UNIVERSITY OF SUSSEX
Scientific Computing
Tutor: Dr. I. Iliev, Office: Pev 3 4C5

Introduction to Python
Adapted from the course textbook, "Numerical Methods in Engineering with Python", 2nd ed. by Jaan Kiusalaas, CUP, 2010. and other sources

Preliminary notes: Before we start, a few notes on obtaining Python for your own laptop/desktop (you can skip this if you are only going to use the university computers), the available Python versions, and some useful tricks.

0.1 Obtaining Python

Note: below are instructions for installing Python on your own machine, NOT on the teaching machines (which you are not allowed to do, anyway, and there Python is already available - see sect. 3 below). Please read these instructions carefully and follow them for your work.

Python is free and I very strongly encourage you to get it installed on your own laptop/PC and use it. It is available for all popular computer platforms: Linux, Windows, or Mac. The Python interpreter can be downloaded from the Python Language Website `http://www.python.org`.

Python normally comes with a nice code editor called Idle that allows you to run programs directly from the editor. An even nicer editor and interactive Python shell is IPython, also available for free at `http://ipython.org/`. This is the one we are going to use for this course. For scientific programming, we also need some additional modules, namely NumPy and Scipy (obtainable from `http://numpy.scipy.org/`). These contain various scientific tools and libraries, as will be explained later. Finally, the Pylab/Matplotlib package (`http://matplotlib.sourceforge.net/`) provides nice plotting capabilities similar to Matlab.

There are also some very nice Python (and generally programming) development enviroments that you can use on your own machine if you like, e.g. Spyder (which I quite like) and Geany. They come with quite convenient and powerful features and if used well can make your life much easier.

All the above sites also provide documentation for downloading and installation on different platforms. If you use Linux or Mac, it is almost certain that Python is already installed (but you may still have to get the additional packages NumPy, Scipy, IPython and Matplotlib). Default Mac versions tend to be older ones, but can be updated. Windows users will likely have to install it. I recommend getting a Python distrubution called Python(x,y) (`https://code.google.com/p/pythonxy/`). This contains Python itself, as well as all scientific packages, IPython, the Spyder programming environment, etc.

I strongly encourage you to use the extensive Python help system (more info below), and any of the many available Python books and web-based materials and tutorials. All above sites have extensive documentation and make tutorials and examples of usage available, so use them! I will include some materials on Study Direct, but those are by no means exhaustive.

0.2 Python versions

There are currently two major flavors of Python:

- Python 2.x and
- Python 3.x

Our discussion will be based on version 2.6, as this is the one installed on the University server we will be using. Any other version of Python 2.x should be fine, too (the latest is 2.8). The numerical extension modules Scipy, Numpy, and Pylab on the university server are setup for Python 2, which is why we will use Python 2.x instead of Python 3.x.

Note: Python 2.x and 3.x are incompatible although the changes only affect a few commands. Make sure that you use version 2.x and also only literature that covers 2.x versions.

0.3 Some useful IPython tricks

IPython is the recommended environment for running Python (see below on how to start it). In it you can do all the usual command-line interaction with Python, but it has some very nice enhancements compared to the basic shell which will make your work easier.

- Several Linux shell commands work also in IPython, such as `ls`('list' – lists all files in current directory/folder), `pwd`('print working directory' – prints current directory/folder), `cd`*dirname* ('change directory' – changes the directory to 'dirname'. If 'dirname' is omitted this sends you to your home folder), etc.
- To get help about objects, functions, etc., type '`help` object'. Just type `help()` to get started.
- Use tab-completion as much as possible: while typing the beginning of an objects name (variable, function, module), press the Tab key and Ipython will complete the expression to match available names. If many names are possible, a list of names is displayed.
- History: press the up (respectively down) arrow to go through all previous (resp. next) instructions starting with the expression on the left of the cursor (put the cursor at the beginning of the line to go through all previous commands)
- You may log your session by using the IPython magic command `%logstart filename.` Your instructions will be saved in a file `filename,` that you can later execute as a script in a different session or e.g. to include in your write-up, if desired.

1 Basic Python Overview

Read carefully this intro and try out all examples. Play with them a bit, making simple modifications/variations, so you can get a better feel how they work. Do the suggested exercises and invent some variations of those. If you encounter any issues please do not hesitate to ask us for help.

Please note that in Python indents *matter* (more details below), thus whenever you do any copying/pasting you should NOT leave any additional empty spaces in front of the commands. Note also that below '>>>' or 'In[num]:' denote the Python/IPython prompts and thus should not be entered by you. Case of letters (capital, lower case) matters too, so e.g. 'Var' is not the same as 'var' or 'vaR'.

1.1 Starting the Python interpreter

On the computer lab PCs, open the 'Programs' menu and then 'Exceed OnDemand' program folder. In it, start the 'xterm' program. This opens an X-terminal window on your screen which is connected to the ITS Linux-based server (this is where the Python software resides). In that window, type '`ipython --pylab`' to start the IPython shell with Matplotlib enabled (note the 2 dashes in front). Often I find it convenient to start two such windows, one to be used for the actual calculations and one to access the help system.

On your own laptop the Python can be started by either:

- On Windows: start IPython (or IDLE).
- On Windows: find the MS-DOS prompt, and type python.exe followed by the return key.
- On Linux/Unix/Mac OS X: find a shell/xterm (called terminal in Applications/Utilities on Mac OS X) and type 'ipython –pylab' (or just 'python' if you want a less sophisticated interface) followed by the return key.

1.2 Setting up your work environment

When you first open the terminal on the Linux server you are in your home directory (folder) - you can confirm this by writing 'pwd' at the command prompt. In principle you can work from there, but in order to keep things neat and usable I advise that you make a new directory where to put your Python codes, by doing 'mkdir *dirname*', where *dirname* is whatever you like to call your directory (better do not use spaces in the name). Then you can do 'cd *dirname*' to change to that new directory and you are all ready to go. When you then start IPython from there you are still in the same place (or you can change to it within IPython itself using the same method). Since we will be accumulating a fair number of codes over this term it is advisable that you organize those in further sub-directories (e.g. one per workshop has been a popular option), but the exact organization you use is up to you.

Note that whenever you request any codes/modules during your work Python looks for those first in the current directory, then in the system Python installation. If the requested code is somewhere else, then it will not be found resulting in an error. Therefore, it is important to know where you are working from and making sure what you need is available there.

1.3 Printing on the screen

This can be done in several slightly different ways (try them out):

```
In [2]: print 'hello'
hello
In [3]: print('hello')
hello
In [4]: print("hello")
hello
```

Printing on screen is clearly useful for directly seeing the results of your code. It is also a very powerful way to check and debug your codes, e.g. by trying it out for different values and checking if the results make sense.

1.4 Using Python as a (quite powerful) calculator

You can print arithmetic expressions directly at the Python prompt and it will evaluate them, e.g (Note: The pound sign (#) denotes the beginning of a comment all characters between # and the end of the line are ignored by the interpreter.):

```
In [5]: 5+8
Out[5]: 13
In [6]: 3**50        # 3 to power of 50
Out[6]: 717897987691852588770249L
In [7]: 2000/25      # integer division
Out[7]: 80
In [8]: 36**0.5      # square root
Out[8]: 6.0
In [9]: 3*(5+10)
Out[9]: 45
```

Note that (unlike many other programming languages) Python can do arbitrary-length arithmetic (when the number has more digits than normal arithmetic Python appends 'L' at the end of the result, for 'Long'). The operations available are:

+	addition
-	subtraction
*	multiplication
/	division
//	integer division
**	exponentiation
%	modular division

Exercise: Try evaluating some more arithmetic expressions on your own.

1.5 Modules

Core Python supports only a few mathematical functions, listed in Figure 1.

`abs(a)`	Absolute value of `a`
`max(`*sequence*`)`	Largest element of *sequence*
`min(`*sequence*`)`	Smallest element of *sequence*
`round(a,n)`	Round `a` to `n` decimal places
`cmp(a,b)`	Returns $\begin{cases} -1 \text{ if } a < b \\ 0 \text{ if } a = b \\ 1 \text{ if } a > b \end{cases}$

Figure 1: Core Python math functions.

What about other mathematical functions like e.g. square root, logarithms, trigonometric functions? Those are not built into the core Python, but are available by loading the math module (we will also discuss other modules later). There are three ways of accessing the functions available in a module. The statement

```
from math import *
```

loads all the function definitions in the math module into the current function or module. The use of this method is *discouraged* because not only is it wasteful, but it can also lead to conflicts with definitions loaded from other modules. A much better way is to load just selected definitions (the ones you actually need) by doing:

```
from math import func1, func2, . . .
```

as illustrated here:

```
In[1]: from math import log,sin
In[2]: print log(sin(0.5))
-0.735166686385
```

The third method, which is used by the majority of programmers, is to make the module available by

```
import math
```

which does not load all functions, as above, but makes them available. The module functions can then be accessed by using the module name as a prefix:

```
In[3]: import math
In[4]: print math.log(math.sin(0.5))
-0.735166686385
```

You can also introduce a different name (e.g. shorter one) for the module:

```
In[3]: import math as m
In[4]: print m.log(math.sin(0.5))
-0.735166686385
```

I strongly recommend that you use the last method for your work. Apart from efficiency, it is a good programming practice and has the advantage that it makes explicit where the function you are using actually comes from. This is sometimes important since similar functions, but with different properties could exist in different modules – we will see examples of that later.

The contents of a module can be printed by calling `dir(module)`. Here is how to obtain a list of the functions in the math module:

```
In [25]: dir(math)  # attributes of  math  object
Out[25]: ['__doc__', '__name__','__package__',
'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil',
'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp',
'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum',
'gamma', 'hypot', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10',
'log1p', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan',
'tanh', 'trunc']
```

Most of these functions should be familiar to you. Note that the module includes two constants: π and e. The same could be done for other modules (or objects of a different type), 'dir()' tells you the attributes of the object being inspected. This type of inquiry is usually referred to as introspection.

Important note: Beware of integer division! Try:

```
print 3/2, 3/2.0, 3.0/2.0, float(3)/2, 3/float(2)
```

What do you get? Does this conform to your expectations? In Python 2.x (NOT in Python 3.x) the division of 2 integers is assumed to be an integer. The best solution is to do:

```
from __future__ import division
```

and try the above example again (here 'future' is Python 3.x). Alternatively, you can make sure you always divide real numbers, rather than integer ones.

Exercises:

1. The Universe is 13.6 billion years old, how many seconds old is it?
2. Compute 2 to the 266th power. This is (roughly) the number of atoms in the Universe.
3. Import the math module and calculate:
   - $\cos(\pi/4)$
   - $\pi - 4atan(1)$.

   Do results conform with your expectations?

1.6 Variables and important data types

A variable can be used to store a certain value or object. In Python, all numbers (and everything else, including functions) are objects. There are several basic types of variables that we will need: integers, floats (real numbers), complex numbers, strings and combinations of these (lists and tuples). Variables are typed dynamically, i.e. their type (which can be checked with 'type(var)') is determined whenever the variable is assigned a value:

```
In[1]: a=1          # integer
In[2]: type(a)
< type  int  >
In[3]: b=1.0        # float/real
In[4]: type(b)
< type  float  >
In[5]: c='1.0'   # string
In[6]: type(c)
< type  str  >
In[7]: d =1+3j      # complex number
In[8]: type(d)
< type  complex  >
```

A string is a sequence of characters enclosed in single or double quotes. Strings on several lines are defined using triple quotes. Strings are concatenated with the plus (+) operator, whereas slicing (:) is used to extract a portion of the string:

```
>>> string1 = 'Press return to exit'
>>> string2 = 'the program'
>>> print string1 + ' ' + string2  # Concatenation
Press return to exit the program
>>> print string1[0:12]  # Slicing
Press return
```

You can also multiply numbers and strings (What does the latter give? Give it a try.).

A string is an immutable object i.e. its individual characters cannot be modified with an assignment statement, and it has a fixed length. An attempt to violate immutability will result in TypeError, as shown here:

```
In [11]: s='Press return to exit'
In [12]: s[0]='p'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/home/iti20/<ipython-input-12-19675791f000> in <module>()
----> 1 s[0]='p'

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

A tuple is a sequence of arbitrary objects separated by commas and enclosed in parentheses. If the tuple contains a single object, a final comma is required; for example, $x = (2,)$. Tuples support the same operations as strings; they are also immutable. Here is an example where the tuple 'rec' contains another tuple $(6, 23, 68)$:

```
>>> rec = ('Smith','John',(6,23,68))   # This is a tuple
>>> lastName,firstName,birthdate = rec # Unpacking the tuple into its parts
>>> print firstName
John
>>> birthYear = birthdate[2]
>>> print birthYear
68
>>> name = rec[1] + ' ' + rec[0]
>>> print name
John Smith
>>> print rec[0:2]
('Smith', 'John')
```

Lists are similar to tuples, but are mutable, i.e. their elements and length can be changed. A list is identified by enclosing it in brackets.

Important: In Python sequences have zero offset, so that $a[0]$ represents the first element, $a[1]$ the second one, and so forth. Similarly, indices in Python always start from 0, so e.g. 2 is the *third* element, etc.

Here is a sampling of operations that can be performed on lists:

```
In [5]: a = [1.0,2.0,3.0]

In [6]: a[0]  # The first element of a
Out[6]: 1.0

In [7]: a[2]  # The third element of a
Out[7]: 3.0

In [8]: a[3]  # Trying to get an element beyond the string length
              # gives an error
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-8-94e7916e7615> in <module>()
----> 1 a[3]

IndexError: list index out of range

In [10]: a.append(4.0) # Append 4.0 to list

In [11]: print(a)
[1.0, 2.0, 3.0, 4.0]

In [12]: print len(a) # gives the length of a
4

In [13]: a[2:4]       # slices elements 3 and 4
Out[13]: [3.0, 4.0]

In [14]: a.insert(0,0.0) # inserts a 0 at the beginning

In [15]: print(a)
[0.0, 1.0, 2.0, 3.0, 4.0]
```

Q: What does a[-1] give?

If a is a mutable object, such as a list, the assignment statement $b = a$ does not result in a new object b, but simply creates a new reference to a. Thus any changes made to b will be reflected in a. To

`a[i]`	returns i-th element of `a`
`a[i:j]`	returns elements i up to $j-1$
`len(a)`	returns number of elements in sequence
`min(a)`	returns smallest value in sequence
`max(a)`	returns largest value in sequence
`x in a`	returns `True` if `x` is element in `a`
`a + b`	concatenates `a` and `b`
`n * a`	creates `n` copies of sequence `a`

Figure 2: Sequence operations.

create an independent copy of a list a, use the statement $c = a[:]$, as shown here:

```
>>> a = [1.0, 2.0, 3.0]
>>> b = a       # 'b' is an alias of 'a'
>>> b[0] = 5.0  # Change 'b'
>>> print a
[5.0, 2.0, 3.0] # The change is reflected in 'a'
>>> c = a[:]    # 'c' is an independent copy of 'a'
>>> c[0] = 1.0  # Change 'c'
>>> print a     # 'a' is not affected by the change
[5.0, 2.0, 3.0]
```

Other operations which could be done on sequences (strings, lists or tuples) are listed in Figure 2. One type of a variable can be converted dynamically into another, e.g.:

```
>>> b = 2     # b is integer type
>>> print b
2
>>> b = b*2.0 # Now b is float type
>>> print b
4.0
```

The assignment $b = 2$ creates an association between the name b and the integer value 2. The next statement evaluates the expression $b * 2.0$ and associates the result with b; the original association with the integer 2 is destroyed. Now b refers to the floating point value 4.0. The functions 'float' and 'int' convert a number (or a string, if it is made of numbers) into a real/integer number. The function 'round' gives the nearest whole number (try them all out):

```
In [1]: a=5.7
```

```
In [3]: int(a)
Out[3]: 5

In [4]: round(a)
Out[4]: 6.0

In [5]: c='345'

In [6]: float(c)
Out[6]: 345.0

In [7]: int(c)
Out[7]: 345
```

Exercises:

1. Create a string variable 'myname' that is initialised to your full name - first, middle and last.
2. Using a slice operator, print your first name only.
3. Using a slice operator, print your last name only.
4. Using the slice and concatenation operators, print your name in the form 'Last name, First name'.
5. Create a new string where your middle name is replaced by your middle initial.
6. Use Python to check how long the 'myname' string is and if it contains the letter 'a'.

1.7 Comparison operators

The comparison operators:

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

are quite useful for programming, as we shall see. The result is a logical expression ('True' or 'False'). These can be used to construct more complicated logical statements using also 'or' and 'and'. Numbers of different type (integer, floating point, etc.) are converted to a common type before the comparison is made. Otherwise, objects of different type are considered to be unequal (e.g. 2 does not equal 2.0). Here are a few examples:

```
In [1]: a=5.7

In [2]: b=4.3

In [3]: print a>b
True

In [4]: a==b
Out[4]: False

In [5]: c=3

In [6]: a>b and c<b
Out[6]: True
```

Exercises:

1. What is the result of the Boolean expression `not` $8 > 5$?
2. What is the result of the Boolean expression `not(True` and False)? Is this what you expected?
3. Write a compound Boolean expression that returns True if the value of the variable `count` is between 1 and 10 inclusive.

1.8 Conditionals

Conditionals are quite useful in construction of algorithms and are an important program control structure. The 'if' construct

```
if condition:
    block
```

executes a block of statements **(which must be indented since that is how Python knows what is inside the 'if' and what is outside!)** if the condition returns true. If the condition returns false, the block is skipped. The if conditional can be followed by any number of 'elif' (short for else if) constructs

```
elif condition:
      block
```

which work in the same manner. The 'else' clause

```
else:
   block
```

can be used to define the block of statements that are to be executed if none of the if-elif clauses is true. The function sign_of_a illustrates the use of the conditionals:

```
def sign_of_a(a):
    if a < 0.0:
        sign = 'negative'
    elif a > 0.0:
        sign = 'positive'
    else:
        sign = 'zero'
    return sign

a = 1.5
print 'a is ' + sign_of_a(a)
```

Here 'def' defines a function, to be discussed below (similar to Matlab functions). Running the program results in the output

```
a is positive
```

Exercises:

1. If you have several nested `if/else` constructs, how does Python know to which `if` an `else` belongs? Try to check your answer using an example.
2. Write a construct that sets the value of a variable called `grade` to the value 4 if a variable named `score` is greater than 70, 3 if `score` is between 60 and 69, 2 if `score` is between 50 and 59, 1 if `score` is between 40 and 49, and 0 otherwise. This statement (roughly) converts British-style grades to American-style ones.

1.9 Loops

One of the most typical tasks we use a computer for is to do a certain task multiple times (e.g. for different input values). Such repetitive tasks are performed by loops. There are two basic types of loops - a 'while' loop and a 'for' loop.

1.9.1 While loops

The while construct

```
while condition:
    block
```

executes a block of (indented) statements if the condition is true. After execution of the block, the condition is evaluated again. If it is still true, the block is executed again. This process is continued until the condition becomes false. The `else` clause

```
else:
    block
```

can be used to define the block of statements that are to be executed if the condition is false. Here is an example that creates the list [1, 1/2, 1/3, . . .]:

```
In [8]: nMax = 5
In [9]: n = 1
In [10]: a = []              # Create empty list
In [11]: while n < nMax:
   ....:         a.append(1.0/n) # Append element to list
   ....:         n = n + 1
   ....:         print a
```

Note the indentations above - they indicate what does or does not belong to the loop (since there is no explicit loop end) and are thus *absolutely necessary*. The IPython shell will do the indentations for you automatically (you end the loop by pressing 'Return' twice), but an external editor might not do it, in which case you have to do it. The output of the program is

```
 [1.0]
 [1.0, 0.5]
 [1.0, 0.5, 0.3333333333333333]
 [1.0, 0.5, 0.3333333333333333, 0.25]
```

1.9.2 For loops

The 'for' loops are explicit in terms of how many times the loop statements will be executed, rather than conditional as the 'while' above:

```
for target in sequence:
    block
```

and repeat the operations in 'block' for all values in 'sequence'. You may add an else clause that is executed after the for loop has finished. The previous program could be written with the for construct as

```
In [12]: nMax = 5
In [13]: a = []              # Create empty list
```

```
In [14]: for n in range(1, nMax):
   ....:         a.append(1.0/n) # Append element to list
   ....:         print a
```

Here n is the target and the list [1, 2, . . . , nMax 1], created by calling the 'range' function (described below), is the sequence. Any loop can be terminated by the break statement. If there is an else cause associated with the loop, it is not executed. The following program, which searches for a name in a list, illustrates the use of break and else in conjunction with a for loop:

```
list = ['Jack', 'Jill', 'Tim', 'Dave']
name = raw_input('Type a name: ') # Python input prompt
for i in range(len(list)):
    if list[i] == name:
       print name,'is number',i + 1,'on the list'
       break
else:
    print name,'is not on the list'
```

Here are the results of two searches:

```
Type a name: 'Tim'
Tim is number 3 on the list

Type a name: 'June'
June is not on the list
```

The 'continue' statement allows us to skip a portion of the statements in an iterative loop. If the interpreter encounters the continue statement, it immediately returns to the beginning of the loop without executing the statements below continue. The following example compiles a list of all numbers between 1 and 99 that are divisible by 7:

```
x = []  # Create an empty list
for i in range(1,100):
   if i%7!= 0: continue # If not divisible by 7, skip rest of loop
   x.append(i) # Append i to the list
print x
```

The printout from the program is

```
[7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84, 91, 98]
```

1.9.3 The range() command

A special type of list is frequently required (often together with for-loops) and therefore a command exists to generate that list: the 'range(n)' command generates a list of integers starting from 0 and going up to but not including n. Here are a few examples:

```
>>> range (3)
[0 , 1 , 2]
>>> range (10)
[0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9]
```

Exercises:

1. Write a `for` loop which calculates the first 10 terms of the series:

$$\sqrt{12}\left(1 - \frac{1}{3 \times 3} + \frac{1}{5 \times 3^2} - \frac{1}{7 \times 3^3} + ...\right)$$

   (This is one way to calculate π using series).

2. Use a `while` loop to calculate

$$x_{k+1} = \frac{1}{2}\left(x_k + \frac{2}{x_k}\right)$$

   until $abs(x_{k+1} - \sqrt{2}) < 0.0001$ (as we will discuss later in the term, this is a way to calculate square roots using just addition and division).

1.10 Reading input and printing output

The intrinsic function for accepting user input (which we already used above) is

```
raw_input(prompt)
```

It displays the prompt and then reads a line of input that is converted to a string. To convert the string into a numerical value, use the function

```
eval(string)
```

The following program illustrates the use of these functions:

```
a = raw_input('Input a: ')
print a, type(a) # Print a and its type
b = eval(a)
print b,type(b) # Print b and its type
```

A convenient way to input a number and assign it to the variable a is

```
a = eval(raw input(prompt))
```

As we saw earlier, output can be displayed with the print statement:

```
print object1, object2, . . .
```

which converts object1, object2, and so on to strings and prints them on the same line, separated by spaces. The newline character '\n' can be used to force a new line. For example,

```
>>> a = 1234.56789
>>> b = [2, 4, 6, 8]
>>> print a,b
1234.56789 [2, 4, 6, 8]
>>> print 'a =',a, '\nb =',b
a = 1234.56789
b = [2, 4, 6, 8]
```

The modulo operator (%) can be used to format a tuple. The form of the conversion statement is

```
'%format1 %format2 ... ' % tuple
```

where format1, format2 ... are the format specifications for each object in the tuple. Typically used format specifications are:

```
wd      Integer
w.df    Floating point notation
w.de    Exponential notation
```

where w is the width of the field and d is the number of digits after the decimal point. The output is right-justified in the specified field and padded with blank spaces (there are provisions for changing the justification and padding). A few examples:

```
>>> a = 1234.56789
>>> n = 9876
>>> print '%7.2f' % a
1234.57
>>> print 'n = %6d' % n # Pad with spaces
n =   9876
>>> print 'n = %06d' % n # Pad with zeroes
n = 009876
>>> print '%12.4e %6d' % (a,n)
1.2346e+003 9876
```

Exercises:

1. Calculate and print on the screen a temperature conversion table from Fahrenheit to Celsius scales. The table should include temperatures from -300 to 210 degrees Fahrenheit (in steps of 5 degrees) and their Celsius equivalents, presented in 2 columns with appropriate headings. Each column should be 10 characters wide, and each temperature should have 3 digits to the right of the decimal point. The conversion formula is:

$$^\circ C = (^\circ F - 32) \times 5/9$$

1.11 Functions

The structure of a Python function is

```
def  func_name(param1,param2,...):
    statements
    return return_values
```

(again, note the indentations!), where param1, param2, ... are the parameters. A parameter can be any Python object, including a function. Parameters may be given default values, in which case the parameter in the function call is optional. If the return statement or return values are omitted, the function returns the null object.

The following example computes the first two derivatives of $\arctan(x)$ by finite differences (we will discuss this later in the course, for now focus on the Python constructs only):

```
from math import atan
def finite_diff(f,x,h=0.0001): # h has a default value
   df =(f(x+h) - f(x-h))/(2.0*h)
   ddf =(f(x+h) - 2.0*f(x) + f(x-h))/h**2
   return df,ddf

x = 0.5
df,ddf = finite_diff(atan,x) # Uses default value of h
print 'First derivative =',df
print 'Second derivative =',ddf
```

Note that the function atan (arctan) is passed to finite_diff as a parameter. The output from the program is

```
First derivative = 0.799999999573
Second derivative = -0.639999991892
```

The number of input parameters in a function definition may be left arbitrary. For example, in the function definition

```
def func(x1,x2,*x3)
```

$x1$ and $x2$ are the usual parameters, also called positional parameters, whereas $x3$ is a tuple of arbitrary length containing the excess parameters (indicated by the star in front of x3). Calling this function with e.g.

```
func(a,b,c,d,e)
```

results in the following correspondence between the parameters:

$$a \leftrightarrow x1, b \leftrightarrow x2, (c, d, e) \leftrightarrow x3$$

The positional parameters must always be listed before the excess parameters. If a mutable object, such as a list, is passed to a function where it is modified, the changes will also appear in the calling program. Here is an example:

```
def squares(a):
    for i in range(len(a)):
    a[i] = a[i]**2

a = [1, 2, 3, 4]
squares(a)
print a
```

The output is:

```
[1, 4, 9, 16]
```

i.e. a was modified on passing through the function.

1.11.1 Lambda Statement

If a function has the form of an expression, it can be defined with the `lambda` statement:

```
func name = lambda param1, param2,...: expression
```

This is called an anonymous function (function literal) and useful for defining small helper function that will be used only once. Here is an example:

```
>>> c = lambda x,y : x**2 + y**2
>>> print c(3,4)
25
```

Exercises:

1. A year is a leap year if it is divisible by 4, unless it is a century that is not divisible by 400. Write a function that takes a year as a parameter and returns `True` if the year is a leap one and `False` otherwise.

2. Write a function that takes two parameters – a pay rate and the number of hours worked – and returns the total pay. Any hours over 40 paid at 1.5 times higher rate.

2 Python shells, scripts and editors

Up to now we have used Python only in interactive mode, i.e. by directly typing commands at the IPython prompt, which then get executed right away. While this is simple, and often convenient, it is also somewhat limiting. An alternative approach is to run Python as scripts/programs. A script is basically a list of Python commands saved in a file, which can then be executed together. Doing this makes it easy to save them for future re-use, re-run them with different input, etc. You can also use scripts written by others. Over the duration of our course there will be many such scripts provided to you, and you will also be writing some of your own.

Creating a script is easy - you should open a text editor and enter the commands in it, taking proper care of any required indentations, etc. Any text editor program is fine for this (but only ones that handle pure text, NOT Word, Notepad, etc.). If you have never used one before I recommend `gedit` as it has a simple, intuitive graphical interface. Examples of more advanced and powerful editors are `emacs` and `vi,` but those have a somewhat steeper initial learning curve. Nevertheless, feel free to try them out. Some of the nice Python programming environments like Geany and Spyder come with their own integrated editor and shell.

You can start the editor from the Xterm window by typing:

```
gedit &
```

(or `emacs,`or `vi` ; **the '&' above is to put the program in background, so you can still use the window for IPython, or anything else you wish**). Alternatively, you can start `gedit` (or other installed software) from within the ipython shell by the same command as above, but preceded by '!' (no quotes), i.e. '`!gedit`& '. (The preceding exclamation tells ipython that this is a normal shell command, not a Python statement.)

Gedit is Python-aware, thus has features like color coding of Python keywords and automatic indentation, which makes your work easier. Before a program can be run, it must be saved as a Python file with the '.py' extension, for example, `myprog.py.` The program can then be executed by typing

```
python myprog.py
```

Note: On Windows double-clicking on the program icon will usually also work. But beware: the program window closes immediately after execution, before you get a chance to read the output. To prevent this from happening, conclude the program with the line

```
raw_input(press return)
```

Double-clicking the program icon also works in Unix and Linux if the first line of the program specifies the path to the Python interpreter (or a shell script that provides a link to Python). The path name must be preceded by the symbols `#!` . On my computer and the teaching machines the path is `/usr/bin/python,` so that all my programs should start with the line

```
#!/usr/bin/python
```

On multiuser systems the path is sometimes `/usr/local/bin/python`.

From within IPython a script can be ran using the magic command 'run':

```
%run prog.py
```

Note that the script file should be in your current IPython working folder/directory. If you are not sure which folder you are currently in, enter `pwd` at the IPython prompt, which tells you the current directory. If you are not at the correct place, you can change it using the `cd directory_name` command.

Once a module (script/function) is saved in a file, you can do `import` just like for any other module. This is a way to use external codes and functions, as well as creating your own. When a module is loaded into a program for the first time with the `import` statement, it is compiled into bytecode and written in a file with the extension .pyc. The next time the program is run, for efficiency the interpreter loads the bytecode rather than the original Python file. If in the meantime changes have been made to the module, the module is automatically recompiled.

It is a good idea to document your modules by adding a docstring at the beginning of each module. The docstring, which is enclosed in triple quotes, should explain what the module does. Here is an example that documents the module error:

```
## module error
''' err(string).

    Prints 'string' and terminates program.

'''
import sys
def err(string):
    print string
    raw_input('Press return to exit')
    sys.exit()
```

The docstring of a module can be printed with the statement

```
print module_name.__doc__
```

(obviously, replace 'module_name' above with the actual name of the module). For example, the docstring of error is displayed by

```
>>> import error
```

```
>>> print error.__doc__
err(string).
     Prints string and terminates program.
```

3 NumPy (Numerical Python) module

The NumPy module is not a part of the standard Python release. As pointed out before, it must be obtained separately and installed on your computer. It is already installed and available on the university machines. You can use it by loading it as a module. This module introduces array objects (vectors, matrices) that are similar to lists, but can be manipulated by numerous functions contained in the module. The size of an array is immutable (i.e. once array is defined you cannot change its size), and no empty elements are allowed. The complete set of functions in NumPy is far too long to be printed in its entirety (you can do `dir(numpy)` to list them). The following list is limited to the most commonly used functions:

```
['complex', 'float', 'abs', 'append', arccos','arccosh', 'arcsin',
'arcsinh', 'arctan', 'arctan2','arctanh', 'argmax', 'argmin', 'cos',
'cosh', 'diag','diagonal', 'dot', 'e', 'exp', 'floor', 'identity',
'inner, 'inv', 'log', 'log10', 'max', 'min','ones','outer', 'pi',
'prod' 'sin', 'sinh', 'size','solve','sqrt', 'sum', 'tan', 'tanh',
'trace','transpose', 'zeros', 'vectorize']
```

We will go over many of those in this tutorial.

3.1 Creating an Array

Arrays can be created in several ways. One of them is to start from a list and use the `array` function to turn it into an array:

```
array(list,dtype = type specification)
```

Notes: Lists can hold any sequence of any Python objects, while arrays can only hold objects of the same type. Arrays are most efficient when the elements are of the basic number types (float, int, complex).

Here are two examples of creating a 2×2 array with floating-point elements:

```
>>> from numpy import array,float
>>> a = array([[2.0, -1.0],[-1.0, 3.0]])
>>> print a
[[ 2. -1.]
[-1. 3.]]
```

```
>>> b = array([[2, -1],[-1, 3]],dtype = float)
>>> print b
[[ 2. -1.]
[-1. 3.]]
```

A couple of quite useful functions which also create arrays are:

```
zeros((dim1,dim2),dtype = type specification)
```

which creates a $dim1 \times dim2$ array and fills it with zeroes, and

```
ones((dim1,dim2),dtype = type specification)
```

which fills the array with ones. The default type in both cases is float. Finally, there is the function

```
arange(from,to,increment)
```

which works just like the range function, but returns an array rather than a list. Here are examples of creating arrays:

```
>>> from numpy import *
>>> print arange(2,10,2)
[2 4 6 8]
>>> print arange(2.0,10.0,2.0)
[ 2. 4. 6. 8.]
>>> print zeros(3)
[ 0. 0. 0.]
>>> print zeros((3),dtype=int)
[0 0 0]
>>> print ones((2,2))
[[ 1. 1.]
[ 1. 1.]]
```

An alternative, similar to Matlab, is to use

```
numpy.linspace(0,n,num=n+1,endpoint=True)
```

We can then 'reshape' the array, converting it from 1D (vector) to 2D (matrix), note that the total sizes of the two should be exactly the same (here - 9 numbers):

```
x = numpy.linspace(0,8,9).reshape(3,3)
```

Caution: *Reshape does not copy - it does not create a new array, but rather creates a different view of the same data.*

We can also do:

```
x = arange(9).reshape(3,3)
```

with the same resulting matrix (but made of integers! If you want real numbers you can do `x=arange(9.).reshape(3,3)`).

Another example, where we first create lists using a function and then convert them into arrays:

```
>>> def f(x):
... return x**3     # sample function
...
>>> n = 5           # no of points in [0,1]
>>> dx = 1.0/(n-1)  # x spacing
>>> xlist = [i*dx for i in range(n)]
>>> ylist = [f(x) for x in xlist]
```

Note that once we have defined the function we can use it as any other function. We can then turn these lists into Numerical Python (NumPy) arrays:

```
>>> import numpy as np
>>> x2 = np.array(xlist)
>>> y2 = np.array(ylist)
```

Alternatively, instead of first making lists with x and $y = f(x)$ data, and then turning lists into arrays, we can make NumPy arrays directly:

```
>>> n = 5                     # number of points
>>> x2 = np.linspace(0, 1, n) # n points in [0, 1]
>>> y2 = np.zeros(n)          # n zeros (float data type)
>>> for i in xrange(n):
...     y2[i] = f(x2[i])
...
```

(`xrange` is similar to `range` but faster).

Caution: Vectors, Column and Row Matrices. Note that an n vector, an $n \times 1$ (column) and $1 \times n$ (row) matrix are three different objects, even if they may contain the same data.

Exercises:

1. Create the following arrays (with correct data types):

```
[[ 1  1  1  1]
 [ 1  1  1  1]
 [ 1  1  1  2]
 [ 1  6  1  1]]

[[0. 0. 0. 0. 0.]
 [2. 0. 0. 0. 0.]
 [0. 3. 0. 0. 0.]
 [0. 0. 4. 0. 0.]
 [0. 0. 0. 5. 0.]
 [0. 0. 0. 0. 6.]]
```

3.2 Accessing and Changing Array Elements

If a is a rank-2 array (i.e. a matrix), then $a[i, j]$ accesses the element in row i and column j, whereas $a[i]$ refers to row i. The elements of an array can be changed by assignment:

```
>>> from numpy import *
>>> a = zeros((3,3),dtype=int)
>>> print a
[[0 0 0]
[0 0 0]
[0 0 0]]
>>> a[0] = [2,3,2]    # Change a row
>>> a[1,1] = 5        # Change an element
>>> a[2,0:2] = [8,-3] # Change part of a row
>>> print a
[[ 2 3 2]
[ 0 5 0]
[ 8 -3 0]]
```

Exercises:

1. Form the following rank-2 array (without typing it in explicitly):

   ```
   [[1  6 11]
   [2  7 12]
   [3  8 13]
   [4  9 14]
   [5 10 15]]
   ```

   and generate a new array containing just its 2nd and 4th rows.

2. Change the fourth row to [1 4 9].

3.3 Operations on Arrays

Arithmetic operators work differently on arrays than they do on tuples and lists – the operation is broadcast to all the elements of the array; that is, the operation is applied to each element in the array individually. Array operations like that are generally very efficient since they are done by underlying optimized and fast C language functions. Here are some examples:

```
>>> from numpy import array
>>> a = array([0.0, 4.0, 9.0, 16.0])
>>> print a/16.0
[ 0.  0.25  0.5625  1. ]
>>> print a - 4.0
[ -4. 0. 5. 12.]
```

All basic arithmetic operations are also performed element-wise (vectors should have the same length, matrices the same dimensions).

```
In [37]: v1=array([6.,7.,9.])
In [38]: v2=array([2.,3.,5.])

In [39]: v1/v2
Out[39]: array([ 3.        ,  2.33333333,  1.8       ])

In [40]: v1*v2
Out[40]: array([ 12.,  21.,  45.])

In [41]: v1+v2
Out[41]: array([  8.,  10.,  14.])

In [42]: v1-v2
Out[42]: array([ 4.,  4.,  4.])
```

The mathematical functions available in NumPy are also broadcast:

```
>>> from numpy import array,sqrt,sin
>>> a = array([1.0, 4.0, 9.0, 16.0])
>>> print sqrt(a)
[ 1. 2. 3. 4.]
>>> print sin(a)
[ 0.84147098 -0.7568025 0.41211849 -0.28790332]
```

Functions imported from the math module will of course work on the individual array elements, but not on the array itself, in which they differ from the NumPy functions (even if they sometimes have the same name). Here is an example:

```
>>> from numpy import array
>>> from math import sqrt
>>> a = array([1.0, 4.0, 9.0, 16.0])
>>> print sqrt(a[1])
2.0
>>> print sqrt(a)
Traceback (most recent call last):
.
.
.
TypeError: only length-1 arrays can be converted to Python scalars
```

Exercises:

1. Create an array x containing 100 equally-spaced numbers from 0 to 2π. Create another array y whose first column contains x, the second one contains $\sin(x)$ and the third one contains $\sin^2(x)$.

3.4 Array Functions

There are numerous functions in NumPy that perform array operations and other useful tasks. Here are a few examples:

```
>>> from numpy import *
>>> A = array([[4,-2,1],[-2,4,-2],[1,-2,3]],dtype=float)
>>> b = array([1,4,3],dtype=float)
>>> print diagonal(A) # Principal diagonal
[ 4. 4. 3.]
>>> print diagonal(A,1) # First subdiagonal
[-2. -2.]
>>> print trace(A) # Sum of diagonal elements
11.0
>>> print argmax(b) # Index of largest element
1
>>> print argmin(A,axis=0) # Indices of smallest col. elements
[1 0 1]
>>> print identity(3) # Identity matrix
[[ 1. 0. 0.]
[ 0. 1. 0.]
[ 0. 0. 1.]]
```

There are three functions in NumPy that compute matrix/vector products. They are illustrated by the program listed below. For more details, see Appendix A2 of the book.

```
from numpy import *
```

```
x = array([7,3])
y = array([2,1])
A = array([[1,2],[3,2]])
B = array([[1,1],[2,2]])

# Dot product
print "dot(x,y) =\n",dot(x,y) #{x}.{y} - scalar (dot) product of 2 vectors
print "dot(A,x) =\n",dot(A,x) #[A]{x}  - maxtrix-vector dot product
print "dot(A,B) =\n",dot(A,B) #[A][B]  - matrix-matrix dot product

# Inner product
print "inner(x,y) =\n",inner(x,y) # {x}.{y}
print "inner(A,x) =\n",inner(A,x) # [A]{x}
print "inner(A,B) =\n",inner(A,B) # [A][B_transpose]

# Outer product
print "outer(x,y) =\n",outer(x,y)
print "outer(A,x) =\n",outer(A,x)
print "Outer(A,B) =\n",outer(A,B)
```

The output of the program is

```
dot(x,y) =
17
dot(A,x) =
[13 27]
dot(A,B) =
[[5 5]
[7 7]]
inner(x,y) =
17
inner(A,x) =
[13 27]
inner(A,B) =
[[ 3 6]
[ 5 10]]
outer(x,y) =
[[14 7]
[ 6 3]]
outer(A,x) =
[[ 7 3]
[14 6]
[21 9]
[14 6]]
Outer(A,B) =
```

```
[[1 1 2 2]
[2 2 4 4]
[3 3 6 6]
[2 2 4 4]]
```

Exercises:

1. Taking the y array from the previous exercise, find the values of its largest and smallest elements in each row.
2. What do `diagonal` and `trace` give you if a 2D array is not with two equal dimensions?

3.5 Linear Algebra Module

NumPy comes with a linear algebra module called linalg that contains routine tasks such as matrix inversion and solution of simultaneous equations (which we will be discussing in more detail soon). For example:

```
>>> from numpy import array
>>> from numpy.linalg import inv,solve

>>> A = array([[ 4.0, -2.0, 1.0], \
[-2.0, 4.0, -2.0], \
[ 1.0, -2.0, 3.0]])
>>> b = array([1.0, 4.0, 2.0])

>>> print inv(A) # Matrix inverse
[[ 0.33333333 0.16666667 0. ]
[ 0.16666667 0.45833333 0.25 ]
[ 0. 0.25 0.5 ]]

>>> print solve(A,b) # Solve [A]{x} = {b}
[ 1. , 2.5, 2. ]
```

Note: The backslash (\) is Python's continuation character.

Exercises:

1. Set up the matrix

```
A=[[1. 2. 3.]
  [4. 5. 6.]
  [7. 8. 1.]]
```

starting from `linspace` and reshaping the array into a 3×3 one.

- Calculate the inverse of A. Check that $AA^{-1} = A^{-1}A = 1$
- Create a vector $b = (6, 15, 16)$. Solve the equation $Ax = b$ for x. Check your answer.

3.6 Copying Arrays

We explained before that if a is a mutable object, such as a list, the assignment statement $b = a$ does not result in a new object b, but simply creates a new reference to a, called a deep copy. This also applies to arrays. To make an independent copy of an array a, use the copy method in the NumPy module:

```
b = a.copy()
```

Exercises:

1. Starting from the array A from the previous exercise, create arrays $B = A$ and `C=A.copy()`.Now change $A[0, 0]$ to 5 and print out both B and C. Did they change?

3.7 Vectorizing Algorithms

Sometimes the broadcasting properties of the mathematical functions in the NumPy module can be utilised to replace loops in the code. This procedure is known as vectorisation. Consider, for example, the expression

$$s = \sum_{i=0}^{100} \sqrt{\frac{i\pi}{100}} \sin \frac{i\pi}{100}$$

The direct approach is to evaluate the sum in a loop, resulting in the following scalar code:

```
from math import sqrt,sin,pi
x=0.0; sum = 0.0
for i in range(0,101):
    sum = sum + sqrt(x)*sin(x)
    x = x + 0.01*pi
print sum
```

The vectorised version of algorithm is

```
from numpy import sqrt,sin,arange,sum
from math import pi
x = arange(0.0,1.001*pi,0.01*pi)
print sum(sqrt(x)*sin(x))
```

Note that the first algorithm uses the scalar versions of `sqrt` and `sin` functions in the math module, whereas the second algorithm imports these functions from the NumPy. The vectorised algorithm is faster, but uses more memory. Mathematical functions in Python without any `if` tests automatically work for both scalar and array (vector) arguments (i.e., no vectorization by the programmer is necessary).

Exercises:

1. Make a function which takes as an input a number n and returns the sum

$$s = \sum_{i=1}^{n} n^2 \ln(n)$$

   using a vectorized algorithm.

2. How much faster is Numpy with vectorised operations compared to standard Python? Try the following two ways to add 1 to a million numbers, first is a standard implicit for loop, the second is a vectorized Numpy array operation:

```
In [1]: lst=range(1000000)
In [2]: %timeit [i+1 for i in lst]

In [3]: arr=arange(1000000)
In [4]: %timeit arr+1
```

   The 'magic' `%timeit` command tests and reports how fast the statement execution is.

3. The execution speed advantage is even better for standard Python functions vs. Numpy functions (`ufunc` in optimized, pre-compiled C-code). Try:

```
In [5]: %timeit [sin(i)**2 for i in arr]
In [6]: %timeit np.sin(arr)**2
```

4 Scoping of Variables

Namespace is a dictionary that contains the names of the variables and their values. The namespaces are automatically created and updated as a program runs. There are three levels of namespaces in Python:

- Local namespace, which is created when a function is called. It contains the variables passed to the function as arguments and the variables created within the function. The namespace is deleted when the function terminates. If a variable is created inside a function, its scope is the function's local namespace. It is not visible outside the function.
- A global namespace is created when a module is loaded. Each module has its own namespace. Variables assigned in a global namespace are visible to any function within the module.

- Built-in namespace is created when the interpreter starts. It contains the functions that come with the Python interpreter. These functions can be accessed by any program unit.

When a name is encountered during execution of a function, the interpreter tries to resolve it by searching the following in the order shown: (1) local namespace, (2) global namespace, and (3) built-in namespace. If the name cannot be resolved, Python raises a `NameError` exception.

Because the variables residing in a global namespace are visible to functions within the module, it is not necessary to pass them to the functions as arguments (although is good programming practice to do so), as the following program illustrates:

```
def divide():
    c = a/b
    print 'a/b =',c
a = 100.0
b = 5.0
divide()
>>>
a/b = 20.0
```

Note that the variable c is created inside the function `divide` and is thus not accessible to statements outside the function. Hence an attempt to move the print statement out of the function fails:

```
def divide():
    c = a/b
a = 100.0
b = 5.0
divide()
print 'a/b =',c


>>>
Traceback (most recent call last):
File ''C:\Python22\scope.py'', line 8, in ?
print c
NameError: name 'c' is not defined
```

Exercises:

1. Create a file called `numpytest.py`. First define two vectors, called x and y. x should be initialized to the vector $(1, 5, -2, 6, 8, 10)$ and y should be $(-1, 0, 1, 1, 2, 3)$. Compute and store in a value a the dot product beteen these two vectors. Store in a vector $z1$ a copy of x but where every value less than zero has been replaced with zero (Hint: you can do this using slices, or the `where` function). Store in $z2$ all of the elements of x that correspond to positions of y that are strictly

greater than zero. Finally, store in $z3$ the sum of all values in y that correspond to even elements of x (Hint: you can use modulus to get evenness: this is % in Python).

5 Visualising Data

Work through the Entought Matplotlib slides (available on Study Direct) and then through the online tutorial at

`http://cs.smith.edu/dftwiki/index.php/MatPlotLib_Tutorial_1`

(there is a link on Study Direct). You can skip any repeated info.

Exercises:

1. Plot a simple graph of a sine function in the range 0 to 3 with a step size of 0.01. Make the line red. Add diamond-shaped markers with size of 5. Add a legend and a grid to the plot.
2. Add to the previous plot a line showing the cosine of the same vector. Make the line blue and dahsed, with circular markers. Change line color and thickness as well as the size and the kind of the marker. Experiment with different styles.
3. Now draw a figure with two sub-plots on separate panels showing the sinus and cosine. Add labels and ticks only to the outermost axes. Place in one of these plots a smaller inset plot showing the exponent of x over the same x interval. Make sure the small plot does not overlap the lines on the larger plot. Use the pylab help and manual if needed.
4. Using the `random` module.

   ```
   import random
   a=random.random() # creates a random number between 0 and 1
   ```

   or `randn(n)`create 2 random vectors x and y, each of length 100. Using the plot command make a scatter plot of x vs. y (hint just use bo as the format).
5. Do the same scatter plot but using the semilogx, semilogy, and loglog plots (check the help). These produce a plot where one or both axes use logscale.
6. See `http://matplotlib.sourceforge.net/gallery.html` and `http://www.scipy.org/Cookbook/Matplotlib` and make a plot (or several) that you think is interesting.