaround if you want backslashes in your string: prefix it with **r** (for **raw**)
- **concatenated** by the + operator, try `'This' + ' and ' + 'That'`

Here I will highlight a few cool **string methods** as a teaser to what you can do with Python

```
my_str = 'a, b, c, d, e'
my_str.replace('b', 'B')
my_str.split(',') '-'.join(my_str.split(', '))
```

Arithmetic using Binary Operators

Strings Formatting:

Python uses C-style string formatting to create new, formatted strings. The "%" operator is used to format a set of variables enclosed in a "tuple" (a fixed size list), together with a format string, which contains normal text together with "argument specifiers", special symbols like "%s" and "%d".

Let's say you have a variablecalled "name" with your user name in it, and you would then liketo print(out a greeting to that user.)

```
# This prints out "Hello, ALabs!“
name = “ALabs“
print("Hello,%s!" % name)
```

To use two or more argument specifiers,use a tuple (parentheses):

```
# This prints out “ALabs is 4 years old.“
name = “ALabs“
age = 4
print("%sis %d years old." % (name,age))
```

Arithmetic using Binary Operators

Strings Formatting:

Any object which is not a string can be formatted using the %s operator as well. The string which returns from the "repr" method of that object is formatted as thestring.

For example:

```
# This prints out:A list: [1,2,3]
mylist = [1,2,3]
print("A list:%s" % mylist)
```

Here are somebasic argumentspecifiers you should know:

- %s - String (or any object with a string representation, like numbers)
- %d - Integers
- %f - Floating point numbers
- %.<number of digits>f - Floating point numbers with a fixed amount of digits tothe right of the dot.
- %x/%X - Integers in hex representation (lowercase/uppercase)

Dealing with strings - Examples

```
In [91]: string0 = 'python'
         string1 = "Data Science"
         string2 = '''This is Data science
                 workshop
                 using Python'''

In [92]: print( string0, string1, string2)

python Data Science This is Data science
       workshop
       using Python

In [93]: string2.find( "Python" )
Out[93]: 53

In [94]: string0.capitalize()
Out[94]: 'Python'

In [95]: string0.upper()
Out[95]: 'PYTHON'

In [96]: len( string2 )
Out[96]: 59

In [97]: string2.split()
Out[97]: ['This', 'is', 'Data', 'science', 'workshop', 'using', 'Python']

In [98]: string2.replace( 'Python', 'R')
Out[98]: 'This is Data science \n        workshop\n        using R'
```

Control Flow with if, elif, else

Python uses boolean variables to evaluate conditions. The booleanvalues True and False are returned when an expressionis compared or evaluated.

"The if statement is one of the most well-known control flow statement types. It checks a conditionwhich,if True, ev aluates the code in the blockthat follows:

```
if x < 0:
    print 'It's negative'
```

An if statement can be optionally followed by one ormore elif blocks anda catch-allelse block if all of the c onditionsare False:

```
if x < 0:
    print 'It's negative' elif x == 0:
    print 'Equal to zero' elif 0 < x < 5:
    print 'Positive but smaller than 5' else:
    print 'Positive and larger than or equal to 5'
```

If any of the conditions is True, nofurtherelif or else blockswill be reached.

```
In [28]: x = 10
         y = 12
         if x > y:
           print ("x>y")
         elif x < y:
           print ("x<y")
         else:
           print ("x=y")

x<y
```

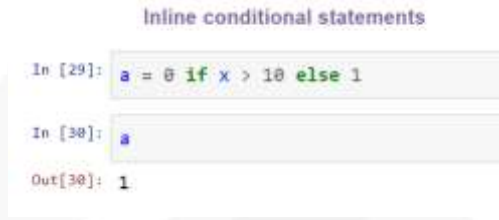
Compound Logic

We can write **compound logic** using boolean operators like **and, or**. Remember that conditions are evaluated left-to-right and will short circuit, i.e, if a True is found in an **or** statement, the remaining ones will not be tested.

```
if 5 < 10 or 8 > 9:
     print 'The second condition was ignored.'
```

You can also write a **ternary if-then-else** on one line, which sometimes helps keep things concise, These statements called as inline statements

```
parity = "even" if x % 2 == 0 else "odd"
```



Control flow statements - Loops

There are two types of loops in Python, for and while.

for Loops: For loops iterate over a given sequence.

These are meant for iteration tasks over a collection (a Python data structure like a Tuple or List.)

Syntax:

```
for value in collection:
    # do something with value
```

Example: Here we print out the squares of the first five natural numbers

```
for x in [1, 2, 3, 4, 5]: print x ** 2
```

- The **continue** keyword advances the **for** loop to the next iteration –skipping the remainder of the block.

For loops can iterate over a sequence of numbers using the "range" and "xrange" functions.
The difference between range and xrange is that the range function returns a new list with numbers of that specified range, whereas xrange returns an iterator, which is more efficient. (Python 3 uses the range function, which acts like xrange).
Note that the range function is zero based.

Control flow statements - Loops

Example: The following loop sums up values, ignoring instances of **None**

```
total = 0
for value in [1, 2, None, 4, None, 5]: if value is None:
        continue total += value
```

- The **break** keyword is used to altogether exit the for loop. Example: This code sums elements of the list until a 5 is reached:

```
until_5 = 0
for value in [1, 4, 2, 0, 7, 5, 1, 4]: if value == 5:
        break until_5 += value
```

Control flow statements - Loops

while Loops: While loops repeat as long as a certain boolean condition is met.

Pyth on has a while loop as well, which works as expected.

```
x = 0
while x < 10:
    print x, "is less than 10" x += 1
```

```
In [31]: for i in range(5):
           print (i)

0
1
2
3
4
```

```
In [32]: i = 1
         while i < 5:
           print(i)
           i = i+1
         print('Bye')

1
2
3
4
Bye
```

```
In [33]: i = 1
         while i < 5:
           print(i)
           i = i+1
           if i == 4:
               break
         print('Bye')

1
2
3
Bye
```

```
In [34]: i = 1
         while i < 5:
           i = i+1
           if i == 3:
               continue
           print(i)
         print('Bye')

2
4
5
Bye
```

Control flow statements - Loops

"break" and "continue" statements:
break is used to exit a for loop or a while loop, whereas **continue** is used to skip the current block, and return to the "for" or "while" statement.

```
# Prints out 0,1,2,3,4

count = 0
while True:
    print(count)
    count += 1
    if count >= 5:
        break

# Prints out only odd numbers - 1,3,5,7,9
for x in range(10):
    # Check if x is even
    if x % 2 == 0:
        continue
    print(x)
```

can we use "else" clause for loops?
we can use **else** for loops. When the loop condition of "for" or "while" statement fails then code part in "else" is executed.

If **break** statement is executed inside for loop then the "else" part is skipped. Note that "else" part is executed even if there is a **continue** statement.

```
# Prints out 0,1,2,3,4 and then it prints "count value reached 5"

count=0
while(count<5):
    print(count)
    count +=1
else:
    print("count value reached %d" %(count))

# Prints out 1,2,3,4
for i in range(1, 10):
    if(i%5==0):
        break
    print(i)
else:
    print("this is not printed because for loop is terminated because of break but not due to fail in condition")
```

Data Structures - Tuples

Python Data Structures are simple, but quite powerful. Understanding them well and mastering their use is critical for a programmer to write effic ient code for doing data science. Here we will learn about Tuples, Lists and Dictionaries – each of which are characterized by how data is stored in them, and the use-cases they're most suitable for.

TUPLES: A **tuple** is a sequence of Python objects that is
- one-dimensional,
- fixed-length,
- immutable.

Syntax: We can create tuples in two ways

-A comma-separated sequence of values assigned to a variable (optional: placed inside parentheses)
Calling `tuple()` on any sequence/iterator (eg. a list)

```
tup = 1, 2, 6
nested_tup = (1, 3, 5), (2, 8), ['a', 'b']

tuple([1, 2, 7]) tuple('forests')
```

Data Structures - Tuples

Subsetting: Elementscanbe accessedwithsquare brackets `[]`,with indexes beginningwith 0.

```
nested_tup[0]
```

Immutability: Once a tuple is created, it'snot possible to modify whichobject is stored at an index.

```
nested_tup[1] = (1, 3)
TypeError: 'tuple' object does not support item assignment
```

[*Note*] The **objects** stores in the tuple (eg. a list) might be mutable, so they can be altered (but not moved.)

```
nested_tup[2].append('c')
```

Concatenation: The '+'operator joinstuples to formlongertuples.

```
(4,None,'foo')+(6,0)+('bar',)
```

Data Structures - Tuples

Tuple Unpacking: In an assignment statement, corresponding elements of a tuple will be assigned to respective objects on the RHS (given that the number of objects is the same as length of the tuple.) This makes is very easy to swap variables.

```
a, b, c = (1, 2, 3)
a, b = b, a
```

Tuple Methods: Press <tab>followinga dot after the tupleobject.Though there aren't too many Tuple methods, *count* is one of the useful ones.

```
nested_tuple.count()
```

Data Structures - Lists

A Python **list** is simply an ordered collection of values or objects. It is similar to what in other languages might be called an *array*, but with some added functionality.
Lists are very similar to arrays. They can contain any type of variable, and they can contain as many variables as you wish. Lists can also be iterated over in a very simple manner.

Lists are

- **one-dimensional**
  - containers for collections of objects of **any type**
- **variable-length**
  - **mutable**, ie, their contents can be modified

Syntax: They can be defined using

- square brackets `[]` or
- using the `list()` type function
- Python functions that produce lists

```
int_list = [1, 2, 3]
mix_list = ["string", 0.1, True]
list_of_lists = [int_list, mix_list, ['A', 'B']] x = range(10)
```

Data Structures - Lists

Single elements can be accessed using their index or position

```
x[4]      # fetches the 5th element
x[-1]     # fetches the last element
```

Subsets of lists (smaller lists) can be accessed using **integer slicing**

```
x[:4]     # first four elements
x[4:]     # all elements from fourth to the end
x[-3:]    # last three elements
```

Data Structures - Lists

1. Adding elements

a. `append()`, to add single elements **at the end** of the list. For example:
`x.append('a')`

b. `extend()`, to add multiple element **at the end** of an existing list. For example:
`x.extend([10, 11, 12])`

[Note] Lists can be combined/concatenated using the + operator. For example:
`[1, 2, 3] + ['a', 'b', 'c']`

c. `insert()`, to insert elements at a specific index (this is an expensive operation) For example:
`x.insert(3, 'a')`

2. Removing elements

a. `pop()`, removes and returns an element at a particular index (default : from the end) For example:
`x.pop(5)`

b. `remove()`, takes an element as input and removes its first occurrence For example: `x.remove(5)`

2. Sorting

a. `.sort()`

Data Structures - Lists

List Functions

x = list('just a string')

- len(), returns the number of elements in the list For example:
  len(x)
- in, checks whether an element belongs to a list and returns a boolean For example:
  't' in x
- sorted(), returns a new list from the elements of given list, sorted in ascending order For example:
  sorted(x)
- reversed(), iterates over the elements of a sequence in reverse order For example:
  reversed(x)

List Unpacking works a lot like tuple unpacking, where you can assign list elements to objects using an assignment statement.

For example: a, b, c, d = [1, 2, 3, 4]

Data Structures - Dictionary

Dictionary (or dict)

dict is likely the most important built-in Python data structure. It is a flexibly-sized collection of **key-value pairs**, where *key* and *value* are Python objects.

We frequently use dictionaries as a simple way to represent structured data.

```
doc_info = {
    "author": "chandra",
    "title": "Data Science in Python", "chapters": 10,
    "tags": ["#data", "#science", "#datascience", "#python", "#analysis"]
}
```

Syntax: dicts are created using curly braces {} and using colons to separate keys and values.

```
empty_dict = {}
a_dict = {'k1':12, 'k2':36}
b_dict = {'a': 'a string', 'b': [1, 2, 3, 4]}
```

Data Structures - Dictionary

Subsetting: We can access or insert the value/object associated with a key by using the curly brackets

```
b_dict[a]              # retrieves 'a string'
b_dict['c'] = 3.142    # adds a new key-value pair to the dict
```

[Note] We can check if a dict contains a key using the **in** keyword

```
'd' in b_dict
```

Data Structures – Dictionary methods

1. Removing/Adding elements
   a. `.keys(), .values(), .items()` – will return the keys, values, pairs of the dict

   For example:
   ```
   a_dict.keys()
   a_dict.values()
   a_dict.items()
   ```

   b. `.del()` – Will remove the key–value pair associated with passed key For example: `del b_dict['a']`

   c. `.pop()` – works in much the same fashion For example: `b_dict.pop('c')`

   d. `.update()` – will merge two given dictionaries

   For example: `b_dict.update(a_dict)`

2. Finding elements
   a. `.get()` – behaves gracefully for missing keys. It is used to fetch a value from a dict. If the key is found, it returns the associated value. It returns a default value if the key isn't contained in the dict. This ensures that an exception is not raised.
   For example: `b_dict.get('d', 'Nothing found')`

Data Structures – Sets

A set is an unordered collection of unique elements. They can be thought of being like dicts, but keys only, no values.

Syntax: A set can be created in two ways:

- via the set function,
- using a set literal with curly braces:

```
set([2, 2, 2, 1, 3, 3])       # produces set([1, 2, 3])
{2, 2, 2, 1, 3, 3}            # produces set([1, 2, 3])
```

Sets support mathematical set operations like union, intersection, difference, and symmetric difference."

User Defined functions in Python

Functions are a convenient way to divide your code into useful blocks, allowing us to order our code, make it more readable, reuse it and save some time. Also functions are a key way to define interfaces so programmers can share their code.

Python makes use of blocks. Where a block line is more Python code (even another block), and the block head is of the following format: block_keyword block_name(argument1,argument2, ...) Block keywords you already know are "if", "for", and "while".

Functions in Python:

- ✓ Functions in python are defined using the block keyword "def", followed with the function's name as the block's name
- ✓ Parameters types are not defined. The types are inferred from values passed to the function.
- ✓ Python functions overloading is implicit
- ✓ Functions may also receive arguments (variables passed from the caller to the function)
- ✓ Functions may return a value to the caller, using the keyword- 'return'. Functions can return multiple parameter
- ✓ If only one or few returned parameters need to be captured and other ignored
- ✓ Simply write the function's name followed by (), placing any required arguments within the brackets.

User Defined functions in Python

Python functions can be optional

The default value for the parameters can be defined in function signatures.

```
In [99]: def addElements( a, b ):
             return a + b

In [100]: addElements( 2, 3 )
Out[100]: 5

In [101]: addElements( 2.3, 4.5 )
Out[101]: 6.8

In [102]: addElements( "python", "workshop" )
Out[102]: 'pythonworkshop'
```

```
In [103]: def addElements( a, b ):
              return a, b, a + b

In [104]: x, y, z = addElements( 2, 3 )

In [105]: addElements( 2.3, 4.5 )
Out[105]: (2.3, 4.5, 6.8)

In [106]: _, _, z = addElements( 4, 5 )

In [107]: x
Out[107]: 2
```

```
In [108]: def addElements( a, b = 4 ):
              return a + b

In [109]: addElements( 2 )
Out[109]: 6

In [110]: addElements( 2, 5 )
Out[110]: 7

In [111]: def add_n(*args):
              sum = 0
              for arg in args:
                  sum = sum + arg
              return sum

In [112]: add_n( 1, 2, 3 )
Out[112]: 6

In [113]: add_n( 1, 2, 3, 4, 5, 6 )
Out[113]: 21

In [114]: add_n()
Out[114]: 0
```

Lambda functions in Python

- ✓ Lambda functions in python are key features. These are functions that can be passed as parameters to another functions.
- ✓ The functions can be anonymous and defined inline, while passing as a parameter.
- ✓ Primarily used to deal with collections, to apply a function or operations on each individual elements of python

```
In [115]: a = lambda x: x * x

In [116]: a( 2 )
Out[116]: 4

In [117]: a( 2 ) * a( 2 )
Out[117]: 16
```

```
In [118]: mylist = [1,2,3,4,5,6,7,8,9]

In [119]: xsquare = []

          for x in mylist:
            xsquare.append( pow( x, 2 ) )

          print( xsquare )
          [1, 4, 9, 16, 25, 36, 49, 64, 81]

In [120]: map( lambda x: pow( x, 2 ), mylist)
Out[120]: <map at 0x45c9fd8>

In [121]: xsquare1 = list( map( lambda x: pow( x, 2 ), mylist) )

In [122]: print( xsquare1 )
          [1, 4, 9, 16, 25, 36, 49, 64, 81]

In [123]: mylist1 = [1,2,3,4,5,6,7,8,9]

In [124]: listprods = list( map( lambda x, y: x * y, mylist, mylist1 ) )

In [125]: listprods
Out[125]: [1, 4, 9, 16, 25, 36, 49, 64, 81]

In [126]: list( filter( lambda x : x < 5, listprods ) )
Out[126]: [1, 4]
```

Classes & Objects

- ✓ Objects are an encapsulation of variables and functions into a single entity. Objects get their variables and functions from classes. Classes are essentially a template to create your objects.

Example:

We have a class defined for Student with student details like name and age. Create new student called student1. Set student1 to be with name as chandra as age as 33.
Return the results as

"chandra is 33 years old and participating in python class"

```
In [2]: class Student:
          workshop = 'python'
          def __init__(self,name,age):
              self.name = name
              self.age = age
          def describe( self ):
              print( self.name, " is ",
                    self.age,
                    " years old and participating in ",
                    Student.workshop,
                    " class ")
              return

In [8]: student1 = Student( "Chandra", 33 )

In [9]: student1.name
Out[9]: 'Chandra'

In [10]: student1.describe()
         ('Chandra', ' is ', 33, ' years old and participating in ', 'python', ' class ')

In [6]: Student.workshop = "python"

In [11]: student1.workshop
Out[11]: 'python'
```

Resources to Learn Python

Python Resources

- ✓ Workshop material
  - ✓ Presentations
  - ✓ Sample codes
  - ✓ casestudies
  - ✓ Projects
- ✓ https://github.com/ipython/ipython/wiki/A-gallery-of-interesting-IPython-Notebooks
- ✓ http://stackoverflow.com/tags/python/info

  http://learnpythonthehardway.org/book/
- ✓ "Web Scraping with Python: Collecting Data from the Modern Web", O'Reilly