Lakehead University - Faculty of Engineering
Python and Jupyter Notebooks

Instructor: Laura Curiel

1 Introduction

Python is a general-purpose interpreted, interactive, object-oriented, and high-level programming language. It was created by Guido van Rossum during 1985-1990. Python source code is available under the GNU General Public License (GPL) and it is therefore an open-source tool.

The Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and explanatory text. For our purposes, we will use it to share and use documents with live Python code on material in the course with visualization and text.

This tutorial will allow you to:
1) learn the basics of Python programming
2) learn how to run a Jupyter notebook

It requires you to have some basis of C, Matlab or any other structured programming language

All instructions in this tutorial guide you on the installation and execution in a Linux platform. You can attempt to use similar instructions for a Mac or Windows environment but I will not be providing guidance during lectures on these platforms.

2 History and Basic Principles

Python is a high-level, interpreted, interactive and object-oriented scripting language. It uses English keywords frequently and it has fewer syntactical constructions than other languages.

1. Python is Interpreted: You do not need to compile your program before executing it.
2. Python is Interactive: You can type commands directly on a command line.
3. Python is Object-Oriented: programming encapsulates code within objects.
4. Python has currently a wide range of applications and libraries available from simple text processing to WWW browsers to games.
5. Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.
6. Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
7. Python is extensible: modules can be added to enable programmers to add to or customize their tools to be more efficient.
8. Python provides interfaces to all major commercial databases.
9. Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows, Macintosh, and the X Window system of Unix.

10. Python can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.

Python was developed by Guido van Rossum in the late eighties and early nineties at the National Research Institute for Mathematics and Computer Science in the Netherlands. It is derived from many other languages, including ABC, Modula-3, C, C++, Algol-68, SmallTalk, and Unix shell and other scripting languages.

Python is copyrighted but the source code is available under the GNU General Public License (GPL).

3 Installation

Python has been installed in the computers in AT4019 on the Ubuntu partition. Login into Ubuntu using your Lakehead University username and password. Then open a terminal window.

To verify installation, type ”python” in the command line and the version that was installed will be shown.

However, there are other ways to install Python, which is the one suggested here for the tutorial if you will be running from your laptop. **Anaconda** is a free open source distribution of the Python and R programming languages for large-scale data processing, predictive analytics, and scientific computing, that aims to simplify package management and deployment. Because we will be using these tools for signal and imaging processing, the use of this kind of distribution is more adequate than individually installing packages as you go.

4 Installing Anaconda

You will find the Anaconda distribution on https://www.anaconda.com/download/

Because this is an open source distribution, changes may occur during time and when this tutorial and notebooks for the course were tested the versions used where

```
Python 2.7.13 |Anaconda 4.4.0 (64-bit)|
```

It is suggested to install the same versions to ensure compatibility, but most functions should be still possible to run in a more recent version.

5 Installing only Python

If you are interested in getting Python only for your Windows platform you can download it from https://www.python.org/downloads/windows/

If you prefer a Linux platform you can download it from https://www.python.org/downloads/ubuntu

The instructions for installation are given in those pages. But in general:

1. Download Gzipped source
2. Extract
3. Configure and Install

Python’s Official Website is at http://www.python.org/

6 Running Python

There are three different ways to start Python:

1. Interactive Interpreter: you can start Python from Unix, DOS, or any other system that provides you a command-line interpreter or shell window.
2. Executing scripts: you can start a series of Python commands by typing the name of a script with those commands
3. Using an integrated development environment
4. Using the interactive web application Jupyter that allows you to create and share documents that contain live code, equations, visualizations and explanatory text.

6.1 Command line

Start coding right in the interactive interpreter.

```
$python              # Unix/Linux
or
C:>python            # Windows/DOS
```

On your laptop you can alternatively open a command prompt from Anaconda (same).

6.2 Script

A Python script (here script.py) can be executed from the command line by invoking the interpreter on your application, as in the following:

```
$python  script.py          # Unix/Linux
```

6.3 Integrated Development Environment

You can run Python from a Graphical User Interface (GUI) environment as well, if you have a GUI application on your system that supports Python. Unix uses IDLE as a Unix IDE for Python. Windows has PythonWin as a Windows interface for Python and is an IDE with a GUI. Macintosh has either MacBinary or BinHex'd as an environment.

6.4 Jupyter

We will use in the course Jupyter as our tool for Python. This allows me to show explanatory text and code embedded units as a notebook for class and you can download the notebooks and modify them as required to practice and complete exercises.

7 First command in Python

We will first try some Python commands using the Interactive Mode Programming so you are familiar with the syntax.

Invoke the interpreter:

```
$ python
Python 3.3.2 (default, Dec 10 2013, 11:35:01)
[GCC 4.6.3] on Linux
Type "help", "copyright", "credits", or "license" for more information.
```

You will have a different prompt that shows you are now in the Python environment. Execute your first Python command:

```
>>> print ("Hello world using Python!")
```

You just ran your first Python command.

Now create a file and type the command in the file saved with extension .py. Invoke the script directly from the command line as:

```
$ python file.py
```

You should obtain the same result.

8 Basics of Python

8.1 Identifiers, comments, suites and indentation

A Python identifier is a name used to identify a variable, function, class, module or other object. An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9). Python does not allow punctuation characters and it is a case sensitive programming language.

There are some reserved words that cannot be used as identifiers:

```
and exec Not as finally or assert for pass break from print class global raise continue if return
def import try del in while elif is with else lambda yield except
```

Python doesn't use braces() to indicate blocks of code for class and function definitions or flow control. It uses line indentation, which is rigidly enforced. The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount.

```
if True:
    print ("True")
else:
  print ("False")
```

Or

```
if True:
    print ("Answer")
    print ("True")
else:
    print "(Answer")
    print ("False")
```

But if in Python the continuous lines are indented with different number of spaces you will have an error:

```
if True:
    print ("Answer")
    print ("True")
else:
    print "(Answer")
       print ("False")
```

A more complex example:

```
#!/usr/bin/python3

import sys

try:
  # open file stream
  file = open(file_name, "w")
except IOError:
  print ("There was an error writing to", file_name)
  sys.exit()
print ("Enter '", file_finish,)
print "' When finished"
while file_text != file_finish:
  file_text = raw_input("Enter text: ")
  if file_text == file_finish:
    # close the file
    file.close
    break
  file.write(file_text)
  file.write("\n")
file.close()
file_name = input("Enter filename: ")
if len(file_name) == 0:
  print ("Next time please enter something")
  sys.exit()
try:
  file = open(file_name, "r")
except IOError:
  print ("There was an error reading file")
  sys.exit()
file_text = file.read()
file.close()
print (file_text)
```

Statements in Python typically end with a new line. Python does, however, allow the use of the line continuation character (
) to denote that the line should continue.

```
total = item_one + \
        item_two + \
        item_three
```

Statements contained within the [], , or () brackets do not need to use the line continuation character.

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

Python accepts single ('), double (") and triple ("' or """) quotes to denote string literals, as long as the same type of quote starts and ends the string. The triple quotes are used to span the string across multiple lines. Some examples of correct use of quotes:

```
word = 'word'
sentence = "This is a sentence."
paragraph = """This is a paragraph. It is
made up of multiple lines and sentences."""
```

Comments in Python are made using a hash sign (#) that is not inside a string.

And blank lines are ignored by Python.

The semicolon (;) allows multiple statements on the single line given that neither statement starts a new code block.

```
import sys; x = 'foo'; sys.stdout.write(x + '\n')
```

A group of individual statements, which make a single code block are called suites. Compound or complex statements, such as if, while, def, and class require a header line and a suite.

A header line begins the statement (with the keyword) and terminate with a colon (:) and are followed by one or more lines which make up the suite.

```
if expression :
   suite
elif expression :
   suite
else :
   suite
```

8.2 Assigning values

Variables do not need explicit declaration to reserve memory space. The equal sign (=) is used to assign values to variables.

```
#!/usr/bin/python3

counter = 100          # An integer assignment
miles   = 1000.0       # A floating point
name    = "John"       # A string

print (counter)
print (miles)
print (name)
```

Result:

```
100
1000.0
John
```

Python allows you to assign a single value to several variables simultaneously.

```
a = b = c = 1
a, b, c = 1, 2, "john"
```

In the first all a, b and c will be equal to 1. In the second example a=1, then b=2 and c=john (string)

8.3 Data Types

Python has five standard data types:

1. Numbers: numeric values that can be integer (decimal, octal, hexadecimal), floats or complex
2. String: characters from 1 to many characters; they are indexed using [] and [:] similar to Matlab but the index starts in 0
3. Set: unordered collection of immutable values
4. List: different items separated by commas, similar to an array but the items belonging to a list can be of different data types

   ```
   list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
   tinylist = [123, 'john']

   print (list)          # Prints complete list
   print (list[0])       # Prints first element of the list
   print (list[1:3])     # Prints elements starting from 2nd till 3rd
   print (list[2:])      # Prints elements starting from 3rd element
   print (tinylist * 2)  # Prints list two times
   print (list + tinylist) # Prints concatenated lists
   ```

   Results in:

   ```
   ['abcd', 786, 2.23, 'john', 70.200000000000003]
   abcd
   [786, 2.23]
   [2.23, 'john', 70.200000000000003]
   [123, 'john', 123, 'john']
   ['abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john']
   ```

5. Tuple: lists that are read-only
6. Dictionary: work like associative arrays and consist of key-value pairs. A dictionary key can be almost any Python type, but are usually numbers or strings. Values can be any arbitrary Python object.

   ```
   dict = {}
   dict['one'] = "This is one"
   dict[2]     = "This is two"

   tinydict = {'name': 'john','code':6734, 'dept': 'sales'}

   print (dict['one'])       # Prints value for 'one' key
   print (dict[2])           # Prints value for 2 key
   print (tinydict)          # Prints complete dictionary
   print (tinydict.keys())   # Prints all the keys
   print (tinydict.values()) # Prints all the values
   ```

   Results in:

   ```
   This is one
   This is two
   {'dept': 'sales', 'code': 6734, 'name': 'john'}
   ['dept', 'code', 'name']
   ['sales', 6734, 'john']
   ```

When to use list vs. tuple vs. dictionary vs. set?

1. A *list* is like an array, it can be used to store homogeneous as well as heterogeneous data types and individual elements of a list can be accessed using indexing and can be manipulated. Only use if the elements in the list need searching.
2. A *tuple* has similar uses to lists, but there data cannot be changed once created. So they are useful for immutable lists.
3. A *set* stores unordered values so they will have no index and cannot have any duplicate data. Once created elements can be added. Sets are significantly faster when searching an element than lists but cannot be indexed.
4. A *dictionary* will be used when a pair of key vs. value is required. All indexing is done by using those keys, not numerical indexes.

8.4 Data Conversion

Conversions between the built-in types are performed by functions. There are several built-in functions to perform conversion from one data type to another. These functions return a new object representing the converted value.

- int(x [,base]): Converts x to an integer. base specifies the base if x is a string.
- float(x): Converts x to a floating-point number.
- complex(real [,imag]): Creates a complex number.
- str(x): Converts object x to a string representation.
- repr(x): Converts object x to an expression string.
- eval(str): Evaluates a string and returns an object.
- tuple(s): Converts s to a tuple.
- list(s): Converts s to a list.

- dict(d): Creates a dictionary. d must be a sequence of (key,value) tuples.
- frozenset(s): Converts s to a frozen set.
- chr(x): Converts an integer to a character.
- unichr(x): Converts an integer to a Unicode character.
- ord(x): Converts a single character to its integer value.
- hex(x): Converts an integer to a hexadecimal string.
- oct(x): Converts an integer to an octal string.

8.5 Operators

Operators are the constructs which can manipulate the value of operands.

1. Arithmetic Operators: +, -, *, /, % (modulus), ** (exponent), // (floor division: quotient)
2. Comparison Operators: == (equality), != (not equality), ¿, ¡, ¿=, ¡=
3. Assignment Operators: =, += (c += a is the same as c = c+a), -=, *=, /=, **=, //=,
4. Logical Operators: and, or, not
5. Bitwise Operators: &, |, ˆ (XOR), ˜(Ones complement or bit-flip)
6. Membership Operators (check for membership in a list, tuple, etc.): in, not in
7. Identity Operators (check if the operator point to the same object): is, is not

The precedence from highest to lowest is:
Exponentiation, Multiplication/Division/Modulus/Floor Division, Addition/Subtraction, Right/Left Bit-shift, And/Or Bitwise, Comparison operators, Equality operators, Assignment operators, Identity operators, Membership operators and Logical operators

8.6 Decisions and Loops

Decision making and loops are used to execute commands depending on the outcome of logical tests. Remember that indentation allows for nesting (group of commands) to happen, so all statements should be indented at the same level or you will get an error.

The basic decision making is if/else:

```
if expression1:
   statement(s)
elif expression2:
   statement(s)
elif expression3:
   statement(s)
else:
   statement(s)
```

Where elif is used as equivalent to ”else, then if” chain.

The while loop:

```
count = 0
while (count < 9):
   print ('The count is:', count)
   count = count + 1
else:
   print (count, " is not less than 9")
```

A while with a single statement is possible and will not need indentation (this loop will only increase count):

```
while (count < 9): count=count+1
```

The for loop:

```
for letter in 'Python':     # traversal of a string sequence
   print ('Current Letter :', letter)
```

By default a for will loop through the elements of a list, which in this case is the string 'Python' but it can be also a list of strings:

```
fruits = ['banana', 'apple',  'mango']
for fruit in fruits:        # traversal of List sequence
   print ('Current fruit :', fruit)
```

Or an index in the list if we use range:

```
fruits = ['banana', 'apple',  'mango']
for index in range(len(fruits)):
   print ('Current fruit :', fruits[index])
```

The built-in function range creates lists containing arithmetic progressions based on the number given, which in this case is the length of the fruits list.

And we can also use an else in a for loop as for while and it will execute when the loop ends normally.

```
numbers=[11,33,55,39,55,75,37,21,23,41,13]
for num in numbers:
    if num%2==0:
        print ('the list contains an even number')
        break
else:
    print ('the list does not contain even number')
```

In the code the else will be executed since there is no even number and therefore we never executed the 'break'.

8.7 Built-in mathematical functions

Python includes the following functions that perform mathematical calculations.

- abs(x): absolute value of x: the (positive) distance between x and zero
- ceil(x): ceiling of x: the smallest integer not less than x
- exp(x): e^x
- fabs(x): absolute value of x (no sign)
- floor(x): floor of x
- log(x): natural logarithm of x
- log10(x): base-10 logarithm of x
- max(x1, x2,...): largest of its arguments
- min(x1, x2,...): smallest of its arguments
- modf(x): fractional and integer parts of x in a two-item tuple
- round(x [,n]): x rounded to n digits from the decimal point
- sqrt(x): square root of x
- randrange([start,] stop [,step]): randomly selected element from range(start, stop, step)
- random(): random float r between 0 and 1

- seed([x]): integer starting value used in generating random numbers (has to be called before random module functions)
- shuffle(lst): randomizes the items of a list in place
- uniform(x, y): random float between x and y
- acos(x): arc cosine of x in radians
- asin(x): arc sine of x in radians
- atan(x): arc tangent of x in radians
- atan2(y, x): atan(y/x) in radians
- cos(x): cosine of x radians
- hypot(x, y): Euclidean norm or $\sqrt{x^2 + y^2}$
- sin(x): sine of x radians
- tan(x): tangent of x radians
- degrees(x): angle x from radians to degrees
- radians(x): angle x from degrees to radians
- pi: mathematical constant pi
- e: mathematical constant e

8.8 Strings and string functions

As discussed before, string handling are one of the strengths or Python. Python does not support a character type but it uses strings of length one.

To access substrings, use the square brackets for slicing along with the index or indices to obtain your substring.

```
var1 = 'Hello World!'
var2 = "Python Programming"

print ("var1[0]: ", var1[0])
print ("var2[1:5]: ", var2[1:5])
```

Will give:

```
var1[0]:  H
var2[1:5]:  ytho
```

Existing strings can be updated by reassigning a variable and you can perform operations on strings using + for concatenation.

```
var1 = 'Hello World!'

print ("Updated String :- ", var1[:6] + 'Python')
```

Similar to what is used in Matlab or C, escape or non-printable characters can be represented with backslash notation to manipulate the output of a string:

- \a Bell or alert
- \b Backspace
- \cx Control-x
- \e Escape

- \f Form-feed
- \n Newline
- \r Carriage return
- \s Space
- \t Tab
- \v Vertical tab

The operators act:

- + Concatenation - Adds values on either side of the operator
- * Repetition - Creates new strings, concatenating multiple copies of the same string a*2 will give -HelloHello

  Slice - Gives the character from the given index between brackets

: Range Slice - Gives the characters from the given range (index starts at 0)

- in Membership - Returns true if a character exists in the given string
- not in Membership - Returns true if a character does not exist in the given string
- r/R Raw String - Suppresses actual meaning of Escape characters
- % Format - Performs String formatting just as in C or Matlab and the printf()

```
print ("My name is %s and weight is %d kg!" % ('Zara', 21))
```

Produces the following result

```
My name is Zara and weight is 21 kg!
```

And the symbols which can be used along with % are: %c (character), %s (string), %i (integer), %d (signed integer), %u (unsigned decimal integer), %o (octal integer), %x or %X (hexadecimal integer), %e or %E (exponential notation), %f (floating real number), %g (the shorter string between float or exponential notation)

Other supported symbols in formatting: * (width or precision), - (left justification), + (display the sign), ¡sp¿ (leave a blank space before a positive number), # (add the octal or hexadecimal with a 0), 0 (pad from left with zeros), %% (single literal %), m.n. (m is the minimum total width and n is the number of digits to display after the decimal point)

Python accepts ', " or "" as quotes as discussed before and the triple quotes allow for multiple line text automatically to display in that format.

Python has some built-in methods to manipulate strings:

- capitalize(): capitalizes first letter of string
- center(width, fillchar): returns a string padded with fillchar with the original string centered to a total of width columns
- count(str, beg= 0,end=len(string)): counts how many times str occurs in string or in a substring of string if starting index beg and ending index end are given
- decode(encoding='UTF-8',errors='strict'): decodes the string using the codec registered for encoding
- encode(encoding='UTF-8',errors='strict'): encodes the string
- endswith(suffix, beg=0, end=len(string)): determines if string or a substring of string (if starting index beg and ending index end are given) ends with suffix; returns true if so and false otherwise
- expandtabs(tabsize=8): expands tabs in string to multiple spaces; defaults to 8 spaces per tab if tabsize not provided

- find(str, beg=0 end=len(string)): determines if str occurs in string or in a substring of string if starting index beg and ending index end are given returns index if found and -1 otherwise
- index(str, beg=0, end=len(string)): same as find(), but raises an exception if str not found
- isalnum(): returns true if string has at least 1 character and all characters are alphanumeric and false otherwise
- isalpha(): returns true if string has at least 1 character and all characters are alphabetic and false otherwise
- isdigit(): returns true if string contains only digits and false otherwise
- islower(): returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise
- isnumeric(): returns true if a unicode string contains only numeric characters and false otherwise
- isspace(): returns true if string contains only whitespace characters and false otherwise
- istitle(): returns true if string is properly ”titlecased” and false otherwise
- isupper(): returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise
- join(seq): merges or concatenates the string representations of elements in sequence seq into a string, with separator string
- len(string): returns the length of the string
- ljust(width[, fillchar]): returns a space-padded string with the original string left-justified to a total of width columns
- lower(): converts all uppercase letters in string to lowercase
- lstrip(): removes all leading whitespace in string
- maketrans(): returns a translation table to be used in translate function
- max(str): returns the max alphabetical character from the string str
- min(str): returns the min alphabetical character from the string str
- replace(old, new [, max]): replaces all occurrences of old in string with new or at most max occurrences if max given
- rfind(str, beg=0,end=len(string)): same as find(), but search backwards in string
- rindex(str, beg=0, end=len(string)): same as index(), but search backwards in string
- rjust(width,[, fillchar]): returns a space-padded string with the original string right-justified to a total of width columns
- rstrip(): removes all trailing whitespace of string
- split(str=””, num=string.count(str)): splits string according to delimiter str (space if not provided) and returns list of substrings; split into at most num substrings if given
- splitlines(num=string.count('\n')): splits string at all (or num) NEWLINEs and returns a list of each line with NEWLINEs removed
- startswith(str, beg=0,end=len(string)): determines if string or a substring of string (if starting index beg and ending index end are given) starts with substring str; returns true if so and false otherwise
- strip([chars]): performs both lstrip() and rstrip() on string
- swapcase(): inverts case for all letters in string

- title(): returns ”titlecased” version of string, that is, all words begin with uppercase and the rest are lowercase
- translate(table, deletechars=””): translates string according to translation table str(256 chars), removing those in the del string
- upper(): converts lowercase letters in string to uppercase
- zfill (width): returns original string leftpadded with zeros to a total of width characters; intended for numbers, zfill() retains any sign given (less one zero)

8.9 Lists and list functions

Python’s basic data structure are sequences: list, tuples, sets and dictionaries. The basic operations on a sequence include indexing, slicing, adding, multiplying, and checking for membership. In addition, Python has built-in functions for finding the length of a sequence and for finding its largest and smallest elements.

We will cover lists since most of the usage of all sequences is similar.

The list is a most versatile datatype available in Python which can be written as a list of comma-separated values (items) between square brackets. Important thing about a list: items in a list need not be of the same type!

Creating a list:

```
list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5 ];
list3 = ["a", "b", "c", "d"];
```

Accessing values in lists:

```
list1 = ['physics', 'chemistry', 1997, 2000]
list2 = [1, 2, 3, 4, 5, 6, 7 ]

print ("list1[0]: ", list1[0])
print ("list2[1:5]: ", list2[1:5])
```

will give:

```
list1[0]:  physics
list2[1:5]:  [2, 3, 4, 5]
```

Updating lists can be done by reassigning, applying operators (+, -, *) or using append() or other functions. To remove a list element, you can use either the del statement if you know exactly which element(s) you are deleting or the remove() method if you do not know.

Because lists are sequences, indexing and slicing work the same way for lists as they do for strings. Some examples on indexing and built-in functions and methods:

- L[2] on ’Python’ will give t
- L[-2] counts from the right
- L[1:] on [’Java’, ’Python’] will slice from the 2nd element onward
- cmp(list1, list2): compares elements of both lists
- len(list): gives the total length of the list
- max(list): returns item from the list with max value
- min(list): returns item from the list with min value
- list(seq): converts a tuple into list
- list.append(obj): this is not a function but a method and it appends object obj to list
- list.count(obj): this method returns count of how many times obj occurs in list
- list.extend(seq): this method appends the contents of seq to list

- list.index(obj): this method returns the lowest index in list that obj appears
- list.insert(index, obj): this method inserts object obj into list at offset index
- list.pop(obj=list[-1]): this method removes and returns last object or obj from list
- list.remove(obj): this method removes object obj from list
- list.reverse(): this method reverses objects of list in place
- list.sort([func]): this method sorts objects of list, use compare func if given

8.10 Other Built-in functions

Some other (not exhaustive) list of built-in functions:

- all(iterable): returns True if all elements of the iterable are true (or if the iterable is empty)
- any(iterable): returns True if any element of the iterable is true
- bin(x): convert an integer number to a binary string
- callable(object): returnr True if the object argument appears callable
- complex([real[, imag]]): returns a complex number with the value real + imag*1j or converts a string or number to a complex number
- eval(expression[, globals[, locals]]): evaluates the expression
- filter(function, iterable): constructs a list from those elements of iterable for which function returns true
- float([x]): returns a floating point number constructed from a number or string x
- format(value[, format_spec]): converts a value to a formatted representation
- frozenset([iterable]): returns a new frozenset object made from iterable
- help([object]): invokes the built-in help system for the object
- hex(x): converts an integer number (of any size) to a lowercase hexadecimal string
- input([prompt]): obtains input from the keyboard
- int(x=0): returns an integer object constructed from a number or string x
- iter(o[, sentinel]): returns an iterator object
- len(s): returns the length (the number of items) of an object
- list([iterable]): returns a list whose items are the same and in the same order as iterable's items
- next(iterator[, default]): retrieves the next item from the iterator
- oct(x): converts an integer number (of any size) to an octal string
- open(name[, mode[, buffering]]): opens a file, returning an object of the file type
- print(objects, sep=' ', end='\n ', file \= sys.stdout): prints objects to the stream file or screen
- range(start, stop[, step]): function to create lists containing arithmetic progressions. It is most often used in for loops. The arguments must be plain integers. If the step argument is omitted, it defaults to 1. If the start argument is omitted, it defaults to 0. The full form returns a list of plain integers [start, start + step, start + 2 * step, ...]. If step is positive, the last element is the largest start + i * step less than stop; if step is negative, the last element is the smallest start + i * step greater than stop. Make sure that step is not zero.

```
>>>range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1, 11)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> range(0, 30, 5)
[0, 5, 10, 15, 20, 25]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(0, -10, -1)
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> range(0)
[]
>>> range(1, 0)
[]
```

- sorted(iterable[, cmp[, key[, reverse]]]): returns a new sorted list from the items in iterable
- sum(iterable[, start]): sums the items of an iterable from left to right and returns the total
- tuple([iterable]): returns a tuple whose items are the same and in the same order as iterable's items

8.11 Defining and using functions

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

Besides the built-in functions you can create your own functions. You can define functions to provide the required functionality. Here are simple rules to define a function in Python:

- Function blocks begin with the keyword **def** followed by the function name and parentheses (( )).
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be optional if it is between ""
- The code block within every function starts with a colon (:) and is indented
- The statement **return [expression]** exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

```
def functionname( parameters ):
   "function_docstring"
   function_suite
   return [expression]
```

- By default, parameters have a positional behaviour and you need to inform them in the same order that they were defined.
- Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code. You can execute it by calling it from another function or directly from the Python prompt.

```
#!/usr/bin/python3

# Function definition is here
def changeme( mylist ):
   "This changes a passed list into this function"
   print ("Values inside the function before change: ", mylist)
   mylist[2]=50
   print ("Values inside the function after change: ", mylist)
   return
```

```
# Now you can call changeme function
mylist = [10,20,30]
changeme( mylist )
print ("Values outside the function: ", mylist)
```

- You can call a function by using required, keyword, default and variable-length arguments: required arguments are the arguments passed to a function in correct positional order; default argument is an argument that assumes a default value if a value is not provided in the function call for that argument (the def would have an assigned value in that case); if a function may or may not have all arguments you can use variable arguments (those are preceded by * in def)

8.12 Local and global variables

Variables that are defined inside a function body have a local scope, and those defined outside have a global scope. This means that local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body by all functions. When you call a function, the variables declared inside it are brought into scope.

8.13 Modules

If you quit from the Python interpreter and enter it again, the definitions you have made (functions and variables) are lost. Therefore, if you want to write a somewhat longer program, you need to use a text editor to prepare the input for the interpreter and running it with that file as input instead. This is known as creating a script. As your program gets longer, you may want to split it into several files for easier maintenance. You may also want to use a handy function that you have written in several programs without copying its definition into each program.

To support this, Python created what is called modules: definitions from a module can be imported into other modules or into the main module. A module is a file containing Python definitions and statements. The file name is the module name with the suffix .py appended.

Simply, a module is a file consisting of Python code. A module can define functions, classes and variables. A module can also include runnable code. It can be seen as similar to a *.m file in Matlab that includes definition of functions.

Many modules exist and are available for Python users that can be reused by the community. We will be using some of those in the course.

In order for you to use any Python source file as a module you need to execute an import statement:

```
import module1[, module2[,... moduleN]
```

When the interpreter encounters an import statement, it imports the module if the module is present in the search path. A module is loaded only once, regardless of the number of times it is imported. This prevents the module execution from happening over and over again if multiple imports occur.

Once you import a module, the module's name (as a string) is available as the value of the global variable __name__. For instance, if we have a file:

```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b

def fib2(n):   # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

That is called fibo.py you can import it using

```
import fibo
```

And that allows you to execute

```
fibo.fib(1000)
```

And if you desire to do so, you can assign that function to another local name and keep using the local name

```
fiblocal=fido.fib
fiblocal(1400)
```

You can alternatively only import from a module the names that you require. The from...import is used for this:

```
from modname import name1[, name2[, ... nameN]]
```

This statement does not import the entire modname module into the current namespace, only the item itname from the module modname into the global symbol table of the importing module. So:

```
from fibo import fib
```

Will import only fib from fibo into the current namespace and to use that function you will now:

```
fib(1000)
```

Which means that you do not need to call the module name and you will not need to assign to a local variable, it is directly done. If you use an from import with an * all names are imported, but it can cause difficulty in understanding your code.

When you import a module, the Python interpreter searches for the module in the following sequences:

1. The current directory
2. Each directory in the shell variable PYTHONPATH
3. The default path

So beware duplicate modules that can cause differences in execution.

A Python statement can access variables in a local namespace and in the global namespace. If a local and a global variable have the same name, the local variable shadows the global variable.

Python makes educated guesses on whether variables are local or global. It assumes that any variable assigned a value in a function is local.

In order to assign a value to a global variable within a function, you must first use the global statement.

You can know which variables are local or global by calling locals() and globals() from that function.

8.14 Files

The basic I/O functions available in Python are:

- Printing to screen
- Capturing from keyboard
- Using I/O files

The simplest way to produce output is using the print statement where you can pass zero or more expressions separated by commas. This is obtained by the **print** function that converts the expressions you pass into a string and writes the result to standard output.

Capturing from the keyboard is the simplest way to obtain an input for a program. The **input([prompt])** function accomplishes this.

Python provides basic functions and methods necessary to manipulate files by default. You can do most of the file manipulation using a built-in file object. The basic steps to access a file are:

1. Open the file
2. Read or write from/to the file
3. Close the file

Before you can read or write a file, you have to open it using Python's built-in open() function. This function creates a file object, which would be utilized to call other support methods associated with it.

```
file object = open(file_name [, access_mode][, buffering])
```

Open parameter details:

1. file_name: The file_name argument is a string value that contains the name of the file that you want to access.
2. access_mode: The access_mode determines the mode in which the file has to be opened, i.e., read, write, append, etc.
3. buffering: If the buffering value is set to 0, no buffering takes place. If the buffering value is 1, line buffering is performed while accessing a file. If you specify the buffering value as an integer greater than 1, then buffering action is performed with the indicated buffer size. If negative, the buffer size is the system default(default behaviour).

The different modes of opening a file:

- r: File for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
- rb: File for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.
- r+: File for both reading and writing. The file pointer placed at the beginning of the file.
- rb+: File for both reading and writing in binary format. The file pointer placed at the beginning of the file.
- w: File for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
- wb: File for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
- w+: File for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
- wb+: File for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
- a: File for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
- ab: File for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
- a+: File for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
- ab+: File for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

Once a file is opened and you have one file object, you can get various information related to that file. You access that information using methods of that object:

- 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

Example

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

This opens the file foo.txt in write mode for binary data and prints the name of the file, says its name and that it is not closed as well as the mode in which it was opened. It finally closes the file and therefore flushes any unwritten information and closes the file object, after which no more writing can be done. Python automatically closes a file when the reference object of a file is reassigned to another file but it is a good practice to use the close() method to close a file.

To write into a file we use the **write** method. The write method writes any string (note that a string is not just text string, it can be any kind of string or list from Python) to an open file. It is important to then be aware that Python strings can have binary data and not just text. The write method does not add a newline character to the end of the string unless specified.

```
fileObject.write(string);
```

Example:

```
# Open a file
fo = open("foo.txt", "w")
fo.write( "Python is a great language.\nYeah its great!!\n")

# Close opend file
fo.close()
```

The above creates foo.txt file and writes given content in that file and finally it would close that file. Because the string that was written was text, the file ends being a text file, but it could have been written in binary if the string had been binary. However, if the file is open as binary (wb) it will be written in binary only.

To read from a file we use the **read** method.

```
fileObject.read([count]);
```

Here, we pass as parameter the number of bytes to be read from the opened file. This method starts reading from the beginning of the file and if count is missing, then it tries to read as much as possible, maybe until the end of file.

You can use the **tell()** method to obtain the current position within the file; in other words, the next read or write will occur at that many bytes from the beginning of the file. The **seek(offset[, from])** method changes the current file position.

Some other methods allow manipulation of files at the OS level: rename(), remove(), mkdir(), chdir(), getcwd(), rmdir().

9 Jupyter Notebooks

The Jupyter Notebook is a web application that allows you to create and share documents that contain live code, equations, visualizations and explanatory text.

It supports over 40 programming languages, including those popular in data science such as Python, R, Julia and Scala.

We will use this tool to share code in Python and to execute interactively that code.

The nbviewer in Jupyter renders the notebooks but it does not hosts the notebooks. Those are hosted anywhere in the web and by giving an URL nbviewer can execute them.

Our first notebook will have the most basic first example for a code in Python. You can also directly browse collections of notebooks in public GitHub repositories, for example the IPython examples. I have hosted notebooks for our course in my GitHub repository and made them public, so those URLs will be available to you.

To see a notebook you do not require anything but the URL. There are two ways to render a Jupyter Notebook:

1. Directly from a public GitHub repository if they are available from one (the ones for the course are)
2. Using the nbviewer webtool

From the GitHub repository you can find it in the URL for my public GitHub repository:

`https://github.com/lcurielramirez/ENGI-5631-FA2017/blob/master/01SimpleExample.ipynb`

This directly renders the notebook.

You can also go to

`http://nbviewer.jupyter.org/`

And type the same URL.

However you can only render the notebook that is shared, but you cannot really execute the code or make any changes, which is how you would interact with the code.

To run and/or modify a notebook, you will require:

1. To have Jupyter notebook modules (code) in Python which are found in the Anaconda distribution of Python
2. To download the notebook locally so you can modify it in you own repository (local)

If you download and install the Anaconda distribution of Python, all modules required to execute Jupyter will be there. You can find more information at:

`http://jupyter.org/install.html`

We will first download the Anaconda distribution of Python which is an open data science platform with many available Python modules. It is a high performance distribution of Python and R and includes over 100 Python, R and Scala packages for data science. It is a powerful tool for engineering and scientific calculations and it is what we will be using.

We will install Anaconda on your terminals. Go to https://www.continuum.io/downloads and follow all the instructions to download and install for Linux (unless you decide to work on your laptop, for which you should chose the right OS).

Once you download you will be proposed to receive a cheat sheet for Anaconda, you are welcome to read through later.

Before typing the bash command make sure you are in your Downloads directory. To install in Linux type

```
bash Anaconda3-4.4.0-Linux-ppc64le.sh
```

Or the exact name of the bash file that downloads from the continuum site. This will install Anaconda for Python 3.6 version in Linux.

You have now Anaconda, you can install similarly on a PC or Mac if you want to. Unfortunately, Anaconda has installed in the exact terminal you are working on, so if you change terminals in the future, this needs to be repeated. I strongly suggest that you remain on the same terminal for the course and that you install Anaconda on your personal computer to work on your exercises after class.

In your terminal you can verify you have conda by typing

```
conda -info
```

If the command is not recognized, you did not install properly.

NOTE: you will have to close and reopen your command window after install for the command to be recognized (path is not up yet).

You now have modules to edit/run Jupyter notebooks but you don't have any local notebook to work on yet.

To download the notebook the simplest way is to use nbviewer, which when rendering the notebook will have a "down" arrow at the top right of the webpage (to the right of a "cat" from GitHub). A file called SimpleExample.ipynb (i.e. iPython Notebook) will download into your computer. That is the source for this notebook and you will be editing and running it.

You can alternatively go directly to the URL of my GitHub, request the "raw" code and save as a file that should have the extension ipynb.

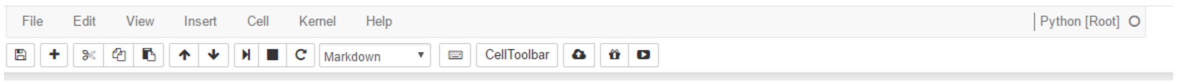
Move the file to the folder where you desire to edit locally this notebook (I suggest you do not keep it in Downloads where it can be crashed by the "old" version accidentally and you will loose your edits). We will now execute Jupyter.

First open a prompt from Anaconda and then move to the folder where the notebook is located and run:

```
jupyter notebook
```

That will automatically start a webpage with jupyter that looks almost as the one from nbviewer, but you can now modify the text.

Now that you can modify the notebook, new tools are open to you in Jupyter. Let's explore them:



Your notebook is made of "Cells" that can be text (Markdown) or code. The toolbar will allow you editing, adding, cutting and moving through your cells.

You can run your pieces of code by typing Shift+Enter in a code cell. Pleas do so for your only code cell.

To see the difference, edit that string to anything and re-run, you should see a different output now. You can also change the Markdown and so on. You are now editing a Notebook!

For many examples during lectures we will be reading through Notebooks that I made public for the course. You should download those notebooks before class, move them to your local repositories and be ready to modify the code during the lecture so you can test features in it yourself. Questions about that notebook will be in the assignments and you will have required to run the notebook to answer them. Class is the best time to troubleshoot, and interactive editing is the best way to make sure you understand the concepts.

10 References

1. http://mathesaurus.sourceforge.net/matlab-python-xref.pdf
2. https://docs.python.org/3/tutorial/index.html
3. https://nbviewer.jupyter.org/