Introduction to Python

December 5, 2016

Introduction to Python - available from https://gitlab.erc.monash.edu.au/andrease/Python4Maths.git

The original version was written by Rajath Kumar and is available at https://github.com/rajathkumarmp/Python-Lectures. The notes have been updated for Python 3 and amended for use in Monash University mathematics courses by Andreas Ernst

1 Python-Lectures

1.1 Introduction

Python is a modern, robust, high level programming language. It is very easy to pick up even if you are completely new to programming.

Python, similar to other languages like matlab or R, is interpreted hence runs slowly compared to C++, Fortran or Java. However writing programs in Python is very quick. Python has a very large collection of libraries for everything from scientific computing to web services. It caters for object oriented and functional programming with module system that allows large and complex applications to be developed in Python.

These lectures are using jupyter notebooks which mix Python code with documentation. The python notebooks can be run on a webserver or stand-alone on a computer.

To give an indication of what Python code looks like, here is a simple bit of code that defines a set $N = \{1, 3, 4, 5, 7\}$ and calculates the sum of the squared elements of this set:

$$\sum_{i \in N} i^2 = 100$$

```
In [2]: N={1,3,4,5,7,8}
        print('The sum of ∑_i∈N i*i =',sum( i**2 for i in N ) )
```

```
The sum of ∑_i∈N i*i = 164
```

1.2 Contents

This course is broken up into a number of notebooks (chapters).

- 00 This introduction with additional information below on how to get started in running python
- 01 Basic data types and operations (numbers, strings)
- 02 String manipulation
- 03 Data structures: Lists and Tuples
- 04 Data structures (continued): dictionaries
- 05 Control statements: if, for, while, try statements
- 06 Functions
- 07 Classes and basic object oriented programming
- 08 Scipy: libraries for arrays (matrices) and plotting
- 09 Mixed Integer Linear Programming using the mymip library.

This is a tutorial style introduction to Python. For a quick reminder / summary of Python syntax the following Quick Reference Card may be useful. A longer and more detailed tutorial style introduction to python is available from the python site at: https://docs.python.org/3/tutorial/

1.3 Installation

1.3.1 Loging into the web server

The easiest way to run this and other notebooks for staff and students at Monash University is to log into the Jupyter server at `https://sci-web17-v01.ocio.monash.edu.au/hub`. The steps for running notebooks are: * Log in using your monash email address. The first time you log in an empty account will automatically be set up for you. * Press the start button (if prompted by the system) * Use the menu of the jupyter system to upload a .ipynb python notebook file or to start a new notebook.

1.3.2 Installing

Python runs on windows, linux, mac and other environments. There are many python distributions available. However the recommended way to install python under Microsoft Windows or Linux is to use the Anaconda distribution available at `https://www.continuum.io/downloads`. Make sure to get the Python 3.5 version, not 2.7. This distribution comes with the SciPy collection of scientific python tools as well as the iron python notebook. For developing python code without notebooks consider using spyder (also included with Anaconda)

To open a notebook with anaconda installed, from the terminal run:

```
ipython notebook
```

Note that for the Monash University optimisation course additional modules relating to the commercial optimisation library CPLEX and possibly Gurobi will be used. These libraries are not available as part of any standard distribution but are available under academic licence and are included on the Monash server.

1.4 How to learn from this resource?

Download all the notebooks from Moodle or https://gitlab.erc.monash.edu.au/andrease/Python4Maths.git

Upload them to the monash server and lauch them or launch ipython notebook from the folder which contains the notebooks. Open each one of them

Cell > All Output > Clear

This will clear all the outputs and now you can understand each statement and learn interactively.

1.5 License

All of these python notebooks are available at [https://gitlab.erc.monash.edu.au/andrease/Python4Maths.git]

2 Getting started

Python can be used like a calculator. Simply type in expressions to get them evaluated.

2.1 Basic syntax for statements

The basic rules for writing simple statments and expressions in Python are: * No spaces or tab characters allowed at the start of a statement: Indentation plays a special role in Python (see the section on control statements). For now simply ensure that all statements start at the beginning of the line. * The '#' character indicates that the rest of the line is a comment * Statements finish at the end of the line: * Except when there is an open bracket or paranthesis:

```
1+2
+3  #illegal continuation of the sum
(1+2
             + 3) # perfectly OK even with spaces
```

- A single backslash at the end of the line can also be used to indicate that a statement is still incomplete

  ```
  1 + \
     2 + 3 # this is also OK
  ```

  The jupyter notebook system for writting Python intersperses text (like this) with Python statements. Try typing something into the cell (box) below and press the 'run cell' button above (triangle+line symbol) to execute it.

```
In [1]: 1+2+3
```

```
Out[1]: 6
```

Python has extensive help built in. You can execute **help()** for an overview or **help(x)** for any library, object or type **x** to get more information. For example:

```
In [ ]: help()
```

```
Welcome to Python 3.5's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/3.5/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics".  Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".

help> keywords

Here is a list of the Python keywords.  Enter any keyword to get more help.

False               def                 if                  raise
None                del                 import              return
True                elif                in                  try
and                 else                is                  while
as                  except              lambda              with
assert              finally             nonlocal            yield
break               for                 not
class               from                or
continue            global              pass

help> False
Help on bool object:

class bool(int)
 |  bool(x) -> bool
 |
 |  Returns True when the argument x is true, False otherwise.
 |  The builtins True and False are the only two instances of the class bool.
 |  The class bool is a subclass of the class int, and cannot be subclassed.
 |
```

```
 |  Method resolution order:
 |      bool
 |      int
 |      object
 |
 |  Methods defined here:
 |
 |  __and__(self, value, /)
 |      Return self&value.
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |
 |  __or__(self, value, /)
 |      Return self|value.
 |
 |  __rand__(self, value, /)
 |      Return value&self.
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  __ror__(self, value, /)
 |      Return value|self.
 |
 |  __rxor__(self, value, /)
 |      Return value^self.
 |
 |  __str__(self, /)
 |      Return str(self).
 |
 |  __xor__(self, value, /)
 |      Return self^value.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from int:
 |
 |  __abs__(self, /)
 |      abs(self)
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __bool__(self, /)
 |      self != 0
 |
 |  __ceil__(...)
 |      Ceiling of an Integral returns itself.
 |
 |  __divmod__(self, value, /)
 |      Return divmod(self, value).
 |
 |  __eq__(self, value, /)
 |      Return self==value.
```

```
|
|  __float__(self, /)
|      float(self)
|
|  __floor__(...)
|      Flooring an Integral returns itself.
|
|  __floordiv__(self, value, /)
|      Return self//value.
|
|  __format__(...)
|      default object formatter
|
|  __ge__(self, value, /)
|      Return self>=value.
|
|  __getattribute__(self, name, /)
|      Return getattr(self, name).
|
|  __getnewargs__(...)
|
|  __gt__(self, value, /)
|      Return self>value.
|
|  __hash__(self, /)
|      Return hash(self).
|
|  __index__(self, /)
|      Return self converted to an integer, if self is suitable for use as an index into a list.
|
|  __int__(self, /)
|      int(self)
|
|  __invert__(self, /)
|      ~self
|
|  __le__(self, value, /)
|      Return self<=value.
|
|  __lshift__(self, value, /)
|      Return self<<value.
|
|  __lt__(self, value, /)
|      Return self<value.
|
|  __mod__(self, value, /)
|      Return self%value.
|
|  __mul__(self, value, /)
|      Return self*value.
|
|  __ne__(self, value, /)
|      Return self!=value.
|
|  __neg__(self, /)
|      -self
|
|  __pos__(self, /)
|      +self
|
|  __pow__(self, value, mod=None, /)
|      Return pow(self, value, mod).
|
|  __radd__(self, value, /)
|      Return value+self.
|
|  __rdivmod__(self, value, /)
|      Return divmod(value, self).
|
|  __rfloordiv__(self, value, /)
|      Return value//self.
|
|  __rlshift__(self, value, /)
|      Return value<<self.
|
|  __rmod__(self, value, /)
|      Return value%self.
|
|  __rmul__(self, value, /)
|      Return value*self.
|
|  __round__(...)
|      Rounding an Integral returns itself.
|      Rounding with an ndigits argument also returns an integer.
|
|  __rpow__(self, value, mod=None, /)
|      Return pow(value, self, mod).
|
|  __rrshift__(self, value, /)
|      Return value>>self.
|
|  __rshift__(self, value, /)
|      Return self>>value.
|
|  __rsub__(self, value, /)
|      Return value-self.
|
|  __rtruediv__(self, value, /)
|      Return value/self.
|
|  __sizeof__(...)
|      Returns size in memory, in bytes
|
|  __sub__(self, value, /)
|      Return self-value.
|
|  __truediv__(self, value, /)
|      Return self/value.
```

```
|
|  __trunc__(...)
|      Truncating an Integral returns itself.
|
|  bit_length(...)
|      int.bit_length() -> int
|
|      Number of bits necessary to represent self in binary.
|      >>> bin(37)
|      '0b100101'
|      >>> (37).bit_length()
|      6
|
|  conjugate(...)
|      Returns self, the complex conjugate of any int.
|
|  from_bytes(...) from builtins.type
|      int.from_bytes(bytes, byteorder, *, signed=False) -> int
|
|      Return the integer represented by the given array of bytes.
|
|      The bytes argument must be a bytes-like object (e.g. bytes or bytearray).
|
|      The byteorder argument determines the byte order used to represent the
|      integer.  If byteorder is 'big', the most significant byte is at the
|      beginning of the byte array.  If byteorder is 'little', the most
|      significant byte is at the end of the byte array.  To request the native
|      byte order of the host system, use `sys.byteorder' as the byte order value.
|
|      The signed keyword-only argument indicates whether two's complement is
|      used to represent the integer.
|
|  to_bytes(...)
|      int.to_bytes(length, byteorder, *, signed=False) -> bytes
|
|      Return an array of bytes representing an integer.
|
|      The integer is represented using length bytes.  An OverflowError is
|      raised if the integer is not representable with the given number of
|      bytes.
|
|      The byteorder argument determines the byte order used to represent the
|      integer.  If byteorder is 'big', the most significant byte is at the
|      beginning of the byte array.  If byteorder is 'little', the most
|      significant byte is at the end of the byte array.  To request the native
|      byte order of the host system, use `sys.byteorder' as the byte order value.
|
|      The signed keyword-only argument determines whether two's complement is
|      used to represent the integer.  If signed is False and a negative integer
|      is given, an OverflowError is raised.
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from int:
|
|  denominator
|      the denominator of a rational number in lowest terms
|
|  imag
|      the imaginary part of a complex number
|
|  numerator
|      the numerator of a rational number in lowest terms
|
|  real
|      the real part of a complex number
```

3 Variables & Values

A name that is used to denote something or a value is called a variable. In python, variables can be declared and values can be assigned to it as follows,

```
In [32]: x = 2          # anything after a '#' is a comment
         y = 5
         xy = 'Hey'
         print(x+y, xy) # not really necessary as the last value in a bit of code is displayed by defau
```

```
7 Hey
```

Multiple variables can be assigned with the same value.

```
In [33]: x = y = 1
         print(x,y)
```

```
1 1
```

The basic types build into Python include `float` (floating point numbers), `int` (integers), `str` (unicode character strings) and `bool` (boolean). Some examples of each:

```
In [34]: 2.0            # a simple floating point number
         1e100          # a googol
         -1234567890    # an integer
         True or False # the two possible boolean values
         'This is a string'
         "It's another string"
         print("""Triple quotes (also with '''), allow strings to break over multiple lines.
         Alternatively \n is a newline character (\t for tab, \\ is a single backslash)""")
```

```
Triple quotes (also with '''), allow strings to break over multiple lines.
Alternatively
 is a newline character (         for tab, \ is a single backslash)
```

Python also has complex numbers that can be written as follows. Note that the brackets are required.

```
In [35]: complex(1,2)
         (1+2j) # the same number as above
```

```
Out[35]: (1+2j)
```

4 Operators

4.1 Arithmetic Operators

Symbol	Task Performed
+	Addition
-	Subtraction
/	division
%	mod
	multiplication
//	floor division
	to the power of

```
In [36]: 1+2

Out[36]: 3

In [37]: 2-1

Out[37]: 1

In [38]: 1*2

Out[38]: 2

In [39]: 3/4

Out[39]: 0.75
```

In many languages (and older versions of python) $1/2 = 0$ (truncated division). In Python 3 this behaviour is captured by a separate operator that rounds down: (ie a // b= $\lfloor \frac{a}{b} \rfloor$)

```
In [40]: 3//4.0

Out[40]: 0.0

In [41]: 15%10

Out[41]: 5
```

Python natively allows (nearly) infinite length integers while floating point numbers are double precision numbers:

```
In [42]: 11**300

Out[42]: 2617010996188399907017032528972038342491649416953000260240805955827972056685382434497090341496

In [43]: 11.0**300


        ---------------------------------------------------------------------------

        OverflowError                             Traceback (most recent call last)

        <ipython-input-43-1453d90b43cf> in <module>()
    ----> 1 11.0**300


        OverflowError: (34, 'Numerical result out of range')
```

4.2 Relational Operators

Symbol	Task Performed
==	True, if it is equal
!=	True, if not equal to
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

Note the difference between `==` (equality test) and `=` (assignment)

```
In [44]: z = 2
         z == 2

Out[44]: True

In [45]: z > 2

Out[45]: False
```

Comparisons can also be chained in the mathematically obvious way. The following will work as expected in Python (but not in other languages like C/C++):

```
In [46]: 0.5 < z <= 1

Out[46]: False
```

4.3 Boolean and Bitwise Operators

|Operator|Meaning | | Symbol | Task Performed | |—-|— | |—-|—| |`and`| Logical and | | & | Bitwise And | |`or` | Logical or | | | | Bitwise OR | | | | | ˆ | Exclusive or | |`not` | Not | | ˜ | Negate | | | | | >> | Right shift | | | | | << | Left shift |

```
In [47]: a = 2 #binary: 10
         b = 3 #binary: 11
         print('a & b =',a & b,"=",bin(a&b))
         print('a | b =',a | b,"=",bin(a|b))
         print('a ^ b =',a ^ b,"=",bin(a^b))

a & b = 2 = 0b10
a | b = 3 = 0b11
a ^ b = 1 = 0b1

In [48]: print( not (True and False), "==", not True or not False)

True == True
```

5 Built-in Functions

Python comes with a wide range of functions. However many of these are part of stanard libraries like the `math` library rather than built-in.

5.1 Converting values

Conversion from hexadecimal to decimal is done by adding prefix **0x** to the hexadecimal value or vice versa by using built in **hex( )**, Octal to decimal by adding prefix **0** to the octal value or vice versa by using built in function **oct( )**.

```
In [49]: hex(170)

Out[49]: '0xaa'

In [50]: 0xAA

Out[50]: 170
```

int( ) converts a number to an integer. This can be a single floating point number, integer or a string. For strings the base can optionally be specified:

```
In [51]: print(int(7.7), int('111',2),int('7'))

7 7 7
```

Similarly, the function **str( )** can be used to convert almost anything to a string

```
In [52]: print(str(True),str(1.2345678),str(-2))

True 1.2345678 -2
```

5.2 Mathematical functions

Mathematical functions include the usual suspects like logarithms, trigonometric fuctions, the constant π and so on.

```
In [53]: import math
         math.sin(math.pi/2)
         from math import * # avoid having to put a math. in front of every mathematical function
         sin(pi/2) # equivalent to the statement above

Out[53]: 1.0
```

5.3 Simplifying Arithmetic Operations

round( ) function rounds the input value to a specified number of places or to the nearest integer.

```
In [54]: print( round(5.6231) )
         print( round(4.55892, 2) )

6
4.56
```

complex( ) is used to define a complex number and **abs( )** outputs the absolute value of the same.

```
In [62]: c =complex('5+2j')
         print( abs(c) )

5.385164807134504
```

divmod(x,y) outputs the quotient and the remainder in a tuple(you will be learning about it in the further chapters) in the format (quotient, remainder).

```
In [63]: divmod(9,2)

Out[63]: (4, 1)
```

5.4 Accepting User Inputs

input(prompt), prompts for and returns input as a string. A useful function to use in conjunction with this is **eval()** which takes a string and evaluates it as a python expression.

```
In [64]: abc =  input("abc = ")
         abcValue=eval(abc)
         print(abc,'=',abcValue)
```

```
abc = '3'*3
'3'*3 = 333
```

All of these python notebooks are available at [https://gitlab.erc.monash.edu.au/andrease/Python4Maths.git]

6 Working with strings

6.1 The Print Statement

As seen previously, The **print()** function prints all of its arguments as strings, separated by spaces and follows by a linebreak:

```
- print("Hello World")
- print("Hello",'World')
- print("Hello", <Variable Containing the String>)
```

Note that **print** is different in old versions of Python (2.7) where it was a statement and did not need parenthesis around its arguments.

```
In [2]: print("Hello","World")
```

```
Hello World
```

The print has some optional arguments to control where and how to print. This includes `sep` the separator (default space) and `end` (end charcter) and `file` to write to a file.

```
In [3]: print("Hello","World",sep='...',end='!!')
```

```
Hello...World!!
```

6.2 String Formating

There are lots of methods for formating and manipulating strings built into python. Some of these are illustrated here.

String concatenation is the “addition” of two strings. Observe that while concatenating there will be no space between the strings.

```
In [6]: string1='World'
        string2='!'
        print('Hello' + string1 + string2)
```

```
HelloWorld!
```

The **%** operator is used to format a string inserting the value that comes after. It relies on the string containing a format specifier that identifies where to insert the value. The most common types of format specifiers are:

```
- %s -> string
- %d -> Integer
- %f -> Float
- %o -> Octal
- %x -> Hexadecimal
- %e -> exponential
```

```
In [7]: print("Hello %s" % string1)
        print("Actual Number = %d" %18)
        print("Float of the number = %f" %18)
        print("Octal equivalent of the number = %o" %18)
        print("Hexadecimal equivalent of the number = %x" %18)
        print("Exponential equivalent of the number = %e" %18)
```

```
Hello World
Actual Number = 18
Float of the number = 18.000000
Octal equivalent of the number = 22
Hexadecimal equivalent of the number = 12
Exponential equivalent of the number = 1.800000e+01
```

When referring to multiple variables parenthesis is used. Values are inserted in the order they appear in the paranthesis (more on tuples in the next lecture)

```
In [8]: print("Hello %s %s. This meaning of life is %d" %(string1,string2,42))
```

```
Hello World !
```

We can also specify the width of the field and the number of decimal places to be used. For example:

```
In [10]: print('Print width 10: |%10s|'%'x')
         print('Print width 10: |%-10s|'%'x') # left justified
         print("The number pi = %.2f to 2 decimal places"%3.1415)
         print("More space pi = %10.2f"%3.1415)
         print("Pad pi with 0 = %010.2f"%3.1415) # pad with zeros
```

```
Print width 10: |         x|
Print width 10: |x         |
The number pi = 3.14 to 2 decimal places
More space pi =       3.14
Pad pi with 0 = 0000003.14
```

6.3 Other String Methods

Multiplying a string by an integer simply repeats it

```
In [12]: print("Hello World! "*5)
```

```
Hello World! Hello World! Hello World! Hello World! Hello World!
```

Strings can be tranformed by a variety of functions:

```
In [16]: s="hello wOrld"
         print(s.capitalize())
         print(s.upper())
         print(s.lower())
         print('|%s|' % "Hello World".center(30)) # center in 30 characters
         print('|%s|'% "     lots of space             ".strip()) # remove leading and trailing whitesp
         print("Hello World".replace("World","Class"))
```

```
Hello world
HELLO WORLD
hello world
|         Hello World          |
|lots of space|
Hello Class
```

There are also lost of ways to inspect or check strings. Examples of a few of these are given here:

```
In [21]: s="Hello World"
         print("The length of '%s' is"%s,len(s),"characters") # len() gives length
         s.startswith("Hello") and s.endswith("World") # check start/end
         # count strings
         print("There are %d 'l's but only %d World in %s" % (s.count('l'),s.count('World'),s))
         print('"el" is at index',s.find('el'),"in",s) #index from 0 or -1
```

```
The length of 'Hello World' is 11 characters
There are 3 'l's but only 1 World in Hello World
"el" is at index 1 in Hello World
```

6.4 String comparison operations

Strings can be compared in lexicographical order with the usual comparisons. In addition the `in` operator checks for substrings:

```
In [29]: 'abc' < 'bbc' <= 'bbc'

Out[29]: True

In [30]: "ABC" in "This is the ABC of Python"

Out[30]: True
```

6.5 Accessing parts of strings

Strings can be indexed with square brackets. Indexing starts from zero in Python.

```
In [31]: s = '123456789'
         print('First charcter of',s,'is',s[0])
         print('Last charcter of',s,'is',s[len(s)-1])
```

```
First charcter of 123456789 is 1
Last charcter of 123456789 is 9
```

Negative indices can be used to start counting from the back

```
In [32]: print('First charcter of',s,'is',s[-len(s)])
         print('Last charcter of',s,'is',s[-1])
```

```
First charcter of 123456789 is 1
Last charcter of 123456789 is 9
```

Finally a substring (range of characters) an be specified as using $a : b$ to specify the characters at index $a, a + 1, \ldots, b - 1$. Note that the last charcter is <u>not</u> included.

```
In [33]: print("First three charcters",s[0:3])
         print("Next three characters",s[3:6])
```

```
First three charcters 123
Next three characters 456
```

An empty beginning and end of the range denotes the beginning/end of the string:

```
In [34]: print("First three characters", s[:3])
         print("Last three characters", s[-3:])
```

```
First three characters 123
Last three characters 789
```

6.6 Strings are immutable

It is important that strings are constant, immutable values in Python. While new strings can easily be created it is not possible to modify a string:

```
In [35]: s='012345'
         sX=s[:2]+'X'+s[3:] # this creates a new string with 2 replaced by X
         print("creating new string",sX,"OK")
         sX=s.replace('2','X') # the same thing
         print(sX,"still OK")
         s[2] = 'X' # an error!!!
```

```
creating new string 01X345 OK
01X345 still OK
```

```
        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-35-7e72ddead73e> in <module>()
          4 sX=s.replace('2','X') # the same thing
          5 print(sX,"still OK")
    ----> 6 s[2] = 'X' # an error!!!


        TypeError: 'str' object does not support item assignment
```

```
In [ ]:
```

All of these python notebooks are available at [https://gitlab.erc.monash.edu.au/andrease/Python4Maths.git]

7 Data Structures

In simple terms, It is the the collection or group of data in a particular structure.

7.1 Lists

Lists are the most commonly used data structure. Think of it as a sequence of data that is enclosed in square brackets and data are separated by a comma. Each of these data can be accessed by calling it's index value.

Lists are declared by just equating a variable to '[ ]' or list.

```
In [81]: a = []
In [82]: type(a)

Out[82]: list
```

One can directly assign the sequence of data to a list x as shown.

```
In [83]: x = ['apple', 'orange']
```

7.1.1 Indexing

In python, indexing starts from 0 as already seen for strings. Thus now the list x, which has two elements will have apple at 0 index and orange at 1 index.

```
In [84]: x[0]

Out[84]: 'apple'
```

Indexing can also be done in reverse order. That is the last element can be accessed first. Here, indexing starts from -1. Thus index value -1 will be orange and index -2 will be apple.

```
In [85]: x[-1]

Out[85]: 'orange'
```

As you might have already guessed, x[0] = x[-2], x[1] = x[-1]. This concept can be extended towards lists with more many elements.

```
In [86]: y = ['carrot','potato']
```

Here we have declared two lists x and y each containing its own data. Now, these two lists can again be put into another list say z which will have it's data as two lists. This list inside a list is called as nested lists and is how an array would be declared which we will see later.

```
In [87]: z  = [x,y]
         print( z )

[['apple', 'orange'], ['carrot', 'potato']]
```

Indexing in nested lists can be quite confusing if you do not understand how indexing works in python. So let us break it down and then arrive at a conclusion.

Let us access the data 'apple' in the above nested list. First, at index 0 there is a list ['apple','orange'] and at index 1 there is another list ['carrot','potato']. Hence z[0] should give us the first list which contains 'apple' and 'orange'. From this list we can take the second element (index 1) to get 'orange'

```
In [88]: print(z[0][1])

orange
```

Lists do not have to be homogenous. Each element can be of a different type:

```
In [89]: ["this is a valid list",2,3.6,(1+2j),["a","sublist"]]

Out[89]: ['this is a valid list', 2, 3.6, (1+2j), ['a', 'sublist']]
```

7.1.2 Slicing

Indexing was only limited to accessing a single element, Slicing on the other hand is accessing a sequence of data inside the list. In other words "slicing" the list.

Slicing is done by defining the index values of the first element and the last element from the parent list that is required in the sliced list. It is written as parentlist[a : b] where a,b are the index values from the parent list. If a or b is not defined then the index value is considered to be the first value for a if a is not defined and the last value for b when b is not defined.

```
In [90]: num = [0,1,2,3,4,5,6,7,8,9]
         print(num[0:4])
         print(num[4:])

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

You can also slice a parent list with a fixed length or step length.

```
In [91]: num[:9:3]

Out[91]: [0, 3, 6]
```

7.1.3 Built in List Functions

To find the length of the list or the number of elements in a list, **len( )** is used.

```
In [92]: len(num)

Out[92]: 10
```

If the list consists of all integer elements then **min( )** and **max( )** gives the minimum and maximum value in the list. Similarly **sum** is the sum

```
In [93]: print("min =",min(num),"  max =",max(num),"  total =",sum(num))

min = 0   max = 9   total = 45

In [94]: max(num)

Out[94]: 9
```

Lists can be concatenated by adding, '+' them. The resultant list will contain all the elements of the lists that were added. The resultant list will not be a nested list.

```
In [95]: [1,2,3] + [5,4,7]

Out[95]: [1, 2, 3, 5, 4, 7]
```

There might arise a requirement where you might need to check if a particular element is there in a predefined list. Consider the below list.

```
In [96]: names = ['Earth','Air','Fire','Water']
```

To check if 'Fire' and 'Rajath' is present in the list names. A conventional approach would be to use a for loop and iterate over the list and use the if condition. But in python you can use 'a in b' concept which would return 'True' if a is present in b and 'False' if not.

```
In [97]: 'Fire' in names
```

```
Out[97]: True
```

```
In [98]: 'Space' in names
```

```
Out[98]: False
```

In a list with string elements, **max( )** and **min( )** are still applicable and return the first/last element in lexicographical order.

```
In [99]: mlist = ['bzaa','ds','nc','az','z','klm']
         print("max =",max(mlist))
         print("min =",min(mlist))
```

```
max = z
min = az
```

Here the first index of each element is considered and thus z has the highest ASCII value thus it is returned and minimum ASCII is a. But what if numbers are declared as strings?

```
In [100]: nlist = ['1','94','93','1000']
          print("max =",max(nlist))
          print('min =',min(nlist))
```

```
max = 94
min = 1
```

Even if the numbers are declared in a string the first index of each element is considered and the maximum and minimum values are returned accordingly.

But if you want to find the **max( )** string element based on the length of the string then another parameter `key` can be used to specify the function to use for generating the value on which to sort. Hence finding the longest and shortest string in `mlist` can be doen using the `len` function:

```
In [101]: print('longest =',max(mlist, key=len))
          print('shortest =',min(mlist, key=len))
```

```
longest = bzaa
shortest = z
```

Any other built-in or user defined function can be used.

A string can be converted into a list by using the **list()** function, or more usefully using the **split()** method, which breaks strings up based on spaces.

```
In [102]: print(list('hello world !'),'Hello   World !!'.split())
```

```
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd', ' ', '!'] ['Hello', 'World', '!!']
```

append( ) is used to add a single element at the end of the list.

```
In [103]: lst = [1,1,4,8,7]
          lst.append(1)
          print(lst)
```

```
[1, 1, 4, 8, 7, 1]
```

Appending a list to a list would create a sublist. If a nested list is not what is desired then the **extend( )** function can be used.

```
In [104]: lst.extend([10,11,12])
          print(lst)
```

```
[1, 1, 4, 8, 7, 1, 10, 11, 12]
```

count( ) is used to count the number of a particular element that is present in the list.

```
In [105]: lst.count(1)
```

```
Out[105]: 3
```

index( ) is used to find the index value of a particular element. Note that if there are multiple elements of the same value then the first index value of that element is returned.

```
In [106]: lst.index(1)
```

```
Out[106]: 0
```

insert(x,y) is used to insert a element y at a specified index value x. **append( )** function made it only possible to insert at the end.

```
In [107]: lst.insert(5, 'name')
          print(lst)
```

```
[1, 1, 4, 8, 7, 'name', 1, 10, 11, 12]
```

insert(x,y) inserts but does not replace element. If you want to replace the element with another element you simply assign the value to that particular index.

```
In [108]: lst[5] = 'Python'
          print(lst)
```

```
[1, 1, 4, 8, 7, 'Python', 1, 10, 11, 12]
```

pop( ) function return the last element in the list. This is similar to the operation of a stack. Hence it wouldn't be wrong to tell that lists can be used as a stack.

```
In [109]: lst.pop()
```

```
Out[109]: 12
```

Index value can be specified to pop a ceratin element corresponding to that index value.

```
In [110]: lst.pop(0)
```

```
Out[110]: 1
```

pop( ) is used to remove element based on it's index value which can be assigned to a variable. One can also remove element by specifying the element itself using the **remove( )** function.

```
In [111]: lst.remove('Python')
          print(lst)
```

```
[1, 4, 8, 7, 1, 10, 11]
```

Alternative to **remove** function but with using index value is **del**

```
In [112]: del lst[1]
          print(lst)
```

```
[1, 8, 7, 1, 10, 11]
```

The entire elements present in the list can be reversed by using the **reverse()** function.

```
In [113]: lst.reverse()
          print(lst)
```

```
[11, 10, 1, 7, 8, 1]
```

Note that the nested list [5,4,2,8] is treated as a single element of the parent list lst. Thus the elements inside the nested list is not reversed.

Python offers built in operation **sort( )** to arrange the elements in ascending order. Alternatively **sorted()** can be used to construct a copy of the list in sorted order

```
In [114]: lst.sort()
          print(lst)
          print(sorted([3,2,1])) # another way to sort
```

```
[1, 1, 7, 8, 10, 11]
[1, 2, 3]
```

For descending order, By default the reverse condition will be False for reverse. Hence changing it to True would arrange the elements in descending order.

```
In [115]: lst.sort(reverse=True)
          print(lst)
```

```
[11, 10, 8, 7, 1, 1]
```

Similarly for lists containing string elements, **sort( )** would sort the elements based on it's ASCII value in ascending and by specifying reverse=True in descending.

```
In [116]: names.sort()
          print(names)
          names.sort(reverse=True)
          print(names)
```

```
['Air', 'Earth', 'Fire', 'Water']
['Water', 'Fire', 'Earth', 'Air']
```

To sort based on length key=len should be specified as shown.

```
In [118]: names.sort(key=len)
          print(names)
          print(sorted(names,key=len,reverse=True))
```

```
['Air', 'Fire', 'Water', 'Earth']
['Water', 'Earth', 'Fire', 'Air']
```

7.1.4 Copying a list

Assignment of a list does not imply copying. It simply creates a second reference to the same list. Most of new python programmers get caught out by this initially. Consider the following,

```
In [119]: lista= [2,1,4,3]
          listb = lista
          print(listb)
```

```
[2, 1, 4, 3]
```

Here, We have declared a list, lista = [2,1,4,3]. This list is copied to listb by assigning it's value and it get's copied as seen. Now we perform some random operations on lista.

```
In [120]: lista.sort()
          lista.pop()
          lista.append(9)
          print("A =",lista)
          print("B =",listb)

A = [1, 2, 3, 9]
B = [1, 2, 3, 9]
```

listb has also changed though no operation has been performed on it. This is because you have assigned the same memory space of lista to listb. So how do fix this?

If you recall, in slicing we had seen that parentlist[a:b] returns a list from parent list with start index a and end index b and if a and b is not mentioned then by default it considers the first and last element. We use the same concept here. By doing so, we are assigning the data of lista to listb as a variable.

```
In [121]: lista = [2,1,4,3]
          listb = lista[:] # make a copy by taking a slice from beginning to end
          print("Starting with:")
          print("A =",lista)
          print("B =",listb)
          lista.sort()
          lista.pop()
          lista.append(9)
          print("Finnished with:")
          print("A =",lista)
          print("B =",listb)

Starting with:
A = [2, 1, 4, 3]
B = [2, 1, 4, 3]
Finnished with:
A = [1, 2, 3, 9]
B = [2, 1, 4, 3]
```

7.2 List comprehension

A very powerful concept in Python (that also applies to Tuples, sets and dictionaries as we will see below), is the ability to define lists using list comprehension (looping) expression. For example:

```
In [122]: [i**2 for i in [1,2,3]]

Out[122]: [1, 4, 9]
```

As can be seen this constructs a new list by taking each element of the original `[1,2,3]` and squaring it. We can have multiple such implied loops to get for example:

```
In [123]: [10*i+j for i in [1,2,3] for j in [5,7]]

Out[123]: [15, 17, 25, 27, 35, 37]
```

Finally the looping can be filtered using an **if** expression with the **for - in** construct.

```
In [124]: [10*i+j for i in [1,2,3] if i%2==1 for j in [4,5,7] if j >= i+4] # keep odd i and  j larger t

Out[124]: [15, 17, 37]
```

7.3 Tuples

Tuples are similar to lists but only big difference is the elements inside a list can be changed but in tuple it cannot be changed. Think of tuples as something which has to be True for a particular something and cannot be True for no other values. For better understanding, Recall **divmod()** function.

```
In [125]: xyz = divmod(10,3)
          print(xyz)
          print(type(xyz))
```

```
(3, 1)
<class 'tuple'>
```

Here the quotient has to be 3 and the remainder has to be 1. These values cannot be changed whatsoever when 10 is divided by 3. Hence divmod returns these values in a tuple.
To define a tuple, A variable is assigned to paranthesis ( ) or tuple( ).

```
In [126]: tup = ()
          tup2 = tuple()
```

If you want to directly declare a tuple it can be done by using a comma at the end of the data.

```
In [127]: 27,
```

```
Out[127]: (27,)
```

27 when multiplied by 2 yields 54, But when multiplied with a tuple the data is repeated twice.

```
In [128]: 2*(27,)
```

```
Out[128]: (27, 27)
```

Values can be assigned while declaring a tuple. It takes a list as input and converts it into a tuple or it takes a string and converts it into a tuple.

```
In [129]: tup3 = tuple([1,2,3])
          print(tup3)
          tup4 = tuple('Hello')
          print(tup4)
```

```
(1, 2, 3)
('H', 'e', 'l', 'l', 'o')
```

It follows the same indexing and slicing as Lists.

```
In [130]: print(tup3[1])
          tup5 = tup4[:3]
          print(tup5)
```

```
2
('H', 'e', 'l')
```

7.3.1 Mapping one tuple to another

Tupples can be used as the left hand side of assignments and are matched to the correct right hand side elements - assuming they have the right length

```
In [131]: (a,b,c)= ('alpha','beta','gamma') # are optional
          a,b,c= 'alpha','beta','gamma' # The same as the above
          print(a,b,c)
          a,b,c = ['Alpha','Beta','Gamma'] # can assign lists
          print(a,b,c)
          [a,b,c]=('this','is','ok') # even this is OK
          print(a,b,c)
```

```
alpha beta gamma
Alpha Beta Gamma
this is ok
```

More complex nexted unpackings of values are also possible

```
In [132]: (w,(x,y),z)=(1,(2,3),4)
          print(w,x,y,z)
          (w,xy,z)=(1,(2,3),4)
          print(w,xy,z) # notice that xy is now a tuple
```

```
1 2 3 4
1 (2, 3) 4
```

7.3.2 Built In Tuple functions

count() function counts the number of specified element that is present in the tuple.

```
In [134]: d=tuple('a string with many "a"s')
          d.count('a')

Out[134]: 3
```

index() function returns the index of the specified element. If the elements are more than one then the index of the first element of that specified element is returned

```
In [135]: d.index('a')

Out[135]: 0
```

7.4 Sets

Sets are mainly used to eliminate repeated numbers in a sequence/list. It is also used to perform some standard set operations.

Sets are declared as set() which will initialize a empty set. Also `set([sequence])` can be executed to declare a set with elements

```
In [136]: set1 = set()
          print(type(set1))
```

```
<class 'set'>
```

```
In [137]: set0 = set([1,2,2,3,3,4])
          set0 = {1,2,2,3,3,4} # equivalent to the above
          print(set0)
```

```
{1, 2, 3, 4}
```

elements 2,3 which are repeated twice are seen only once. Thus in a set each element is distinct. However be warned that **{}** is **NOT** a set, but a dictionary (see next chapter of this tutorial)

```
In [138]: type({})

Out[138]: dict
```

7.4.1 Built-in Functions

```
In [139]: set1 = set([1,2,3])

In [140]: set2 = set([2,3,4,5])
```

union( ) function returns a set which contains all the elements of both the sets without repition.

```
In [141]: set1.union(set2)

Out[141]: {1, 2, 3, 4, 5}
```

add( ) will add a particular element into the set. Note that the index of the newly added element is arbitrary and can be placed anywhere not neccessarily in the end.

```
In [142]: set1.add(0)
          set1

Out[142]: {0, 1, 2, 3}
```

intersection( ) function outputs a set which contains all the elements that are in both sets.

```
In [143]: set1.intersection(set2)

Out[143]: {2, 3}
```

difference( ) function ouptuts a set which contains elements that are in set1 and not in set2.

```
In [144]: set1.difference(set2)

Out[144]: {0, 1}
```

symmetric_difference( ) function ouputs a function which contains elements that are in one of the sets.

```
In [145]: set2.symmetric_difference(set1)

Out[145]: {0, 1, 4, 5}
```

issubset( ), isdisjoint( ), issuperset( ) is used to check if the set1/set2 is a subset, disjoint or superset of set2/set1 respectively.

```
In [146]: set1.issubset(set2)

Out[146]: False

In [147]: set2.isdisjoint(set1)

Out[147]: False

In [148]: set2.issuperset(set1)
```

```
Out[148]: False
```

pop( ) is used to remove an arbitrary element in the set

```
In [150]: set1.pop()
          print(set1)
```

```
{1, 2, 3}
```

remove( ) function deletes the specified element from the set.

```
In [151]: set1.remove(2)
          set1
```

```
Out[151]: {1, 3}
```

clear( ) is used to clear all the elements and make that set an empty set.

```
In [152]: set1.clear()
          set1
```

```
Out[152]: set()
```

All of these python notebooks are available at [https://gitlab.erc.monash.edu.au/andrease/Python4Maths.git]

7.5 Strings

Strings have already been discussed in Chapter 02, but can also be treated as collections similar to lists and tuples. For example

```
In [1]:
```

```
In [4]: S = 'Taj Mahal is beautiful'
        print([x for x in S if x.islower()]) # list of lower case charactes
        words=S.split() # list of words
        print("Words are:",words)
        print("--".join(words)) # hyphenated
        " ".join(w.capitalize() for w in words) # capitalise words
```

```
['a', 'j', 'a', 'h', 'a', 'l', 'i', 's', 'b', 'e', 'a', 'u', 't', 'i', 'f', 'u', 'l']
Words are: ['Taj', 'Mahal', 'is', 'beautiful']
Taj--Mahal--is--beautiful
```

```
Out[4]: 'Taj Mahal Is Beautiful'
```

String Indexing and Slicing are similar to Lists which was explained in detail earlier.

```
In [3]: print(S[4])
        print(S[4:])
```

```
M
Mahal is beautiful
```

7.6 Dictionaries

Dictionaries are mappings between keys and items stored in the dictionaries. Alternatively one can think of dictionaries as sets in which something stored against every element of the set. They can be defined as follows:

To define a dictionary, equate a variable to { } or dict()

```
In [5]: d = dict() # or equivalently d={}
        print(type(d))
        d['abc'] = 3
        d[4] = "A string"
        print(d)
```

```
<class 'dict'>
{4: 'A string', 'abc': 3}
```

As can be guessed from the output above. Dictionaries can be defined by using the { `key : value` } syntax. The following dictionary has three elements

```
In [6]: d = { 1: 'One', 2 : 'Two', 100 : 'Hundred'}
        len(d)
```

```
Out[6]: 3
```

Now you are able to access 'One' by the index value set at 1

```
In [32]: print(d[1])
```

```
1
```

There are a number of alternative ways for specifying a dictionary including as a list of `(key,value)` tuples. To illustrate this we will start with two lists and form a set of tuples from them using the **zip()** function Two lists which are related can be merged to form a dictionary.

```
In [9]: names = ['One', 'Two', 'Three', 'Four', 'Five']
        numbers = [1, 2, 3, 4, 5]
        [ (name,number) for name,number in zip(names,numbers)] # create (name,number) pairs
```

```
Out[9]: [('One', 1), ('Two', 2), ('Three', 3), ('Four', 4), ('Five', 5)]
```

Now we can create a dictionary that maps the name to the number as follows.

```
In [21]: a1 = dict((name,number) for name,number in zip(names,numbers))
         print(a1)
```

```
{'Three': 3, 'Four': 4, 'Five': 5, 'One': 1, 'Two': 2}
```

Note that the ordering for this dictionary is not based on the order in which elements are added but on its own ordering (based on hash index ordering). It is best never to assume an ordering when iterating over elements of a dictionary.

By using tuples as indexes we make a dictionary behave like a sparse matrix:

```
In [13]: matrix={ (0,1): 3.5, (2,17): 0.1}
         matrix[2,2] = matrix[0,1] + matrix[2,17]
         print(matrix)
```

```
{(0, 1): 3.5, (2, 17): 0.1, (2, 2): 3.6}
```

Dictionary can also be built using the loop style definition.

```
In [17]: a2 = { name : len(name) for name in names}
         print(a2)
```

```
{'Three': 5, 'Four': 4, 'Five': 4, 'One': 3, 'Two': 3}
```

7.6.1 Built-in Functions

The **len()** function and **in** operator have the obvious meaning:

```
In [14]: print("a1 has",len(a1),"elements")
         print("One is in a1",'One' in a1,"but not Zero", 'Zero' in a1)
```

```
a1 has 5 elements
One is in a1 True but not Zero False
```

clear( ) function is used to erase all elements.

```
In [20]: a2.clear()
         print(a2)
```

```
{}
```

values( ) function returns a list with all the assigned values in the dictionary. (Acutally not quit a list, but something that we can iterate over just like a list to construct a list, tuple or any other collection):

```
In [23]: [ v for v in a1.values() ]
```

```
Out[23]: [3, 4, 5, 1, 2]
```

keys( ) function returns all the index or the keys to which contains the values that it was assigned to.

```
In [24]: { k for k in a1.keys() }
```

```
Out[24]: {'Five', 'Four', 'One', 'Three', 'Two'}
```

items( ) is returns a list containing both the list but each element in the dictionary is inside a tuple. This is same as the result that was obtained when zip function was used - except that the ordering has been 'shuffled' by the dictionary.

```
In [26]: ",  ".join( "%s = %d" % (name,val) for name,val in a1.items())
```

```
Out[26]: 'Three = 3,  Four = 4,  Five = 5,  One = 1,  Two = 2'
```

pop( ) function is used to get the remove that particular element and this removed element can be assigned to a new variable. But remember only the value is stored and not the key. Because the is just a index value.

```
In [27]: val = a1.pop('Four')
         print(a1)
         print("Removed",val)
```

```
{'Three': 3, 'Five': 5, 'One': 1, 'Two': 2}
removed 4
```

```
In [ ]:
```

All of these python notebooks are available at [https://gitlab.erc.monash.edu.au/andrease/Python4Maths.git]

8 Control Flow Statements

The key thing to note about Python's control flow statements and program structure is that it uses indentation to mark blocks. Hence the amount of white space (space or tab characters) at the start of a line is very important. This generally helps to make code more readable but can catch out new users of python.

8.1 Conditionals

8.1.1 If

`python if some_condition:     code block`

```
In [16]: x = 12
         if x > 10:
             print("Hello")
```

```
Hello
```

8.1.2 If-else

`python if some_condition:     algorithm else:     algorithm`

```
In [17]: x = 12
         if 10 < x < 11:
             print("hello")
         else:
             print("world")
```

```
world
```

8.1.3 Else if

`python if some_condition:       algorithm elif some_condition:     algorithm else: algorithm`

```
In [18]: x = 10
         y = 12
         if x > y:
             print("x>y")
         elif x < y:
             print("x<y")
         else:
             print("x=y")
```

```
x<y
```

if statement inside a if statement or if-elif or if-else are called as nested if statements.

```
In [19]: x = 10
         y = 12
         if x > y:
             print( "x>y")
         elif x < y:
             print( "x<y")
             if x==10:
                 print ("x=10")
             else:
                 print ("invalid")
         else:
             print ("x=y")
```

```
x<y
x=10
```

8.2 Loops

8.2.1 For

`python for variable in something:     algorithm`
When looping over integers the **range()** function is useful which generates a range of integers: * range(n) = 0, 1, . . . , n-1 * range(m,n)= m, m+1, . . . , n-1 * range(m,n,s)= m, m+s, m+2s, . . . , m + ((n-m-1)//s) * s

```
In [20]: for ch in 'abc':
             print(ch)
         total = 0
         for i in range(5):
             total += i
         for i,j in [(1,2),(3,1)]:
             total += i**j
         print("total =",total)
```

```
a
b
c
total = 14
```

In the above example, i iterates over the 0,1,2,3,4. Every time it takes each value and executes the algorithm inside the loop. It is also possible to iterate over a nested list illustrated below.

```
In [21]: list_of_lists = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
         for list1 in list_of_lists:
                 print(list1)
```

```
[1, 2, 3]
[4, 5, 6]
[7, 8, 9]
```

A use case of a nested for loop in this case would be,

```
In [22]: list_of_lists = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
         total=0
         for list1 in list_of_lists:
             for x in list1:
                 total = total+x
         print(total)
```

```
45
```

There are many helper functions that make **for** loops even more powerful and easy to use. For example **enumerate()**, **zip()**, **sorted()**, **reversed()**

```
In [23]: print("reversed: ",end="")
         for ch in reversed("abc"):
             print(ch,end=";")
         print("\nenuemerated: ")
         for i,ch in enumerate("abc"):
             print(i,"=",ch,end="; ")
         print("\nzip'ed: ")
         for a,x in zip("abc","xyz"):
             print(a,":",x)
```

```
reversed: c;b;a;
enuemerated:
0 = a; 1 = b; 2 = c;
zip'ed:
a : x
b : y
c : z
```

8.2.2 While

`python while some_condition:       algorithm`

```
In [24]: i = 1
         while i < 3:
             print(i ** 2)
             i = i+1
         print('Bye')
```

```
1
4
Bye
```

8.2.3 Break

As the name says. It is used to break out of a loop when a condition becomes true when executing the loop.

```
In [25]: for i in range(100):
             print(i)
             if i>=7:
                 break
```

```
0
1
2
3
4
5
6
7
```

8.2.4 Continue

This continues the rest of the loop. Sometimes when a condition is satisfied there are chances of the loop getting terminated. This can be avoided using continue statement.

```
In [26]: for i in range(10):
             if i>4:
                 print("Ignored",i)
                 continue
             # this statement is not reach if i > 4
             print("Processed",i)
```

```
Processed 0
Processed 1
Processed 2
Processed 3
Processed 4
Ignored 5
Ignored 6
Ignored 7
Ignored 8
Ignored 9
```

8.3 Catching exceptions

To break out of deeply nested exectution sometimes it is useful to raise an exception. A try block allows you to catch exceptions that happen anywhere during the exeuction of the try block: `python try:     code except <Exception Type> as <variable name>:     # deal with error of this type except:     # deal with any error`

```
In [27]: try:
             count=0
             while True:
                 while True:
                     while True:
                         print("Looping")
                         count = count + 1
                         if count > 3:
                             raise Exception("abort") # exit every loop or function
         except Exception as e: # this is where we go when an exception is raised
             print("Caught exception:",e)
```

```
Looping
Looping
Looping
Looping
Caught exception: abort
```

This can also be useful to handle unexpected system errors more gracefully:

```
In [28]: try:
             for i in [2,1.5,0.0,3]:
                 inverse = 1.0/i
         except: # no matter what exception
             print("Cannot calculate inverse")
```

```
Cannot calculate inverse
```

All of these python notebooks are available at [https://gitlab.erc.monash.edu.au/andrease/Python4Maths.git]

9 Functions

Functions can represent mathematical functions. More importantly, in programmming functions are a mechansim to allow code to be re-used so that complex programs can be built up out of simpler parts.

This is the basic syntax of a function

`python def funcname(arg1, arg2,... argN):     ''' Document String'''     statements return <value>`

Read the above syntax as, A function by name "funcname" is defined, which accepts arguements "arg1,arg2,....argN". The function is documented and it is '"Document String"'. The function after executing the statements returns a "value".

Return values are optional (by default every function returns **None** if no return statement is executed)

```
In [1]: print("Hello Jack.")
        print("Jack, how are you?")
```

```
Hello Jack.
Jack, how are you?
```

Instead of writing the above two statements every single time it can be replaced by defining a function which would do the job in just one line.

Defining a function firstfunc().

```
In [2]: def firstfunc():
            print("Hello Jack.")
            print("Jack, how are you?")
        firstfunc() # execute the function
```

```
Hello Jack.
Jack, how are you?
```

firstfunc() every time just prints the message to a single person. We can make our function **firstfunc()** to accept arguements which will store the name and then prints respective to that accepted name. To do so, add a argument within the function as shown.

```
In [3]: def firstfunc(username):
            print("Hello %s." % username)
            print(username + ',' ,"how are you?")
```

```
In [4]: name1 = 'sally' # or use input('Please enter your name : ')
```

So we pass this variable to the function **firstfunc()** as the variable username because that is the variable that is defined for this function. i.e name1 is passed as username.

```
In [5]: firstfunc(name1)
```

```
Hello sally.
sally, how are you?
```

9.1 Return Statement

When the function results in some value and that value has to be stored in a variable or needs to be sent back or returned for further operation to the main algorithm, a return statement is used.

```
In [6]: def times(x,y):
            z = x*y
            return z
```

The above defined **times( )** function accepts two arguements and return the variable z which contains the result of the product of the two arguements

```
In [7]: c = times(4,5)
        print(c)
```

```
20
```

The z value is stored in variable c and can be used for further operations.

Instead of declaring another variable the entire statement itself can be used in the return statement as shown.

```
In [8]: def times(x,y):
            '''This multiplies the two input arguments'''
            return x*y
```

```
In [9]: c = times(4,5)
        print(c)
```

```
20
```

Since the **times( )** is now defined, we can document it as shown above. This document is returned whenever **times( )** function is called under **help( )** function.

```
In [10]: help(times)
```

```
Help on function times in module __main__:

times(x, y)
    This multiplies the two input arguments
```

Multiple variable can also be returned as a tuple. However this tends not to be very readable when returning many value, and can easily introduce errors when the order of return values is interpreted incorrectly.

```
In [11]: eglist = [10,50,30,12,6,8,100]
```

```
In [12]: def egfunc(eglist):
             highest = max(eglist)
             lowest = min(eglist)
             first = eglist[0]
             last = eglist[-1]
             return highest,lowest,first,last
```

If the function is just called without any variable for it to be assigned to, the result is returned inside a tuple. But if the variables are mentioned then the result is assigned to the variable in a particular order which is declared in the return statement.

```
In [13]: egfunc(eglist)
```

```
Out[13]: (100, 6, 10, 100)
```

```
In [16]: a,b,c,d = egfunc(eglist)
         print(' a =',a,' b =',b,' c =',c,' d =',d)
```

```
a = 100  b = 6  c = 10  d = 100
```

9.2 Default arguments

When an argument of a function is common in majority of the cases this can be specified with a default value. This is also called an implicit argument.

```
In [18]: def implicitadd(x,y=3,z=0):
             print("%d + %d + %d = %d"%(x,y,z,x+y+z))
             return x+y+z
```

implicitadd( ) is a function accepts up to three arguments but most of the times the first argument needs to be added just by 3. Hence the second argument is assigned the value 3 and the third argument is zero. Here the last two arguments are default arguments.

Now if the second argument is not defined when calling the **implicitadd( )** function then it considered as 3.

```
In [19]: implicitadd(4)
```

```
4 + 3 + 0 = 7
```

```
Out[19]: 7
```

However we can call the same function with two or three arguments. A useful feature is to explicitly name the argument values being passed into the function. This gives great flexibility in how to call a function with optional arguments. All off the following are valid:

```
In [20]: implicitadd(4,4)
         implicitadd(4,5,6)
         implicitadd(4,z=7)
         implicitadd(2,y=1,z=9)
         implicitadd(x=1)
```

```
4 + 4 + 0 = 8
4 + 5 + 6 = 15
4 + 3 + 7 = 14
2 + 1 + 9 = 12
1 + 3 + 0 = 4
```

```
Out[20]: 4
```

9.3 Any number of arguments

If the number of arguments that is to be accepted by a function is not known then a asterisk symbol is used before the name of the argument to hold the remainder of the arguments. The following function requires at least one argument but can have many more.

```
In [21]: def add_n(first,*args):
             "return the sum of one or more numbers"
             reslist = [first] + [value for value in args]
             print(reslist)
             return sum(reslist)
```

The above function defines a list of all of the arguments, prints the list and returns the sum of all of the arguments.

```
In [22]: add_n(1,2,3,4,5)
```

```
[1, 2, 3, 4, 5]
```

```
Out[22]: 15
```

```
In [23]: add_n(6.5)
```

```
[6.5]
```

```
Out[23]: 6.5
```

Arbitrary numbers of named arguments can also be accepted using `**`. When the function is called all of the additional named arguments are provided in a dictionary

```
In [27]: def namedArgs(**names):
             'print the named arguments'
             # names is a dictionary of keyword : value
             print("  ".join(name+"="+str(value)
                             for name,value in names.items()))

         namedArgs(x=3*4,animal='mouse',z=(1+2j))
```

```
z=(1+2j)  animal=mouse  x=12
```

9.4 Global and Local Variables

Whatever variable is declared inside a function is local variable and outside the function in global variable.

```
In [29]: eg1 = [1,2,3,4,5]
```

```
Out[29]: 33
```

In the below function we are appending a element to the declared list inside the function. eg2 variable declared inside the function is a local variable.

```
In [32]: def egfunc1():
             x=1
             def thirdfunc():
                 x=2
                 print("Inside thirdfunc x =", x)
             thirdfunc()
             print("Outside x =", x)
```

```
In [33]: egfunc1()
```

```
Inside thirdfunc x = 2
Outside x = 1
```

If a **global** variable is defined as shown in the example below then that variable can be called from anywhere. Global values should be used sparingly as they make functions harder to re-use.

```
In [ ]: eg3 = [1,2,3,4,5]
```

```
In [36]: def egfunc1():
             x = 1.0 # local variable for egfunc1
             def thirdfunc():
                 global x # globally defined variable
                 x = 2.0
                 print("Inside thirdfunc x =", x)
             thirdfunc()
             print("Outside x =", x)
```

```
In [37]: egfunc1()
         print("Globally defined x =",x)
```

```
Inside thirdfunc x = 2.0
Outside x = 1.0
Globally defined x = 2.0
```

9.5 Lambda Functions

These are small functions which are not defined with any name and carry a single expression whose result is returned. Lambda functions comes very handy when operating with lists. These function are defined by the keyword **lambda** followed by the variables, a colon and the respective expression.

```
In [ ]: z = lambda x: x * x
```

```
In [ ]: z(8)
```

9.5.1 Composing functions

Lambda functions can also be used to compose functions

```
In [44]: def double(x):
             return 2*x
         def square(x):
             return x*x
         def f_of_g(f,g):
             "Compose two functions of a single variable"
             return lambda x: f(g(x))
         doublesquare= f_of_g(double,square)
         print("doublesquare is a",type(doublesquare))
         doublesquare(3)
```

```
doublesquare is a <class 'function'>
```

```
Out[44]: 18
```

```
In [ ]:
```

All of these python notebooks are available at [https://gitlab.erc.monash.edu.au/andrease/Python4Maths.git]

10 Classes

Variables, Lists, Dictionaries etc in python are objects. Without getting into the theory part of Object Oriented Programming, explanation of the concepts will be done along this tutorial.

A class is declared as follows

`python class class_name:     methods (functions)`

```
In [3]: class FirstClass:
            "This is an empty class"
            pass
```

pass in python means do nothing. The string defines the documentation of the class, accessible via `help(FirstClass)`

Above, a class object named "FirstClass" is declared now consider a "egclass" which has all the characteristics of "FirstClass". So all you have to do is, equate the "egclass" to "FirstClass". In python jargon this is called as creating an instance. "egclass" is the instance of "FirstClass"

```
In [4]: egclass = FirstClass()
```

```
In [5]: type(egclass)
```

```
Out[5]: __main__.FirstClass
```

```
In [6]: type(FirstClass)
```

```
Out[6]: type
```

Objects (instances of a class) can hold data. A variable in an object is also called a field or an attribute. To access a field use the notation `object.field`. For example:x

```
In [7]: obj1 = FirstClass()
        obj2 = FirstClass()
        obj1.x = 5
        obj2.x = 6
        x = 7
        print("x in object 1 =",obj1.x,"x in object 2=",obj2.x,"global x =",x)
```

```
x in object 1 = 5 x in object 2= 6 global x = 7
```

Now let us add some “functionality” to the class. A function inside a class is called as a “Method” of that class

```
In [12]: class Counter:
             def reset(self,init=0):
                 self.count = init
             def getCount(self):
                 self.count += 1
                 return self.count
         counter = Counter()
         counter.reset(0)
         print("one =",counter.getCount(),"two =",counter.getCount(),"three =",counter.getCount())
```

```
one = 1 two = 2 three = 3
```

Note that the `reset()` and function and the `getCount()` method are callled with one less argument than they are declared with. The `self` argument is set by Python to the calling object. Here `counter.reset(0)` is equivalent to `Counter.reset(counter,0)`. Using **self** as the name of the first argument of a method is simply a common convention. Python allows any name to be used.

Note that here it would be better if we could initialise Counter objects immediately with a default value of `count` rather than having to call `reset()`. A constructor method is declared in Python with the special name `__init__`:

```
In [10]: class FirstClass:
             def __init__(self,name,symbol):
                 self.name = name
                 self.symbol = symbol
```

Now that we have defined a function and added the __init__ method. We can create a instance of FirstClass which now accepts two arguments.

```
In [7]: eg1 = FirstClass('one',1)
        eg2 = FirstClass('two',2)
```

```
In [8]: print(eg1.name, eg1.symbol)
        print(eg2.name, eg2.symbol)
```

```
one 1
two 2
```

dir( ) function comes very handy in looking into what the class contains and what all method it offers

```
In [16]: print("Contents of Counter class:",dir(Counter) )
         print("Contents of counter object:", dir(counter))
```

```
Contents of Counter class: ['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__
Contents of counter object: ['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format_
```

dir( ) of an instance also shows it’s defined attributes so the object has the additional ‘count’ attribute. Note that Python defines several default methods for actions like comparison (__le__ is $\leq$ operator). These and other special methods can be defined for classes to implement specific meanings for how object of that class should be compared, added, multiplied or the like.

Changing the FirstClass function a bit,

Just like global and local variables as we saw earlier, even classes have it’s own types of variables.

Class Attribute : attributes defined outside the method and is applicable to all the instances.

Instance Attribute : attributes defined inside a method and is applicable to only that method and is unique to each instance.

```
In [18]: class FirstClass:
             test = 'test'
             def __init__(self,n,s):
                 self.name = n
                 self.symbol = s
```

Here test is a class attribute and name is a instance attribute.

```
In [22]: eg3 = FirstClass('Three',3)

In [24]: print(eg3.test,eg3.name,eg3.symbol)

TEST Three 3
```

10.1 Inheritance

There might be cases where a new class would have all the previous characteristics of an already defined class. So the new class can "inherit" the previous class and add it's own methods to it. This is called as inheritance.

Consider class SoftwareEngineer which has a method salary.

```
In [25]: class SoftwareEngineer:
             def __init__(self,name,age):
                 self.name = name
                 self.age = age
             def salary(self, value):
                 self.money = value
                 print(self.name,"earns",self.money)

In [26]: a = SoftwareEngineer('Kartik',26)

In [27]: a.salary(40000)

Kartik earns 40000

In [29]: [ name for name in dir(SoftwareEngineer) if not name.startswith("_")]

Out[29]: ['salary']
```

Now consider another class Artist which tells us about the amount of money an artist earns and his artform.

```
In [33]: class Artist:
             def __init__(self,name,age):
                 self.name = name
                 self.age = age
             def money(self,value):
                 self.money = value
                 print(self.name,"earns",self.money)
             def artform(self, job):
                 self.job = job
                 print(self.name,"is a", self.job)

In [34]: b = Artist('Nitin',20)

In [35]: b.money(50000)
         b.artform('Musician')
```

```
Nitin earns 50000
Nitin is a Musician
```

```
In [38]: [ name for name in dir(b) if not name.startswith("_")]
```

```
Out[38]: ['age', 'artform', 'job', 'money', 'name']
```

money method and salary method are the same. So we can generalize the method to salary and inherit the SoftwareEngineer class to Artist class. Now the artist class becomes,

```
In [37]: class Artist(SoftwareEngineer):
             def artform(self, job):
                 self.job = job
                 print self.name,"is a", self.job
```

```
In [38]: c = Artist('Nishanth',21)
```

```
In [39]: dir(Artist)
```

```
Out[39]: ['__doc__', '__init__', '__module__', 'artform', 'salary']
```

```
In [40]: c.salary(60000)
         c.artform('Dancer')
```

```
Nishanth earns 60000
Nishanth is a Dancer
```

Suppose say while inheriting a particular method is not suitable for the new class. One can override this method by defining again that method with the same name inside the new class.

```
In [39]: class Artist(SoftwareEngineer):
             def artform(self, job):
                 self.job = job
                 print(self.name,"is a", self.job)
             def salary(self, value):
                 self.money = value
                 print(self.name,"earns",self.money)
                 print("I am overriding the SoftwareEngineer class's salary method")
```

```
In [40]: c = Artist('Nishanth',21)
```

```
In [41]: c.salary(60000)
         c.artform('Dancer')
```

```
Nishanth earns 60000
I am overriding the SoftwareEngineer class's salary method
Nishanth is a Dancer
```

If the number of input arguments varies from instance to instance asterisk can be used as shown.

```
In [42]: class NotSure:
             def __init__(self, *args):
                 self.data = ' '.join(list(args))
```

```
In [43]: yz = NotSure('I', 'Do' , 'Not', 'Know', 'What', 'To','Type')
```

```
In [44]: yz.data
```

```
Out[44]: 'I Do Not Know What To Type'
```

10.2 Introspection

We have already seen the `dir()` function for working out what is in a class. Python has many facilities to make introspection easy (that is working out what is in a Python object or module). Some useful functions are **hasattr**, **getattr**, and **setattr**:

```
In [45]: ns = NotSure('test')
         if hasattr(ns,'data'): # check if ns.data exists
             setattr(ns,'copy', # set ns.copy
                     getattr(ns,'data')) # get ns.data
         print('ns.copy =',ns.copy)
```

```
ns.copy = test
```

10.3 Scientific Python

10.3.1 Matrices

Dealing with vectors and matrices efficiently requires the **numpy** library. For the sake of brevity we will import this with a shorter name:

```
In [2]: import numpy as np
```

The numpy supports arrays and matrices with many of the features that would be familiar to matlab users. See here quick summary of numpy for matlab users.

Appart from the convenience, the numpy methods are also much faster at performing operations on matrices or arrays than performing arithmetic with numbers stored in lists.

```
In [3]: x = np.array([1,2,3,4,5])
        y = np.array([2*i for i in x])
        x+y # element wise addition
```

```
Out[3]: array([ 3,  6,  9, 12, 15])
```

```
In [4]: X = x[:4].reshape(2,2) # turn into a matrix/table
        2*X # multiply by a scalar
```

```
Out[4]: array([[2, 4],
               [6, 8]])
```

However watch out: array is not quite a matrix. For proper matrix operations you need to use the matrix type. Unlike **array**s that can have any number of dimensions, matrices are limited to 2 dimension. However matrix multiplication does what you would expect from a linear algebra point of view, rather than an element-wise multiplication:

```
In [8]: Y = np.matrix(X)
        print("X=Y=\n",Y)
        print("array X*X=\n",X*X,'\nmatrix Y*Y=\n',Y*Y)
```

```
X=Y=
 [[1 2]
 [3 4]]
array X*X=
 [[ 1  4]
 [ 9 16]]
matrix Y*Y=
 [[ 7 10]
 [15 22]]
```

Much more information on how to use numpy is available at quick start tutorial

10.3.2 Plotting

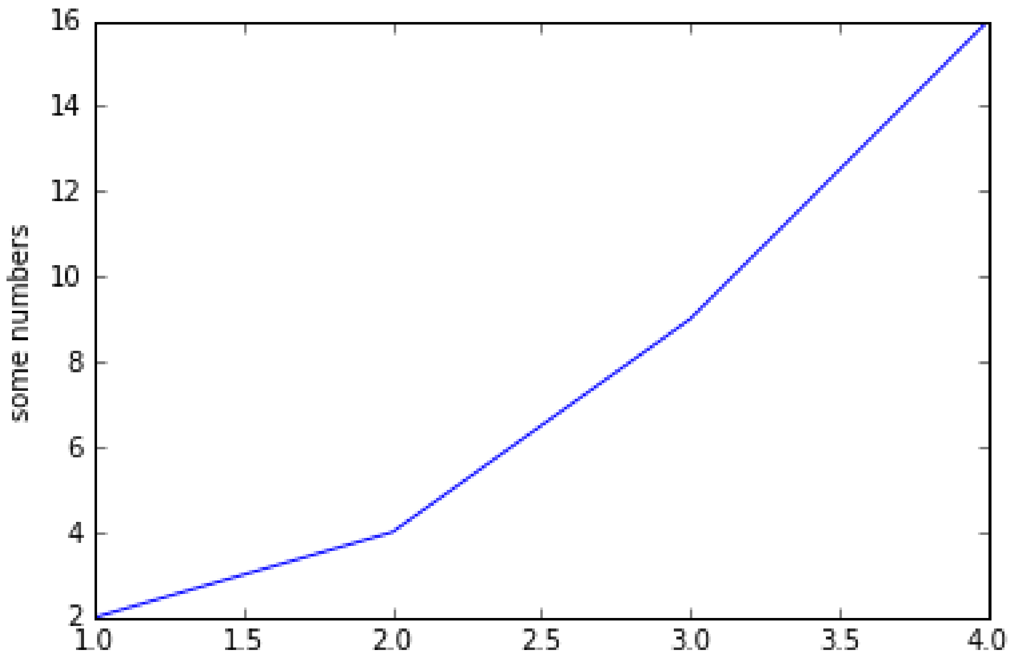
There are lots of configuration options for the **matplotlib** library that we are using here. For more information see [http://matplotlib.org/users/beginner.html]

To get started we need the following bit of 'magic' to make the plotting work:

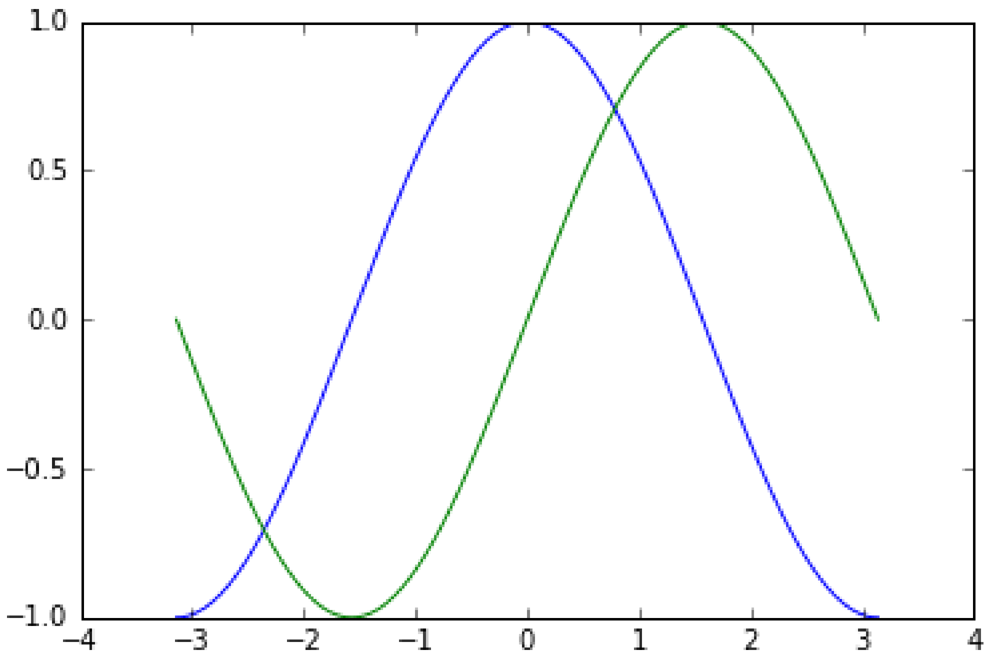
```
In [2]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
```

Now we can try something simple:

```
In [10]: plt.plot([1,2,3,4])
         plt.ylabel('some numbers')
         plt.show()
         plt.plot(range)
         plt.ylabel('some numbers')
         plt.show()
```

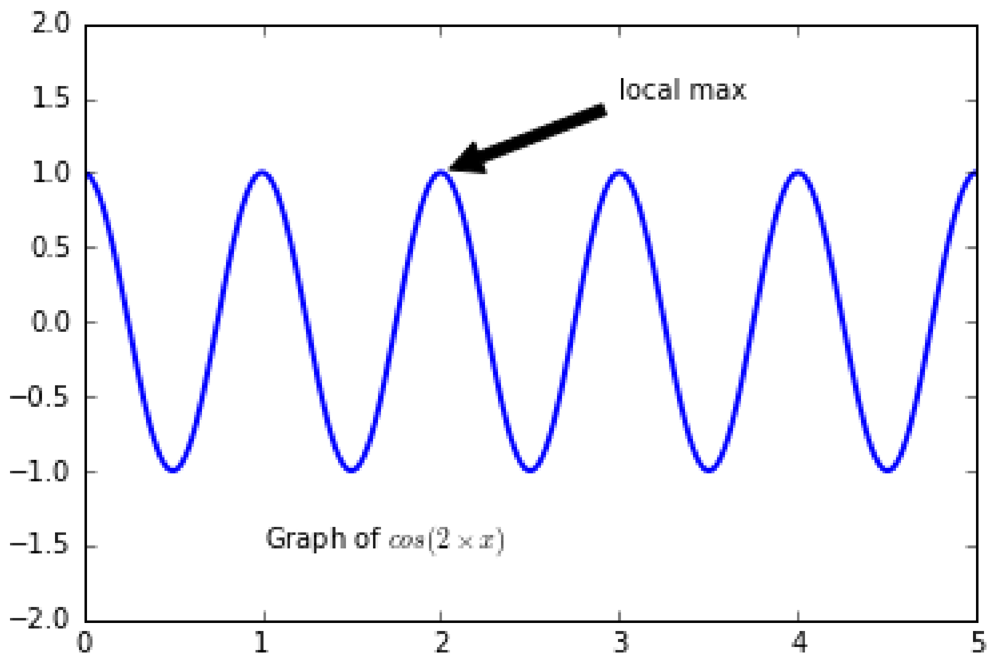


```
In [7]: # A slightly more complicated plot with the help of numpy
        X = np.linspace(-np.pi, np.pi, 256, endpoint=True)
        C, S = np.cos(X), np.sin(X)

        plt.plot(X, C)
        plt.plot(X, S)

        plt.show()
```



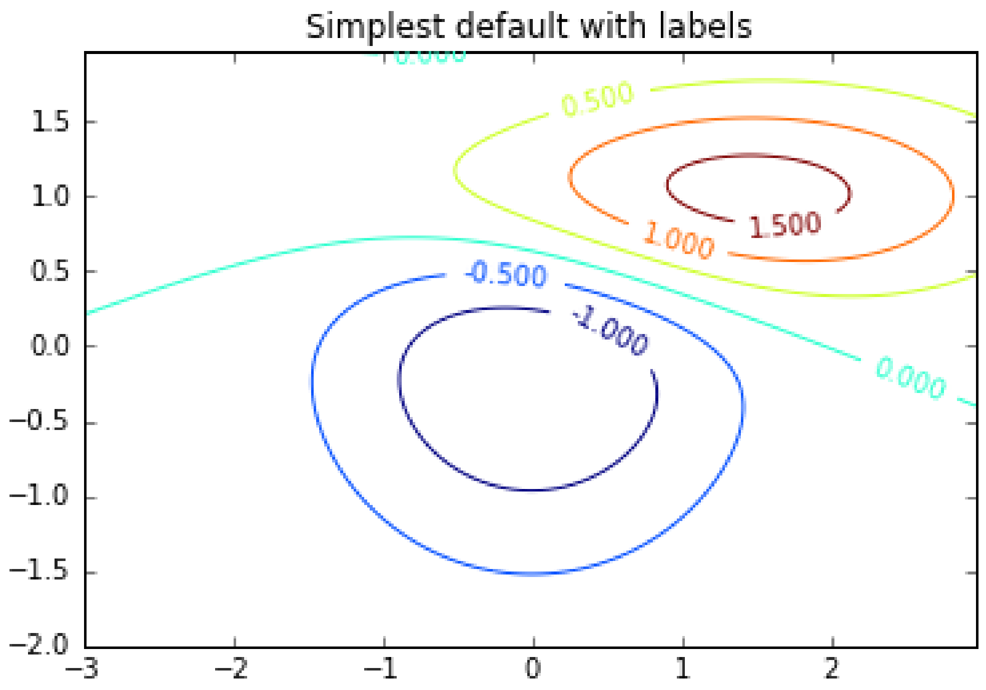
Annotating plots can be done with methods like **text()** to place a label and **annotate()**. For example:

```
In [4]: t = np.arange(0.0, 5.0, 0.01)
        line, = plt.plot(t, np.cos(2*np.pi*t), lw=2)
        plt.annotate('local max', xy=(2, 1), xytext=(3, 1.5),
                     arrowprops=dict(facecolor='black', shrink=0.05),
                     )
        # text can include basic LaTeX commands - but need to mark
        # string as raw (r"") or escape '\' (by using '\\')
        plt.text(1,-1.5,r"Graph of $cos(2\pi x)$")
        plt.ylim(-2,2)
        plt.show()
```



Here is an example of how to create a basic surface contour plot.

```
In [7]: import matplotlib.mlab as mlab # for bivariate_normal to define our surface

        delta = 0.025
        x = np.arange(-3.0, 3.0, delta)
        y = np.arange(-2.0, 2.0, delta)
        X, Y = np.meshgrid(x, y) # define mesh of points
        Z1 = mlab.bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)
        Z2 = mlab.bivariate_normal(X, Y, 1.5, 0.5, 1, 1)
        Z = 10.0 * (Z2 - Z1) # difference of Gaussians

        # Create a simple contour plot with labels using default colors.  The
        # inline argument to clabel will control whether the labels are draw
        # over the line segments of the contour, removing the lines beneath
        # the label
        plt.figure()
        CS = plt.contour(X, Y, Z)
        plt.clabel(CS, inline=1, fontsize=10)
        plt.title('Simplest default with labels')
        plt.show()
```



```
In [ ]:
```

11 Integer & Linear Programming

11.1 An example

Setting up data: cost matrix, demand, supply

```
In [1]: F = range(2) # 2 factories
        R = range(12) # 12 retailers
        C = [[1,2,2,1,3,4,5,7,5,2,3,2],
            [4,5,5,4,1,3,1,2,1,2,4,6]]
        demand = [9,4,2,6,4,5,7,8,3,6,9,5]
        print('Total demand =',sum(demand))
        supply = [ 34, 45]
```

```
Total demand = 68
```

Now we can define a linear program

$$\min \sum_{i\in F}\sum_{j\in R} c_{ij}x_{ij}$$

Subject To

$$\sum_{r\in R} x_{fr} \le s_r \quad \forall f \in F$$

$$\sum_{f\in F} x_{fr} = d_f \quad \forall r \in R$$

$$x \ge 0$$

```
In [2]: from mymip.mycplex import Model

        lp = Model()
        # define double indexed variables and give them a meaningful names
        x = [ [lp.variable('x%dto%d'%(i,j)) for j in R]
             for i in F ]

        lp.min( sum( C[i][j] * x[i][j] for i in F for j in R))
        # constraints can be given names too:
        lp.SubjectTo({"Supply%d"%f:   sum(x[f][r] for r in R) <= supply[f]   for f in F})
        lp.SubjectTo(("Demand%02d"%r, sum(x[f][r] for f in F) == demand[r] ) for r in R)
        for f in F:
            for r in R: x[f][r] >= 0   # all variables non-negative
        lp.param["SCRIND"]=1    # set parameter to show CPLEX output
        lp.optimise()

        print("The minimum cost is",lp.objective())
        for r in R:
            for f in F:
                if x[f][r].x > 0: # amount is not zero
                    print("%.1f from F%d to R%02d"%(x[f][r].x,f,r))
```

```
Tried aggregator 1 time.
LP Presolve eliminated 0 rows and 4 columns.
Aggregator did 12 substitutions.
Reduced LP has 2 rows, 8 columns, and 16 nonzeros.
Presolve time = 0.00 sec. (0.02 ticks)

Iteration log . . .
Iteration:     1   Dual objective     =           122.000000
The minimum cost is 122.0
9.0 from F0 to R00
4.0 from F0 to R01
2.0 from F0 to R02
6.0 from F0 to R03
4.0 from F1 to R04
5.0 from F1 to R05
7.0 from F1 to R06
8.0 from F1 to R07
3.0 from F1 to R08
6.0 from F1 to R09
8.0 from F0 to R10
1.0 from F1 to R10
5.0 from F0 to R11
```

To see how the solve sees this problem, try writing it out to file and printing the contents of the file:

```
In [3]: lp.write("myfirst.lp")
        print(open("myfirst.lp","r").read())
```

```
\ENCODING=ISO-8859-1
\Problem name: Model

Minimize
 obj: x0to0 + 2 x0to1 + 2 x0to2 + x0to3 + 3 x0to4 + 4 x0to5 + 5 x0to6 + 7 x0to7
```

```
      + 5 x0to8 + 2 x0to9 + 3 x0to10 + 2 x0to11 + 4 x1to0 + 5 x1to1 + 5 x1to2
      + 4 x1to3 + x1to4 + 3 x1to5 + x1to6 + 2 x1to7 + x1to8 + 2 x1to9
      + 4 x1to10 + 6 x1to11
Subject To
 Supply0:  x0to0 + x0to1 + x0to2 + x0to3 + x0to4 + x0to5 + x0to6 + x0to7
           + x0to8 + x0to9 + x0to10 + x0to11 <= 34
 Supply1:  x1to0 + x1to1 + x1to2 + x1to3 + x1to4 + x1to5 + x1to6 + x1to7
           + x1to8 + x1to9 + x1to10 + x1to11 <= 45
 Demand00: x0to0 + x1to0  = 9
 Demand01: x0to1 + x1to1  = 4
 Demand02: x0to2 + x1to2  = 2
 Demand03: x0to3 + x1to3  = 6
 Demand04: x0to4 + x1to4  = 4
 Demand05: x0to5 + x1to5  = 5
 Demand06: x0to6 + x1to6  = 7
 Demand07: x0to7 + x1to7  = 8
 Demand08: x0to8 + x1to8  = 3
 Demand09: x0to9 + x1to9  = 6
 Demand10: x0to10 + x1to10  = 9
 Demand11: x0to11 + x1to11  = 5
End
```

11.2 Advanced Usage

All of the raw CPLEX callable library functions as per the can be accessed if required using C interface (see IBM's CPLEX documentation) can be accessed by using the `cplex` object.

For example to identify a minimal conflict set leading to infeasiblity of the problem we can use the CPXrefineconflict and write out the conflict set using CPXclpwrite

```
In [4]: from mymip.mycplex import cplex
        lp.SubjectTo(x[0][0]+x[1][1] >= 50) # make it infeasible
        lp.optimise() # error!
        cplex.CPXrefineconflict(lp.Env,lp.LP,0,0) # 2 null pointers
        cplex.CPXclpwrite(lp.Env,lp.LP,b"conflict.lp") # note binary (ascii) string
        print("#"*10,"Conflict LP","#"*10)
        print(open("conflict.lp","r").read())
```

```
Iteration log . . .
Iteration:     1   Dual objective     =           204.000000
########## Conflict LP ##########
\ENCODING=ISO-8859-1
\Problem name: Model_conflict

Minimize
 obj:
Subject To
 Demand00: x0to0 + x1to0  = 9
 Demand01: x0to1 + x1to1  = 4
 C15:      x0to0 + x1to1 >= 50
\Sum of equality rows in the conflict:
\ sum_eq: x0to0 + x0to1 + x1to0 + x1to1  = 13
Bounds
      x0to0 Free
      x1to1 Free
End
```