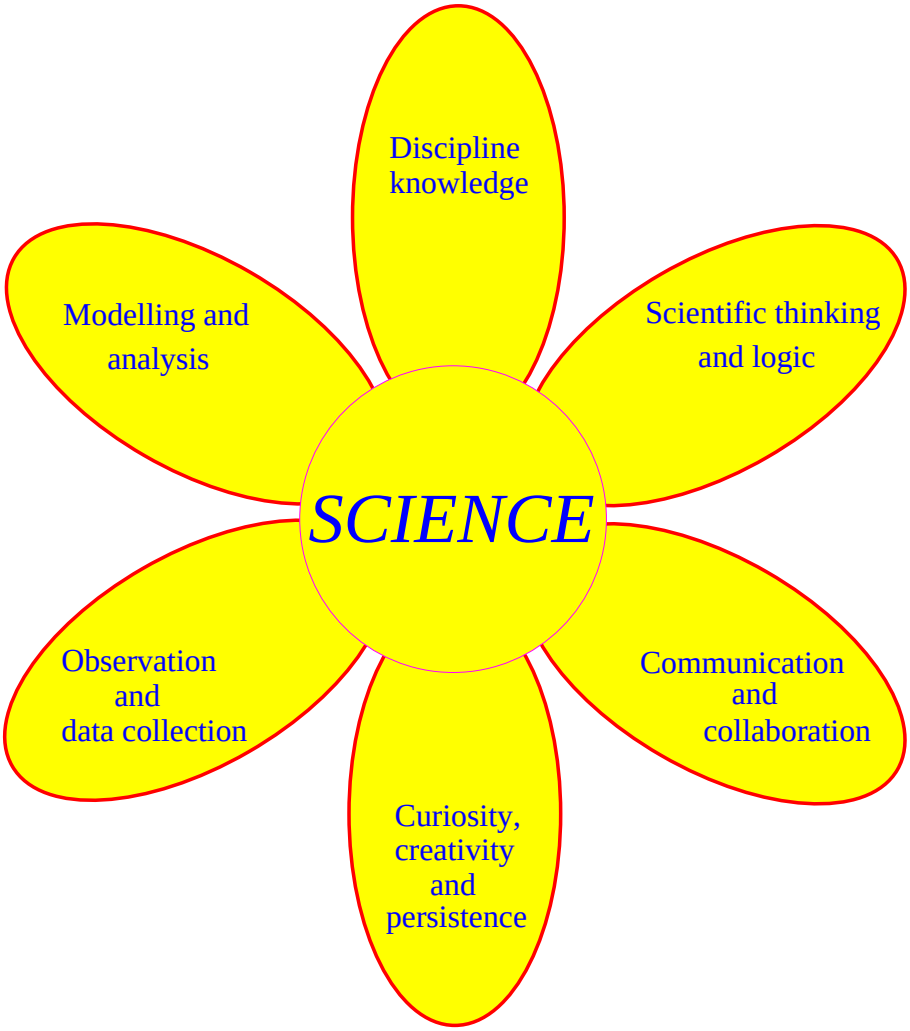
SCIE1000

Theory and Practice in Science



Python tutorial notes

Third edition, 2010

Contents

1 Introduction to Python

1.1 Why Python?

An important part of science is coming up with models for real phenomena. Computation is important when formulating and applying models, particularly when the system that we are dealing with is a complex one. Computation allows us to simulate the behaviour of our models, compare the results of our computations to the results obtained from experiments and then make predictions about future outcomes.

Every computer program and computer model must be implemented in some computer *language*. A computer language is a system of instructions and commands that can be interpreted by a computer to cause the computer to perform certain operations and calculations.

There are many different computer languages, each suited to particular uses. In this course, we use the language *Python*. Python is modern, freely available, fairly easy to learn, used in real science applications, and illustrates many important general computing concepts.

1.2 Using these notes

The purpose of these notes is to teach you how to use Python to represent models and answer questions in science. These notes are designed to be used in conjunction with your weekly computer lab sheets. Each week you will be told when to read which sections of these Python notes. In this way, you will be reading the material in these notes, then doing lab activities to reinforce your learning.

If you are keen to learn Python more rapidly, feel free to read ahead in these notes. The best way to learn computer skills is often to play around. Try things out, see what you can do, and challenge yourself!

For those interested— *Occasionally these notes will contain boxes like this one, headed "For those interested". These will contain information for those of you who may be interested in finding out more about Python or about a particular topic. This material is not assessable.*

1.3 Installing Python at home

You may want to install Python at home so that you can try out some of this exciting stuff on your own. You can download the installation files from the course website.

Log in to `http://blackboard.elearning.uq.edu.au/` and go to the SCIE1000 course area. Follow the instructions from there.

To complete all the activities we will be doing in this course, you will need to install Python, NumPy and matplotlib (in that order).

For more information on all things relating to Python, see `http://www.python.org/`.

If you would like to find out more about the packages we are using, or to find installations for MacOS or linux, the following links might also be useful:

- **ActivePython** is the distribution of Python which we will be using in this course. For more information see `http://www.activestate.com/Products/activepython/`.

- **NumPy** provides mathematical tools for Python, and is needed to use matplotlib. See `http://numpy.scipy.org/`.
- **matplotlib** provides graphing capabilities for Python, and can be found at `http://matplotlib.sourceforge.net/`.

1.4 Getting started

In this course we will be using Python through a program called `IDLE`, which is an Integrated Development Environment (IDE) for Python. An IDE typically consists of an editor (which often uses colours to highlight features of the language) and other tools to aid in software development and maintenance.

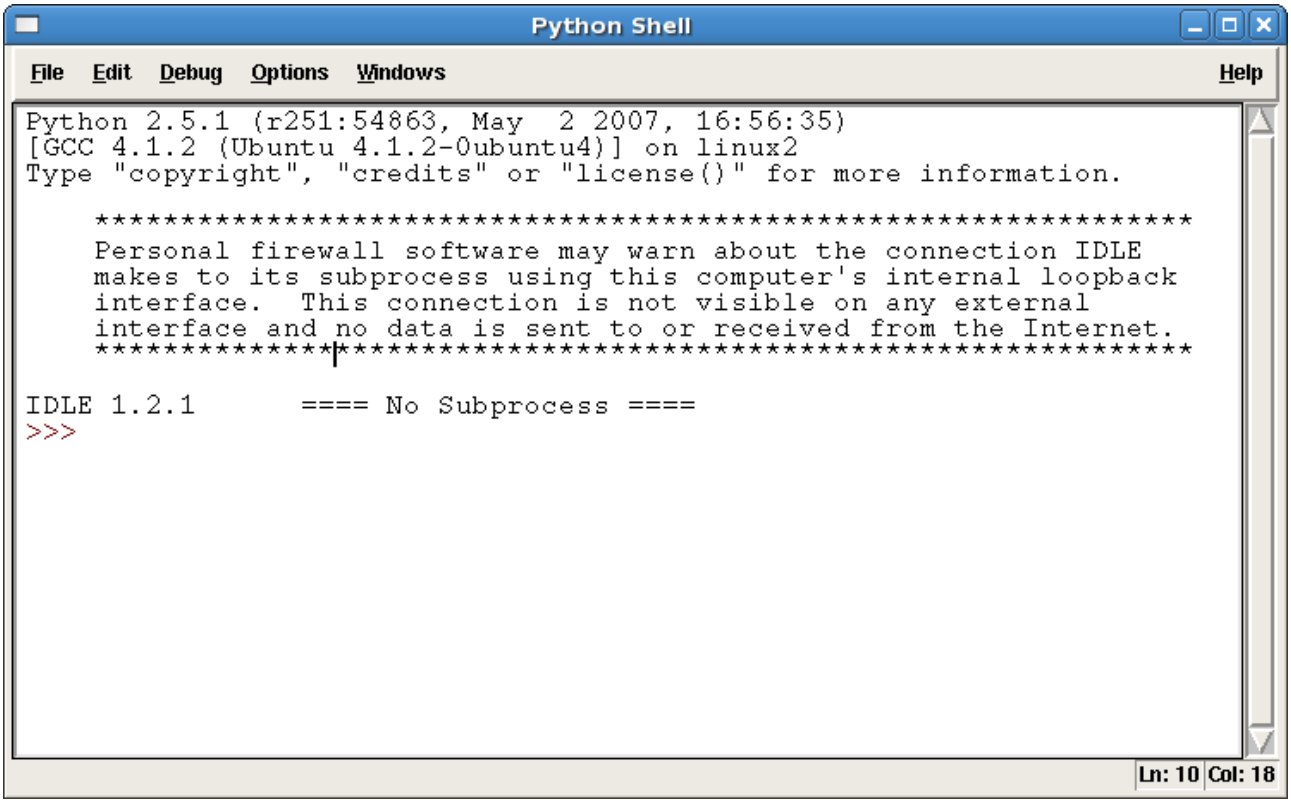
Opening IDLE

To open IDLE, go to the Start Menu and choose:

Programs → ActiveState ActivePython 2.5 → idle.

If you need to do something different from this in order to start IDLE, your tutor will give you instructions.

The screen in `IDLE` will probably look something like this (there may be variations, depending on which version of Python you are running):



This window is the main Python window. All output from your program and any error messages will appear in this screen. (You can also type Python commands directly into this window, but we will not usually do that in SCIE1000.)

2 Basic use of Python

We will be writing and running Python programs throughout semester. In this section we first describe in general what a program is, how to write and save programs, and how to run programs. Then we will cover some Python commands that can be included in your programs.

2.1 Programming

2.1.1 What is a program?

A *program* is simply a set of instructions which a computer is able to interpret and carry out. You may be familiar with programs such as Firefox or Microsoft Word. Even the Python programming language is itself a program.

When we have a problem to solve computationally, we usually want to write a sequence of Python commands, save them to a file, and then run them.

Advantages of doing this include:

- Programs can be run multiple times, and in multiple places.
- Programs can be debugged more easily. For example, once a section of a program has been thoroughly checked (or even proved to be correct) then it does not need to be checked again.
- Teams of people can design and write different parts of the program.
- Problems of much greater complexity can be solved.

2.1.2 Writing and saving programs

To create and save a program:

- In `IDLE`'s "Python Shell" window, select **File** → **New Window**. This will open a new window, called an *editor* window, in which you can enter your code.
- Type in the Python commands that you want in your program. When you type the commands, they will not run straight away. This allows you to enter multiple lines, and go back to fix any errors you make before you run the program.
  - Do not include any spaces or tabs at the start of your lines; we will see why later.
- When you have finished choose **File** → **Save As** in the editor window. You may like to save your program in a special Python programming folder (which you will need to create the first time you want to use it).

To open an existing program, select **File** → **Open** and choose the file.

Naming your programs

Python programs are conventionally named

`something.py`

where `something` *is a name that explains the purpose of the program. The* `.py` *extension is well-recognised for Python programs.*

2.1.3 Running a program

To run a program that is in an editor window, select **Run** → **Run Module** from that editor window. This will load your commands into the main Python window and run them, as if you had typed them directly into the window. (If you have changed the program since you last saved it, you will be prompted to save it again.)

When you run your program, you will notice that the Python window shows the following message:

```
>>> ================ RESTART ================
```

This tells you that Python is clearing the values of any variables you have previously created. This means that earlier calculations do not interfere with your current work, but also that you cannot use values from previous calculations. Any output from your program will appear under that message.

2.2 Comments in Python programs

Any lines in a Python program that starts with a `#` are *comments*, and will be ignored by Python. You should use comments to explain what your program is doing, which will help anyone reading your program to understand what is happening. Always try to use comments where appropriate.

2.3 Importing modules

Like most computer languages, Python provides access to a large collection of mathematical functions and other useful operations. These functions are organised into *modules*.

In this course we will be using the mathematical functions defined in the module called `pylab`. This module contains many useful functions, and we will be using it all of the time.

To instruct Python to allow us to access all of the functions defined in the `pylab` module (as well as to avoid some odd behaviour when dividing integers), use the commands:

```
from __future__ import division
from pylab import *
```

These commands are very important—we will be using them all of the time. These lines should appear at the top of all of your Python programs.

2.4 Printing to the screen

The next Python command we will see is the *print* command, which outputs text and the results of calculations to the screen.

Printing text

To print something to the computer screen, use the `print` *command.*

The `print` command may be used in a number of ways.

If you want to print a blank line to the screen, use:

```
print
```

To print a message to the screen, use:

```
print 'message'
```

The message must be enclosed in quotation marks. You may use either single quotes `'like this'`, or double quotes `"like this"`. You may not mix the two. These two symbols occur on the same key of your keyboard.

Note that if you open quotation marks in a command, **you must close them**. If you try to enter a command without closing your quotation, the command will not be complete. Python will open a new blank line and will not run the command until you close the quotation and then press enter.

To evaluate a mathematical expression and print the answer, use:

```
print expression
```

Note that there are no quotation marks in this case. If you use quotation marks, whatever is *within* the quotation marks will be printed *exactly as it is*, whereas anything that is not inside quotation marks will be evaluated as an expression, and the *answer* will be printed.

If you want to print multiple items (this may include a mixture of expressions and messages), you can separate the items by commas.

The following example is a simple Python program that demonstrates these uses of the `print` command. Note that the line numbers and vertical line at the left have been added into these notes for ease of reference. They are not a part of the program.

(If you like, you can type this program into a Python edit window, then save it and run it. Take care to type the brackets and quotation marks exactly as they appear.)

Python Example 2.4.1

Here is the program:

```
1 from __future__ import division
2 from pylab import *
3
4 # Print some messages.
5
```

```
6 print 'This is a message'
7 print "This is also a message!"
8 print 'This is first', 'and this is second'
9
10 # Print the result of a calculation.
11
12 print 3+4
```

Here is the output from running the program:

```
1 This is a message
2 This is also a message!
3 This is first and this is second
4 7
```

Note that Lines 1 and 7 of the program contain comments. Also, you can use blank lines (like Lines 2, 6 and 8) to make your program more readable.

2.5 Numerical calculations

As suggested by Line 9 of the previous program, Python can use standard mathematical operations. The following table summarises the Python symbols for a number of these operations. (In each case, the letters a and b represent numbers.)

Operation	**Mathematical representation**	**Python spelling**
Addition	$a + b$	`a+b`
Subtraction	$a - b$	`a-b`
Multiplication	$a \times b$	`a*b`
Division	$a \div b$	`a/b`
Exponentiation	a^b	`a**b`
Brackets	(...)	`(...)`
Remainder	a mod b	$a\%b$

Important note

You may have seen `a^b` *used to represent* a^b *on your calculator. In Python* `a^b` *means something completely different so be careful to use* `a**b` *in Python!*

The following example is a simple Python program that demonstrates some mathematical operations.

Python Example 2.5.1

Here is the program:

```
1 from __future__ import division
2 from pylab import *
3
4 print 12/4
```

```
print -6*4
print 10%6
print 10%5

# Python correctly applies order of operations:

print 2+3*4
print (2+3)*4
print 2**5

# You can combine text and calculations in print commands.

print 'All up, 10 cows have',10*4,'legs'
```

Here is the output from running the program:

```
3
-24
4
0
14
20
32
All up, 10 cows have 40 legs
```

Permission to experiment and play

Important: *you have permission to play! Do not be afraid of trying something new. If you type something into Python and it does not work, you will not destroy modern civilisation. You will not even crash the computer.*

In the examples above, none of the expressions which we typed contained spaces. In Python, you can add spaces into an expression anywhere you like, provided that the expression still makes sense. Adding spaces to your expressions in sensible places can make them easier to read and understand, particularly when you start to write more complicated expressions. The following example demonstrates writing expressions with sensible spacing.

Python Example 2.5.2

```
from __future__ import division
from pylab import *

# Adding one space between numbers and symbols is reasonable.
print 6 + 4

# You normally do not need a space between brackets and numbers.
print (2 + 3) * (6 - 4)

# Sometimes spaces are used to visually show order of operations.
print 2 + 3*4 + 5*6
```

When putting spaces into your expressions, remember that your main concern is *communication*. There is no one correct answer. Use your judgement.

2.6 Integers and real numbers

Internally, computers store values in binary. That is, they use 1s and 0s to represent them. Depending on the kind of value, computers may represent values in different ways.

- **integers** can be represented exactly in binary.
- **real numbers** usually cannot be exactly represented using binary. Instead, the computer stores a very close approximation to the number. The term *floating point* number refers to a number which is not stored as an exact integer, but is stored as a real number instead.

Because computers cannot always store real numbers exactly, calculations can sometimes result in numbers which are not exactly what you expect. The errors are usually so small that they can be ignored.

Python uses the symbol `e` to represent scientific notation. So for instance, Python would represent 6.02×10^{23} as `6.02e+023`, and 3×10^{-4} as `3e-04`.

For those interested— *Most programming languages can only store integers in a certain range (e.g. between -2147483648 and 2147483647). Python has the added feature that it can store arbitrarily large integers. Python shows that a number is stored in large integer format by appending the letter* `L` *to the number.*
Try typing `2**256` *and see what happens.*

3 Variables and mathematical functions

3.1 Variables

A powerful feature of computers and programming languages is their ability to "remember" values. This is done by *assigning values* to *variables*.

3.1.1 Variable names

Every variables has a name, which is used to access it. A variable name:

- is made up of one or more letters, digits and underscores.
- must begin with a letter or an underscore.
- is **case sensitive**. For example, `sideLength`, `SideLength` and `SIDELENGTH` are all different variables.

When writing programs, always remember that someone else may need to read your program later. You should always choose *meaningful* variable names which tell the reader what the variables are used for.

3.1.2 Assigning to variables

To assigned a value to a variable use the command

```
variable = expression
```

where `variable` is the name of the variable, and `expression` is either a value (such as `3` or `-2.25`), or an expression (such as `2 + 4`). If you use an expression, Python will calculate the result of the expression and assign that value to the variable.

Once you have assigned a value to a variable, you can use that variable name in subsequent calculations. You can even assign its value to another variable.

For those interested— *Many programming languages require you to declare variables and give them a datatype. In Python you do not need to do this.*

3.1.3 Accessing variables

Once a variable has been created, it is accessed using its name. So if you want to print the value of a variable named `frederic` to the screen you can use the command:

```
print frederic
```

Remember that if you were to write `print 'frederic'` (notice the quotes), then Python would print the word "frederic" to the screen. Because the command above does not use quotes, Python accesses the variable *named* `frederic`.

You can also use variable names in calculations. For example:

Python Example 3.1.1

Here is the program:

```
width = 20
height = 45
print width, height
print width * height
perimeter = 2 * width + 2 * height
print 'The perimeter is', perimeter
```

Here is the output from running the program:

```
20 45
900
The perimeter is 130
```

If you try to access a variable that does not yet have a value, you will get an error message.

3.2 Functions and parameters

As in mathematics, many Python functions act on one or more values (called *arguments*), and produce some output.

Arguments (also known as **parameters**) are values that are passed in to a function. An argument can be a variable, a value or a mathematical expression.

To use a Python function to calculate the result of a mathematical function, type

`functionName(argument)`

where `functionName` is the name Python gives to the function and `argument` is the argument to the function. For example:

Python Example 3.2.1

Here is the program:

```
from __future__ import division
from pylab import *
B = 9
SqB = sqrt(B)
print 'The square root of ',B,' is ',SqB
print 'The square root of 64 is',sqrt(64)
```

Here is the output from running the program:

```
The square root of 9 is 3
The square root of 64 is 8
```

3.2.1 Common mathematical functions

Here is a list of some mathematical functions in Python. In this list, `...` represents an argument.

- `sqrt(...)` which gives the square root of the argument.
- `sin(...)` which gives the sine of the argument (where the argument is given in radians).
- `cos(...)` gives the cosine of the argument (where the argument is in radians).
- `tan(...)` gives the tangent of the argument (where the argument is in radians).
- `exp(...)` gives e raised to the given power.
- `log(...)` gives ln of the given argument.
- `log10(...)` gives $\log_{10}$ of the given argument.

Python also includes the constant `pi`, which (approximately) equals π.

All of the trigonometric functions in Python operate on angles measured in radians. Any angles measured in degrees must first be converted to radians, by multiplying the angle by 2π and dividing by 360 (or, equivalently, multiplying by π and dividing by 180).

The following example demonstrates the use of mathematical functions.

Python Example 3.2.2

Here is the program:

```
from __future__ import division
from pylab import *
# Access the constant pi
print pi

print sin(pi/2)
print exp(1)
print log10(1000)
# Evaluate sin of 90 degrees. We first need to convert degrees to radians.
print sin(90*pi/180)
print "45 degrees = ", 45*pi/180, " radians"
```

Here is the output from running the program:

```
3.1415926535897931
1.0
2.71828182846
3.0
1.0
45 degrees =  0.785398163397  radians
```

3.3 The input command

Often when writing programs it is useful to be able to ask the user for some input values. To do this, we use the `input` command. In Python, `input` acts in a similar way to other functions. There are two ways to use it:

To read a value from the user and store that value into the variable `var`, use the command

```
var = input()
```

When reading a value from the user, you can instruct Python to write some message to the screen as a prompt. Suppose that the message prompt you wish to write to the screen is `prompt`. To do this you would use the command

```
var = input('prompt')
```

The command above will write the prompt to the screen, then input a value and store it in the variable `var`.

The following example program shows how to use the `input` command.

Python Example 3.3.1

Here is the program:

```
1 from __future__ import division
2 from pylab import *
3
4 a = input('Tell me a number: ')
5 b = input('Tell me another number: ')
6
7 c = a + b
8 d = a * b
9
10 print a, '+', b, '=', c
11 print a, 'x', b, '=', d
```

Here is the output from running the program twice:

```
1 >>>
2 Tell me a number: 8
3 Tell me another number: 7
4 8 + 7 = 15
5 8 x 7 = 56
6
7 >>>
8 Tell me a number: 987
9 Tell me another number: 654
10 987 + 654 = 1641
11 987 x 654 = 645498
```

3.4 Software errors and bugs

Even the best computer programmers will sometimes (even often) make errors or *bugs* in their programs. A key skill in programming is minimising the number of errors, and then identifying and fixing any that occur. There are many different types of error, including incomplete problem description, design faults in the software, unanticipated 'special cases', coding errors and logic errors.

In SCIE1000 the result of any errors will be minor. You will probably get a bit frustrated and might even need to ask for help, but no lasting damage will occur. In real life, the consequences arising from programming errors can be very serious: for example, they have caused planes to crash, rockets to explode and entire transport systems to fail. As a result, there is an entire branch of computer science devoted to techniques for minimising errors, and even proving that software is correct.

In this course we will not be giving you control of aeroplanes, rockets or entire transport systems. If your program has an error, you should learn from it and try to fix the problem. You probably will not have killed anyone important.

Avoiding errors

When writing programs, make sure that you:

- *Make sure you understand the question* **before** *you start programming.*
- *Think about the best and most logical way to solve the problem.*
- *Consider planning your program on paper first.*
- *Put comments in your program so you know what you are trying to do.*
- *Test your programs on a range of data;*
- *Check some output carefully to make sure it is correct; and*
- *Pay attention to any error messages!*

A Python error message is shown in the following example:

Python Example 3.4.1

```
1 Traceback (most recent call last):
2   File "<pyshell#243>", line 1, in <module>
3     2/0
4 ZeroDivisionError: integer division or modulo by zero
```

The Python error message may be a bit confusing, but on Line 4 it clearly identifies the kind of error (in this case, trying to divide by zero). In general, the last line of a Python error message will tell you what kind of error occurred. This will help you to diagnose the problem.

Fear not!

Do not be afraid of error messages! Never let the fear of error messages stop you from playing around with Python and trying different commands. Getting an error message **does not** *mean that you will fail the course. If it helps you to figure out what you did wrong, then you have learned something!*
Equally important, do not ignore error messages. They give you useful advice about what is going wrong.

A common error message is:

```
SyntaxError:  invalid syntax
```

This message means that Python cannot make sense of your instruction. This might be because you have forgotten a bracket or accidentally used a symbol which does not mean anything in Python.

3.5 What did I do wrong?

Previously we found out how to use a Python error message to find out what *kind* of error occurred. Now we will look at how we can work out *where* in your program an error occurred.

Python Example 3.5.1

Here is the program:

```
from __future__ import division
from pylab import *

a = input('Tell me a number: ')
b = input('Tell me another number: ')

c = a + b
d = a * bb

print a, '+', b, '=', c
print a, 'x', b, '=', d
```

Here is the output from running the program:

```
>>>
Tell me a number: 8
Tell me another number: 7

Traceback (most recent call last):
  File "example.py", line 7, in <module>
    d = a * bb
NameError: name 'bb' is not defined
```

To help you to identify the error, you should:

1. Look at the last line of the error message to identify the kind of error. In this case, we see

   ```
   NameError:  name 'bb'is not defined
   ```

2. Look at the third last line of the error message (line 10 in the example above). This will tell you *where* the error was detected.

   ```
   File "example.py", line 9, in <module>
   ```

   This tells us the file and the line where the error occurred. You should look very carefully at this line of your program and try to see where you made a mistake. If you look at line 9 of the program in the example above, you will see that the programmer has accidentally typed '`bb`' instead of '`b`'.

To help you find a certain line number in a Python program, the editor window tells you in the bottom-right corner the number of the line where your cursor is currently located. You can also use **Edit → Go to Line** to move to a specific line.

If you know what line the error is in, but you cannot figure out exactly what the problem is, see if you can think of a way of rephrasing your instruction a bit. Try something a little bit different and see what it does. This can help you to diagnose the problem.

If a program contains more than one error, Python will display the message for the first one which it encounters. So after you have found and fixed an error, you may be given a different error message. *This is usually a good sign.* It often means that you have fixed the first error and can move on to fixing the next one.

Some common types of errors are:

Error	Explanation and possible causes
`SyntaxError`	The command is not understood by Python. Perhaps: • You have used incorrect bracket types (e.g. `( )` instead of `[ ]`) • You have forgotten a bracket • Your indentation is incorrect (wrong number of spaces at start of line)
`NameError`	There is no variable with the given name. Perhaps: • You have mistyped the name of a variable. • You have forgotten to set a starting value for a variable.
`ImportError`	A module to be imported does not exist. Perhaps: • You mistyped the name of the module to import.
`OverflowError`	The answer is too large or too small to calculate.
`ValueError`	One of the arguments you have given is not valid for this function.
`IndexError`	You have used an invalid index to an array or sequence.
`TypeError`	One of the arguments you have given is not the correct type—for instance, you may have put quotation marks around a number in a numerical calculation.

Some of these errors you will probably not see until later on. For example, an `IndexError` will not make much sense until you have learned about arrays. These errors are included in the table so that you can use this table as a reference later on.

4 Conditionals

Sometimes when programming, you want the computer to behave in a different way depending on whether a certain condition is true or false. For example, if you had written a program to simulate the metabolism of alcohol by the body, you might want to print a different message depending on the answer to the question "is the blood alcohol content greater than 0.05% after three hours?"

4.1 Boolean values

A *Boolean* value is a value which is either true or false, but cannot be both at the same time. Examples of statements which can be either true or false (but not both) are:

All frogs are green 4 is greater than 3 $x \leq y$, for given values of x and y.

Python uses the special values `True` and `False` to represent boolean values.

4.2 Evaluating conditions

Python has a number of operations which will always result in Boolean values when the operations are used on two numbers.

Operation	**Mathematical representation**	**Python representation**
Greater than	$a > b$	`a > b`
Less than	$a < b$	`a < b`
Greater than or equal	$a \geq b$	`a >= b`
Less than or equal	$a \leq b$	`a <= b`
Equal to	$a = b$	`a == b`
Not equal to	$a \neq b$	`a != b`

You can use these operations in Python in the same way that we used mathematical operations such as `+` or `-`.

Notice that the operator for checking whether two things are equal in Python is `==` and not just a single `=` sign. We have already seen that the single `=` sign is used to assign a value to a variable.

4.3 Combining conditions

Boolean values can also be combined using **and**, **or**, **not**, in the following ways:

- `x and y` is true if and only if both x is true and y is true.
- `x or y` which is true if x is true or y is true, or both x and y are true.
- `not x` which is true if and only if x is false.

The effects of these conditions are illustrated in the following Python program.

Python Example 4.3.1

Here is the program:

```
from __future__ import division
from pylab import *
x = 4
y = 5
print x>4
print x>=4
print x==4
print x==4 and y==5
print x==5 or y==4
```

Here is the output from running the program:

```
False
True
True
True
False
```

4.4 Conditional statements

Now that we can evaluate conditions, we can use them to write statements which are only performed if a given condition is true.

4.4.1 The `if` statement

Simple conditional statements are written in the following way:

```
if condition:
    actions
```

This statement will run `actions`, but only if `condition` is true. Note that the **indentation is important**. Python uses the indentation so it can tell which actions to perform only if the condition is met. If the condition is **not** true, the next command that runs is the **first** command **after** the indented lines.

The following program inputs a driver's blood alcohol content and prints a message if it is illegal to drive in Queensland.

Python Example 4.4.1

Here is the program:

```
from __future__ import division
from pylab import *

# This program inputs your blood alcohol content BAC
# and prints a message if you are over the legal limit.
```

```
6
7 BAC = input("What is your blood alcohol content? ")
8
9 if BAC >= 0.05:
10     print 'You are over the legal limit!'
```

Here is the output from running the program twice:

```
1 What is your blood alcohol content? 0.07
2 You are over the legal limit!
3
4 What is your blood alcohol content? 0.04
```

Note that in the second case there is no message printed, as it is legal to drive with a blood alcohol content of 0.04.

4.4.2 The `else` statement

In the previous example, the program outputs a message if it is illegal to drive, but gives no output if it legal to drive. The program would be more useful if it printed out a different message if it is legal to drive.

When programming, it is often the case that we have **two** possible situations, with a need to execute one or other of the cases depending on the value of some Boolean condition. This is done in the following way.

The following program inputs a driver's blood alcohol content and prints a message stating whether it is legal to drive in Queensland.

Python Example 4.4.2

Here is the program:

```
1 from __future__ import division
2 from pylab import *
3
4 # This program inputs your blood alcohol content BAC
5 # and prints a message indicating whether or not it is legal to drive.
6
7 BAC = input("What is your blood alcohol content? ")
8
9 if BAC >= 0.05:
10     print 'You are over the legal limit!'
11 else:
12     print 'You are legal to drive!'
```

Here is the output from running the program twice:

```
1 What is your blood alcohol content? 0.07
2 You are over the legal limit!
3
4 What is your blood alcohol content? 0.04
5 You are legal to drive!
```

If there are multiple different conditions to check then the following approach is used:

```
if condition1:
    action1
elif condition2:
    action2
elif condition3:
    action3
else:
    other_action
```

The command `elif` is short for "else if", and means that Python should do these actions if the previous conditions are not true, but this condition is. The `else` section is only run if none of the other conditions is true.

Once again, we can use this new form to extend our example.

Python Example 4.4.3

Here is the program:

```
from __future__ import division
from pylab import *

# This program inputs your blood alcohol content BAC and prints a message
# indicating whether or not it islegal to drive, or you have a BAC of 0.

BAC = input("What is your blood alcohol content? ")

if BAC >= 0.05:
    print 'You are over the legal limit!'
elif BAC == 0:
    print 'Well done; you have had nothing to drink.'
else:
    print 'You are legal to drive UNLESS you are on'
    print 'a Learner's permit or a Provisional licence.'
```

Here is the output from running the program three times:

```
What is your blood alcohol content? 0.07
You are over the legal limit!

What is your blood alcohol content? 0.04
You are legal to drive UNLESS you are on
a Learner's permit or a Provisional licence.

What is your blood alcohol content? 0
Well done; you have had nothing to drink.
```

5 Loops

5.1 Why loops?

In all of the programming which we have done so far, each of the lines of code in the program were executed once in the order in which they were written (except for lines inside an `if` statement). Solving problems often requires the program to execute a sequence of code multiple times.

For instance, suppose we were modelling population growth in a predator-prey system. We might write a few lines of programming to describe how the number of predators and number of prey interact over a year. If we wanted to simulate a population over 50 years, it would be inconvenient to have to repeat the same programming 50 times.

The programming concept which allows lines of code to execute multiple times in succession is called a *loop*.

5.2 While loops

Different programming languages have different kinds of loop. In this section we will focus on *while* loops in Python. A *while* loop in Python is constructed in the following way:

```
while condition:
    actions
```

Here `condition` is some Boolean condition (recall that this is an expression which can either be true or false) and `actions` is an indented sequence of instructions forming the loop *body*. The `condition` may include any of the boolean operations which we looked at in the last section. Once again, it is important that the loop body is indented.

When Python executes a *while* loop, it will:

1. Check whether `condition` is true.
2. If `condition` is not true, jump to step 4.
3. If `condition` is true, execute the `actions` in the loop body, then return to step 1.
4. Run the rest of the program, recommencing from the **first** line **after** the loop body.

Python Example 5.2.1

```
from __future__ import division
from pylab import *
# This program uses a loop to calculate and print square and cubic numbers.

i = 1
while i <= 5:
    print i, i*i, i*i*i
    i = i + 1
print 'Done.'
```

The output of this program is:

```
1 1 1
2 4 8
3 9 27
4 16 64
5 25 125
Done.
```

The events in the program are carried out in the following sequence:

- In Line 4, `i` is set to 1.
- Python tests the condition `i <= 5` (Line 5) and finds that it is true.
- Python executes the loop body (Lines 6 and 7), which:
  - Prints 1, 1^2 and 1^3.
  - Sets `i` to equal 2.
- Python tests the condition in Line 5 again and finds that `i <= 5` is still true.
- Python executes the loop body again, which:
  - Prints 2, 2^2 and 2^3.
  - Sets `i` to equal 3.
- Python tests the condition in Line 5 again.
- This process continues until `i` is 5.
- Python tests the condition in Line 5 and finds that `i <= 5` is still true.
- Python executes the loop body which:
  - Prints 5, 5^2 and 5^3.
  - Sets `i` to equal 6.
- Python tests the condition in Line 5 and finds that `i <= 5` is now **false**.
- The program resumes at the first line after the loop body (Line 8), which prints the word "Done.".

5.3 Loop forever...

A *while* loop will continue to run the commands in the loop body until the condition is no longer met. This means that you have to be very careful that you choose a condition which will cause the loop to stop at some stage.

Consider the following loop:

```
1 from __future__ import division
2 from pylab import *
3 x = 3
4 while x < 10:
5     print 'forever...'
```

Notice that there is nothing within the body of the loop that changes the value of `x`. This means that the condition `x < 10` will always be true, so the loop will never terminate. This is called an *infinite loop*. Take care to avoid infinite repetition when writing loops.

Stopping infinite loops

If you run a Python program and it seems to be taking a long time, it **may** *contain an infinite loop. If you suspect that a running program contains an infinite loop, you can terminate it by pressing* `Ctrl+C`.

5.4 Nested loops

Sometimes it may become necessary to have loops within loops. For instance, in a neuroscience model you might want to have one loop which goes through all the neurons in your model, and use this *within* a loop which steps through time to calculate how the system changes over time. This is called *nesting*—one loop is said to be *nested* within the other.

To accomplish this in Python, you can simply put one loop inside the body of another loop. Remember that the body of the **inner** loop must be **double** indented.

Python Example 5.4.1

```
1 # This program demonstrates nested loops.
2
3 a = 1
4 while a <= 3:
5     b = 1
6     while b <= a:
7         print 'a =', a, 'and b =', b
8         b = b + 1
9     a = a + 1
```

In this program:

- Line 4 contains the condition for the outer loop.
- Lines 5 to 9 form the body of the outer loop.
- Line 6 contains the condition for the inner loop.
- Lines 7 and 8 form the body of the inner loop.
- Line 9 is part of the body of the outer loop, but not part of the inner loop.

When the condition of the inner loop is false, the program will resume at the next line which is not part of the body of that loop: line 9. This *is* however part of the body of the outer loop.

This program generates the following output:

```
a = 1 and b = 1
a = 2 and b = 1
a = 2 and b = 2
a = 3 and b = 1
a = 3 and b = 2
a = 3 and b = 3
```

You can also nest conditional (`if`) statements inside loops and vice versa. By combining loops and conditionals, you can create very powerful algorithms to solve scientific problems.

6 Arrays

We have already used Python to store individual data values in variables such as `x` or `y`. Often we need to store many related items of data. In these cases, using an individual variable for each value can make things cumbersome.

Programming languages provide different mechanisms for storing lots of data. In general, a **data structure** is a way of storing and accessing data in a computer program.

A key component of good programming is choosing a data structure that is appropriate to the problem being solved. Choosing a good data structure can allow algorithms to run much more quickly or to make more efficient use of the available computer memory.

Probably the most common data structure is the *array*.

6.1 What is an array?

An array is a group of data items, and may be thought of as a list or table of data. The position of an entry in an array is given by one or more *indices*.

The **number of indices** needed to specify the positions of each element in an array is called the **dimension** of the array. For example in a one-dimensional array, each value in the array can be accessed using a single index, such as 1 or 5 or 16. The number of elements in an array is called the **size** of the array.

> **For those interested**— *In a two-dimensional array, you need two indices to specify the position of a single value in the array. For instance a certain position in the array may be indexed by the pair of values* $(2, 3)$*. Arrays with two or more dimensions are extremely important in programming, but we will not cover them in SCIE1000; we will only consider one-dimensional arrays.*

6.2 Printing arrays

To print the contents of an entire array, you can use the `print` command as we have seen before. Python will print the array on the screen in a neat format. For instance, suppose `A` is an array containing the three elements 10, 20, 30. If you were to write `print A` in a program, Python would print `[10 20 30]`. The square brackets around the entries indicate that the thing being printed is an array.

6.3 Creating arrays

There are two common ways to create arrays in Python. These are outlined below with examples of their use. Always remember that you need to import the module `pylab` before you can use arrays.

6.3.1 Creating an array from values

The `array` command is used to create an array containing a given list of values.

Creating arrays with a list of values

To create a one-dimensional array containing a list of values, use:

```
A = array([...])
```

where `...` *lists the* **values of the entries** *in the array, separated by commas. It is important that you remember to type the square brackets* `[ ]` *and round brackets* `( )` *as shown or the command will not work correctly.*

The following program illustrates this method of creating arrays.

Python Example 6.3.1

Here is the program:

```
# This example illustrates how to create an arrays from a list of values.

from __future__ import division
from pylab import *

# Create a new array with the given values.
A = array([2, 4, 6, 8, 10])
print A
```

The output from running this program is:

```
[ 2  4  6  8 10]
```

6.3.2 Creating an empty array

The `zeros` command is used to create an array of a given size.

Creating arrays of a given size

To create an array of a given size, with all entries equal to zero, use:

```
A = zeros(...)
```

where `...` *gives the* **size** *of the new array.*

This is illustrated in the following program:

Python Example 6.3.2

The program is:

```
# This example illustrates how to create an empty array.

from __future__ import division
from pylab import *
```

```
5
6 # Create a new array containing five 0s.
7 B = zeros(5)
8
9 # Create an array containing seven 0s.
10 C = zeros(7)
11
12 print B
13 print C
```

The output from running this program is:

```
1 [0 0 0 0 0]
2 [0 0 0 0 0 0 0]
```

6.4 Arrays and indices in Python

6.4.1 Accessing arrays

In Python, arrays are given names just like any other variables, and an entire array can be accessed by giving its name. To access **specific elements** in an array, you must give the name of the array, immediately followed by the appropriate index surrounded by square brackets.

For instance, if `A` is an array, then `A[i]` will give the value of the element at position `i`.

Accessing array elements

To access **specific elements** *in an array, you must give the name of the array, immediately followed by the appropriate index surrounded by square brackets.*

6.4.2 Array indices

The index of an element refers to the position of that element in the array. In Python, the **first** entry in an array has index `0`. This is potentially confusing, but it is important to remember. This is also true in many (but not all!) other computer languages.

Let `A` be an array which contains n entries. The valid values of the index are from 0 to $n-1$ (inclusive).

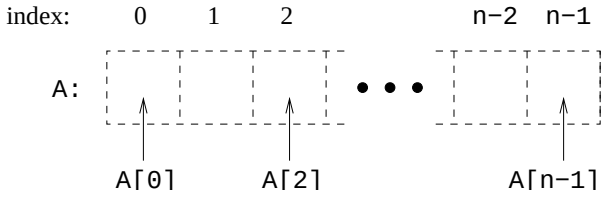


Figure: Indexing elements in a one-dimensional array A with n entries

Array indices

In Python, the **first** *element in an array has index* **0**.

The following example illustrates creating an array and assigning values to its elements.

Python Example 6.4.1

Here is the program:

```
from __future__ import division
from pylab import *

# First create an array called A with 5 entries.
A = zeros(5)
print 'Step one:', A

# Now assign the value 2 to the first element of A.
A[0] = 2
print 'Step two:', A

# Assign the value 10 to the last element of A.
A[4] = 10
print 'Step three:', A

# Access the values we have just assigned and save them in x.
x = A[0] + A[4]
print 'Step four, x =', x

# Assign the value 40 to A[1].
A[1] = 2 * A[4] * A[0]
print 'Step five:', A
```

The output from running this program is:

```
Step one: [0 0 0 0 0]
Step two: [2 0 0 0 0]
Step three: [ 2  0  0  0 10]
Step four, x = 12
Step five: [ 2 40  0  0 10]
```

After running the above program, the array `A` will look as follows:

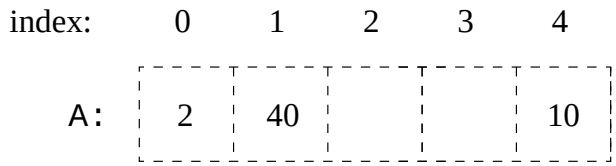


Figure: Assigning and accessing values in an array A with 5 entries

6.5 Copying an existing array

If `oldA` is an array and you want to copy it into a new array named `newA`, use:

```
newA = oldA.copy()
```

Do **not** use `newA = oldA` as this does not create a new copy. (Instead, both `oldA` and `newA` will refer to the **same** data.)

Python Example 6.5.1

```
# This program demonstrates copying an array.
from __future__ import division
from pylab import *

A = array([5,10,15])
B = A.copy()
print B
```

The output of this program is:

```
[ 5 10 15]
```

6.6 Operations on entire arrays

Many of the Python commands we have already seen can act element-by-element on entire arrays at once, producing new arrays as their output.

The operations `+ - * /` act on **two** arrays A and B **of the same size**, and the output is an array (of the same size) resulting from applying the given operation to corresponding pairs of elements from A and B.

The Python functions `sqrt`, `sin`, `cos`, `tan`, `exp`, `log` and `log10` all act on single arrays, with the output an array (of the same size) resulting by applying the operation to each element. For instance, consider look through the following program and its output:

Python Example 6.6.1

```
# This example program shows Python operating on entire arrays.
# First, create an array with 5 elements, then create a new
# array containing the squares of these elements.
A = array([5, 10, 5, 0, 20])
B = A * A

# Create a new array.
C = array([1, 4, 9, 16, 25, 36])

# Display combinations of these arrays.
print B
print A + B
print sqrt(C)
```

The output of this program is:

```
[ 25 100  25   0 400]
[ 30 110  30   0 420]
[ 1.  2.  3.  4.  5.  6.]
```

7 Graphs

7.1 Drawing a graph

One of the major uses of computers in modelling is data visualisation. The simplest type of visualisation is a graph. Graphs usually involve plotting a number of points, given their x- and y-coordinates. Arrays are useful for doing this in Python.

To draw a graph in Python, you need an array of x-coordinates and an array of y-coordinates. These arrays must be the same size. To plot a graph, we use the command `plot(A, B)`. Python pairs up the values in the two arrays to form points of (x, y), and then plots the points.

The command `show()` will display all plotted graphs on the screen. It can only be used once in a program, and should be the **last** command in the program.

Graphing in Python

Suppose A is an array containing the x-coordinates of n points, and B is an array containing the y-coordinates of the n points, **in the same order**. *To draw a graph of the points with lines joining the consecutive points, use the following commands.*

```
plot(A, B)
show()
```

Note that you must type the brackets in `show()` *or the command will not work as expected. The* `show()` *command should only be used once in a program, and should be the* **last** *command in your program.*

For those interested— *To draw multiple plots on the same graph, use the* `plot(...)` *command more than once.*

Python Example 7.1.1

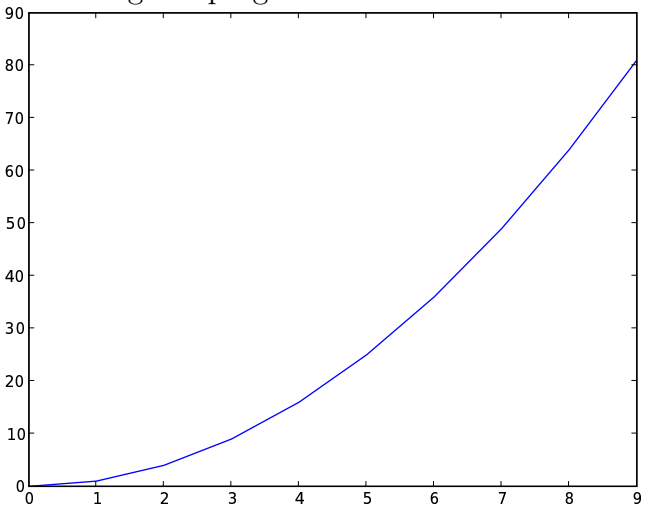
This example shows how to use Python to plot x^2 versus x for integer values of x from 0 to 9 inclusive.

```
# This program plots x**2 vs x for x ranging from 0 to 9 and
# connects the consecutive points.

from __future__ import division
from pylab import *
A = array([0,1,2,3,4,5,6,7,8,9])
B = A * A

plot(A, B)
show()
```

Here is the output from running the program:



7.2 Plotting separate points

Sometimes, instead of plotting lines or smooth curves connecting the data points as in the previous example, you may want to plot discrete points shown as crosses or circles, without lines or curves joining them.

Changing the style of plotting

To select the colour and style when plotting, type

```
plot(X, Y, 'style')
```

where `style` *is replaced by letters representing the colour and marker to use. Valid colours are* `b g r c m y k w`*. Valid marker styles are* `+ , o . s v x < > ^`*. To connect the dots, include the* `-` *character with the settings, for example* `plot(A,B,'ro-')`*, which plots the points in red, marked by circles and joined with a line.*

Python Example 7.2.1

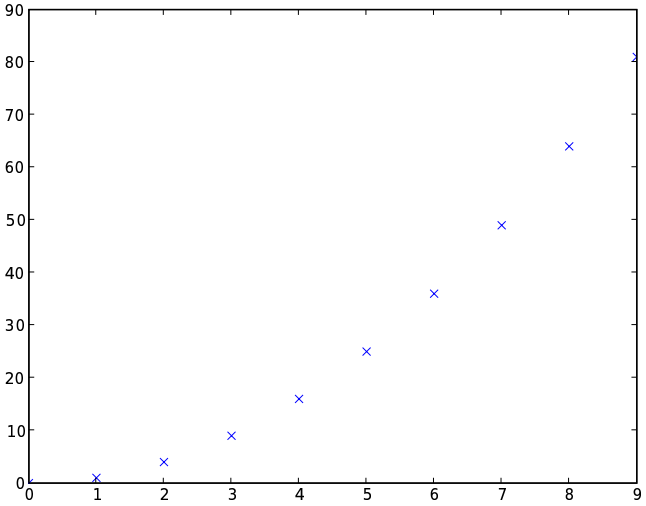
The following example shows how to plot a graph using crosses and with no line joining the points. It has done this by using `'bx'` to tell the `plot` command what style to use.

```
# This program plots x*2 vs x with x ranging from 0 to 9.
# The points are plotted discretely.

from __future__ import division
from pylab import *
A = array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
B = A * A

# Note the additional parameter 'bx' to the plot command.
# This indicates that blue crosses should be used.
plot(A,B,'bx')
show()
```

Here is the output from running the program:



7.3 Graphing mathematical functions

When drawing graphs, you most likely either want to graph:

- measured values (perhaps from an experiment); or
- a mathematical equation that the data must satisfy.

In the examples above, we showed how to plot measured data points.

If you want to plot a mathematical equation, you want the computer to draw a smooth curve. Computers cannot actually draw smooth curves: instead, they approximate smooth curves by drawing straight line segments joining points that are very close together. (If you look at the graph in Example 7.1.1 you may be able to see some straight line segments.) The more data points you have, the smoother your curve will look. The exact number of points that you will need will be different for different problems.

How to graph an equation

To plot a mathematical equation in Python, you first need to get the computer to calculate some points which lie on the curve. The usual approach is to:

- *Create an array with the x-coordinates of "appropriate" points.*
- *Create an array of the corresponding y-coordinates by applying the equation to each x-coordinate.*
- *Plot the graph.*
- *If the graph is not sufficiently smooth, edit your program so that it uses more data points.*

Suppose you are attempting to plot a mathematical equation. Let X represent an array of x-coordinates, and Y be the array of corresponding y-coordinates.

Once you have created X you can easily create Y by applying the function to X. In order to create X you could type each of the x-coordinates, but this takes a long time, especially if you need a lot of data points to make the graph look smooth.

When plotting graphs on computers, it is very common to choose points whose x-coordinates are *equally spaced*. This means that the difference between the x-coordinates of consecutive points is a constant, and this distance is chosen to be sufficiently small for the plots to look smooth. Python has a command `arange` to easily create an array of equally spaced points.

Equally spaced values in Python

To create an array X of equally spaced values in Python, use this command:

```
X = arange(a,b,s)
```

This creates an array of points with values starting at `a`*, increasing by an equally spaced step of* `s` *each time, and stopping at the* **last value less than** *b.*

Take care here: *the final value in the array is* **always strictly less than** `b`*. So, for example,*

```
X = arange(0,1,0.2)
```

does not include the value 1, so

```
X=array([0.,0.2,0.4,0.6,0.8]).
```

The command `arange` is very useful for graphing. If the graph is not sufficiently smooth then use `arange` to create more points.

Python Example 7.3.1

This example demonstrates using the `arange` function to help with plotting $\sin x$ versus x for values of x from 0 to 5.

```
# This program uses arange() to help plot sin(x) vs x for
# values of x ranging from 0 to 5.

from __future__ import division
from pylab import *

# Plot with the x-coordinates separated by 0.5.
X = arange(0.0, 5.1, 0.5)
B = sin(X)
plot(X, B)
show()
```

Upon examining the graph produced by this program (shown at the left, on the next page), we might decide that there are too few points. In an effort to produce a smoother graph, we might modify the program like so:

```
# This program uses arange() to help plot sin(x) vs x for
# values of x ranging from 0 to 5.
```

```
3
4 from __future__ import division
5 from pylab import *
6
7 # Plot with x-coordinates separated by .1 to give a better graph.
8 X = arange(0.0, 5.1, 0.1)
9 B = sin(X)
10 plot(X, B)
11 show()
```

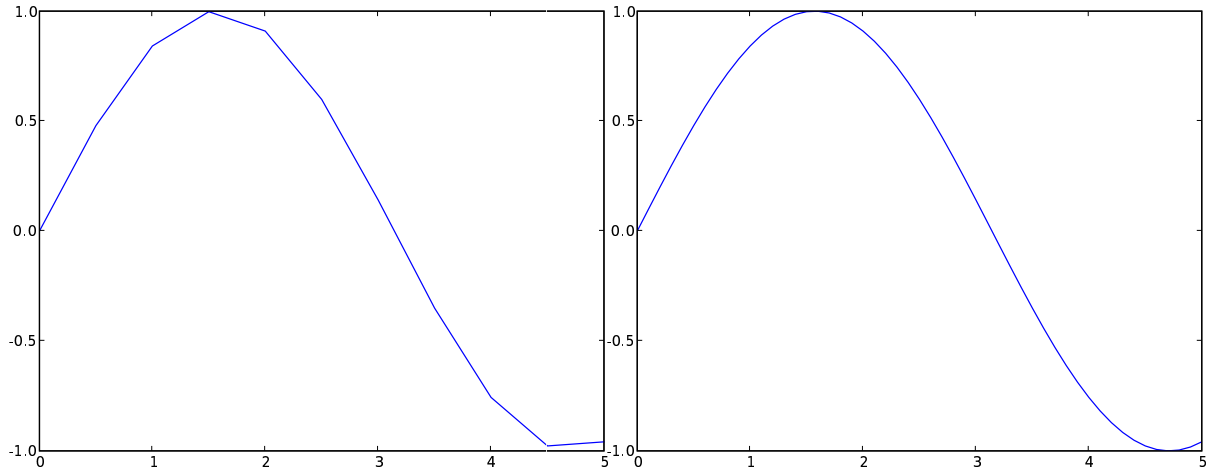


Figure: The results of the two programs above. The first is shown on the left, the second on the right.

7.4 Titles and axes

On any scientific graph it is important to title the graph and label the axes. This can be done in Python using the following commands:

- `title('Graph Title')` – sets the graph title to 'Graph Title'.
- `xlabel('Time (seconds)')` – sets the label on the horizontal axis to say 'Time (seconds)'.
- `ylabel('Heart rate (bpm)')` – sets the label on the vertical axis to say 'Heart rate (bpm)'.

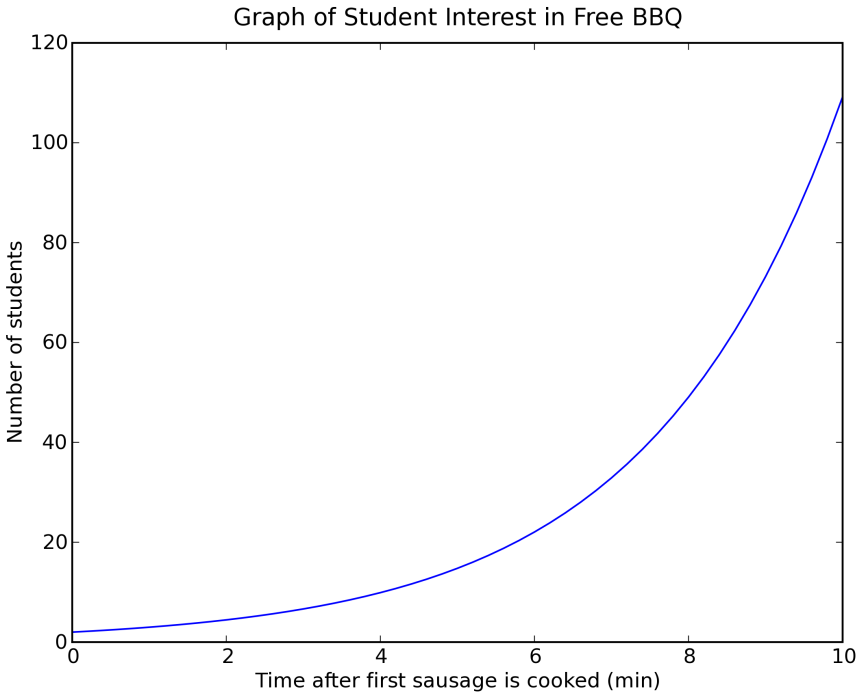
For those interested— *There are also other arguments you can use with the* `title`, `xlabel` *and* `ylabel` *commands. These can be used to change colour, font size, text alignment and other attributes of the title or labels. Information on the pylab plotting functions can be found at* `http://matplotlib.sourceforge.net/tutorial.html`.

Python Example 7.4.1

```
1 # This program uses the title(), xlabel() and ylabel()
2 # commands to demonstrate the use of titles and labels
3 # on graphs.
```

```
4
5  from __future__ import division
6  from pylab import *
7
8  # First we will create an array using arange.
9  x = arange(0.0, 10.1, 0.2)
10
11 # Now we will enter the function to plot.
12 y = 2 * exp(0.4 * x)
13
14 # Draw the graph with title and labels.
15 plot(x, y)
16 title('Graph of Student Interest in Free BBQ')
17 xlabel('Time after first sausage is cooked (min)')
18 ylabel('Number of students')
19
20 # Show the graph.
21 show()
```

The result of this program is:



8 Software design and functions

8.1 Procedural abstraction

When writing programs, you will typically need to use the same kind of computation over and over again. Instead of writing this code out every time you need to use it, a better approach is to write a piece of code (a *function* in Python) that encapsulates the computation in a way that can be reused by other programs.

Every Python function has a name, and is referred to by this name. When you are writing functions, make sure that you choose **meaningful** names so it is clear from the name what the function actually does.

So far we have seen some predefined functions including `sqrt`, `sin` and `log`. These functions have simply taken input arguments and calculated some output from them. A Python function is not restricted to performing mathematical calculations. It can do anything that a program does (such as printing messages, creating arrays and drawing graphs).

Working with functions involves two related but distinct activities:

- *creating* (or *defining*) the function. You *define* a function by giving it a name and specifying the Python commands that actually do what the function requires. The function does not run at the time you define it.
- *using* (or *calling*) the function. You *call* the function by typing its name as one of your Python commands. This will run the function.

8.2 Defining functions

8.2.1 How to define a function

Function definitions have the following form:

```
1 def function-name(function-args):
2     body
```

Here:

- `function-name` is used to identify the function. Anything which is valid as a variable name is also a valid function name. (But do not give a function and a variable the same name in a program!) Refer back to section 3.1.1 if you need refreshing on what makes a valid variable name.
- `function-args` is a list of arguments for the function. This is explained in the next section.
- The word `def` and the colon `:` in the first line are essential!
- `body` is the sequence of Python commands that are executed when the function is . All commands in the body must be indented by a consistent amount so that Python knows that they belong to the function. In all the examples in this course, we **use four spaces for indentation**.

Python Example 8.2.1

This is a simple example of defining Python function. The individual elements of this definition will be explained in more detail later; at the moment, just note how it is defined, and that the title and comment make it quite clear what the function does.

```
def double(n):
#   This function returns 2 times the input.
    return 2 * n
```

8.2.2 Arguments

Functions often need to operate on one or more input values. This allows the function to be general and be used for many different sets of values. The input values for a function are called *arguments* (or sometimes *parameters*).

For example, to find the sine of 3 radians, you would use the Python command `sin(3)`. The value 3 is *passed in* to the function `sin`. The programmer who originally wrote the `sin` function would have specified that the function had one argument.

When defining a function, you must specify what variables will correspond to the arguments that are passed into the function. These are sometimes the *formal arguments* of the function. These variables must be included as a list of names separated by commas. The following examples illustrate this. Pay attention to the first line of each example.

To define a function with no arguments:

```
def myFunction():
#   This functions prints a friendly message.
    print 'Hello world!'
```

To define a function with one argument:

```
def double(n):
#   This function returns 2 times the input.
    return 2 * n
```

To define a function with more than one argument:

```
def CO2Emission(electricity, gas, petrol):
#   This function calculates the CO2 emission for given resource usage.

    elecCO2 = 1 * electricity
    gasCO2 = 0.4 * gas
    petrolCO2 = 2.2 * petrol

    return elecCO2 + gasCO2 + petrolCO2
```

8.2.3 The function body

The function body contains all the instructions which Python should carry out when it runs the function. Consider the function which we looked at earlier:

```
def double(n):
#   This function returns 2 times the input.
    return 2 * n
```

First, notice that the function includes a comment at the top, explaining what the function does. Next, notice that the line in the function body is indented by four spaces. Just as with the lines inside loops, Python uses indentation to tell it that the lines belong to the function. Finally, notice that the last line of the function contains the command `return`. This tells Python what output value the function should return to whoever ran the function.

8.3 Calling functions

Once you have defined a function, you can *call* (or *run*) the function in the same way that you would use a mathematical function like `sin`, `exp` or `log`. When you call a function, you must provide values for the same number of arguments that the function was defined with.

Python Example 8.3.1

This program shows how to call the three previous functions.

```
from __future__ import division
from pylab import *

myFunction()
x = double(3)
print 'The value of x is',x
emi = CO2Emission(140,50,40)
print 'Your emissions are ',emi
```

Here is the output from running the program:

```
Hello world!
The value of x is 6
Your emissions are 248
```

8.4 Some notes on functions

Note that:

- Variables that are given values **inside** a function are **not accessible** outside the function, even if you use the same name inside and outside the function.
- Some functions **return** a value, but functions do not have to do so.
- You can call other functions from within a function.
- You can define multiple functions inside the same file. Remember that indenting shows where the body of each function starts and ends.

A key skill in computer programming is deciding when it is appropriate to use functions. The main reasons to use functions are:

- **clarity**—if you have lots of short, well-designed functions with meaningful names, your programs will be easy to understand and therefore easy to debug and maintain.
- **avoid repetition**—if you need the same calculations to be performed in multiple places, using functions will avoid you having to continually rewrite the same code.

The following example illustrates some of the points outlined above.

Python Example 8.4.1

```
1  def message():
2  #   This function prints the rule for converting from Fahrenheit to Celsius.
3      print 'To convert from Fahrenheit to Celsius,'
4      print 'subtract 32 and divide by 1.8'
5      return
6
7  def FtoC(fahrenheit):
8  #   This function returns the given value converted from Fahrenheit to Celsius.
9      message()
10     Celsius = (fahrenheit - 32) / 1.8
11     return Celsius
12
13 # Main program
14 tempF = input('What is the Fahrenheit temperature? ')
15 tempC = FtoC(tempF)
16 print tempF,' degrees Fahrenheit equals ',tempC,' Celsius.'
```

Note that:

- In Line 1, there is the definition of a function `message` with no arguments. The body of this function occurs in Lines 2 to 5.
- In Line 5, the function `message` returns from running, but does not return a value.
- In Line 7, there is the definition of another function `FtoC` with one argument. The body of this function occurs in Lines 8 to 11.
- In Line 9, `FtoC` calls `message`, with no arguments. Also, because `message` does not return a value, `FtoC` does not make use of a returned value.
- In Line 10, `FtoC` does a temporary calculation, and in Line 11, `FtoC` returns a value.
- Line 15 represents the start of the main program.

Here is an example of running the program.

```
1 What is the Fahrenheit temperature? 32
2 To convert from Fahrenheit to Celsius,
3 subtract 32 and divide by 1.8
4 32 degrees Fahrenheit equals 0 Celsius.
```