PHYC 2050 Class Notes

Thomas J. Duck

Department of Physics and Atmospheric Science

Dalhousie University

tom.duck@dal.ca

February 1, 2010

CHAPTER

ONE

INTRODUCTION TO PROGRAMMING

A *program* is a list of *instructions*, or *commands*, for a computer. The human-readable form of a program is called *source code*, and is written in a *programming language* that has an agreed-upon set of rules called *syntax* (or *grammar*) . Programming languages that are in wide use include python, perl, php, ruby, java, javascript, C, C++, fortran, and assembly.

Source code may be *compiled* into binary computer instructions or *interpreted* for execution by another program. If one source code instruction translates into many binary computer instructions, then that language is said to be *high level*. Low-level languages have a nearly 1:1 translation ratio. Although it is easier and faster to write code in a high-level language, the execution time is usually longer than for low-level code (see Figure).

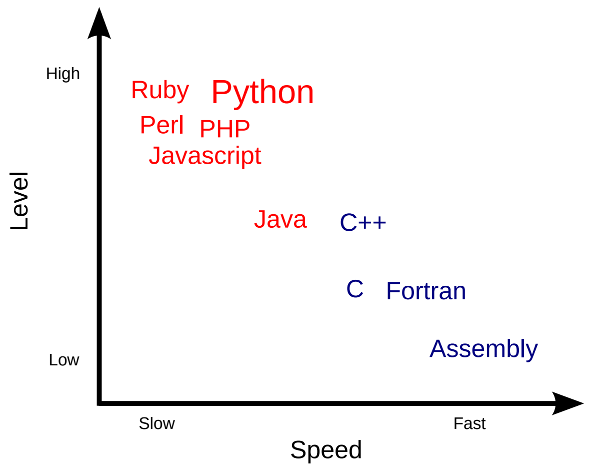

Figure 1.1: Programming language groupings according to execution speed and level. Interpreted languages are in red, and compiled languages are in blue. The precise location of each language in the plot shouldn't be taken too seriously.

There are many other criteria that can be used to differentiate programming languages. Each language has its own niche, and one should always "use the right tool for the job". A list of programming languages sorted into typical application categories is as follows:

- systems programming: C, C++, assembly
- desktop applications: C, C++, java
- web server programming: python, perl, php, java, ruby
- browser scripting: php, javascript
- computations: fortran, C, C++, python

Python (http://www.python.org/) is a high-level interpreted language with computational abilities that are implemented in fast low-level languages. It has an interactive interpreter, and is a good choice for our needs.

1.1 Python as a Scientific Calculator

Python can be used as a sophisticated scientific calculator. Our objective in this section is to plot the sinc function, a special function important for fourier transform and signal processing theory. The sinc function is defined by

$$sinc(x) = \frac{\sin(\pi x)}{\pi x}$$

and looks like:

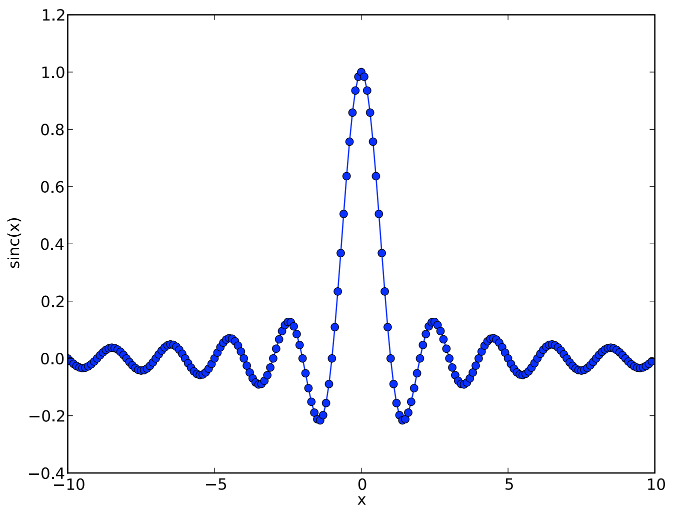

Figure 1.2: The sinc function. Notice that $sinc(0) = 1$.

Plotting the sinc function will require us to learn some python commands. We can experiment with python by running the interactive python interpreter in the bash [1] shell:

```
$ python
>>>
```

The “>>>” is the python prompt, and it indicates that python is prepared for you to start issuing instructions. The instructions must obey the syntax of the python programming language, which we will learn as needed in this and subsequent chapters.

1.1.1 Arithmetic and Numerical Types

Let’s try some basic arithmetic by entering *expressions* at the prompt:

```
>>> 7 + 2
9
```

[1] If you are new to Unix-like systems such as Linux or Mac OS X, you can access the bash command-line prompt from a Terminal program. On Windows python is run as an application.

An *expression* is a command that returns a value. The value returned when the expression "7+2" is evaluated is 9. So far so good!

When python is ready to accept more input, a new prompt is provided:

```
>>> 7 - 2
5
>>> 7 * 2
14
```

Be careful with integer division, which neglects the remainder and returns an integer result:

```
>>> 7 / 2
3
```

The expected answer is obtained if we use floating-point numbers [2] instead:

```
>>> 7.0 / 2.0
3.5
```

We can determine the *type* of a number as follows:

```
>>> type(7)
<type 'int'>
>>> type(7.0)
<type 'float'>
```

You can see that 7 is an 'int' (i.e., an integer) and 7.0 is a 'float' (i.e., a floating-point number). Different types of numbers behave in different ways, as seen above. We can also test the type of the value returned by an expression:

```
>>> type(7/2)
<type 'int'>
```

The expression 7/2 is evaluated first, and then the type of the result is determined. Programming languages employ many different types, and we will introduce them as needed.

Numerical comparisons can also be made. For example:

```
>>> 7 > 2
True
```

Similar to what we did with numbers, we can determine the type of the value `True`:

```
>>> type(True)
<type 'bool'>
```

The type is `bool`, which is short for "boolean". There are exactly two different boolean values, `True` and `False`. We will see how to make use of boolean values in conditional statements later.

The most-used *binary arithmetic operators* are:

[2] Floating point numbers are numbers which have a radix point. For decimal numbers we would call it the decimal point. We will be considering numbers using other bases (e.g., binary, octal, and hexadecimal numbers), hence the more general terms "floating point numbers" and "radix point".

Symbol	Meaning
+	add
-	subtract
*	multiply
/	divide
//	floor divide
%	remainder
**	power
<	less-than
>	greater-than
<=	less-than or equal-to
==	equals
!=	not equal-to

Be careful with equality comparisons:

```
>>> 7=2
  File "<stdin>", line 1
SyntaxError: can't assign to literal
```

The error message indicates that we broke python's syntax rules The '=' sign is used for *assignment*, as we shall soon see.

Round brackets () may be used to create compound arithmetic expressions:

```
>>> (7 + 2) ** 2
81
```

The standard order of operations is respected. We cannot use [] and {} for arithmetic. They are used for other things in python.

The keywords `and` and *or* can be used to create compound boolean expressions. For example,

```
>>> 7>2 and 7<10
True
>>> 7>2 and 10<7
False
>>> 7>2 or 10<7
True
```

One last example: Let's try computing 10^{-8}:

```
>>> 10**(-8)
1e-8
```

You will notice that the number is written using scientific notation, which you should read as 1×10^8.

1.1.2 Functions and Modules

A python function, like in mathematics, operates on zero or more *arguments* and returns a value. We have already seen one function, `type()`, that operates on a single argument.

```
>>> type(7)
<type 'int'>
```

The `type()` function is *built in* to the python language. Python has a variety of built-in functions (see http://docs.python.org/library/functions.html). We will mostly introduce them as they come up. Below are most of the mathematical ones, and you can try them right away. The descriptions for each function have been simplified; see the python documentation for the full details.

abs (*x*)
: Returns the absolute value of the number x.

bool (*x*)
: Convert a value x to a boolean: True if x>=1, False otherwise.

divmod (*a, b*)
: Returns the quotient and remainder of a/b when using long division.

float (*x*)
: Converts x to a floating-point number.

int (*x, [base]*)
: Converts x to a plain integer. The optional base argument gives the base for the conversion (10 by default).

max (*a, b*)
: Returns the maximum of a and b.

min (*a, b*)
: Returns the minimum of a and b.

round (*x, [n]*)
: Rounds the the floating-point value x. The optional argument n specifies the number of digits after the radix point to consider (default 0).

Some capabilities we will need are not built-in, but are instead contained in separate *modules* [3]. For example, more sophisticated mathematical functions and types are contained in the `numpy` (numerical python) module [4]. `numpy` is activated by using an import *statement*:

```
>>> import numpy
```

A *statement* is a command that does not return a value; hence, there is no "reply" from python after the "import numpy" statement is executed. Module *members*, such as the `sin()` function, are evaluated as follows:

```
>>> numpy.sin(0)
0.0
```

`numpy` functions expect arguments in radians rather than degrees. For example, we know that $sin(90°) = 1$, but attempting this in python gives the wrong answer:

```
>>> numpy.sin(90)
0.89399666360055785
```

Using radians returns the expected result:

```
>>> numpy.sin(3.1415926535/2)
1.0
```

`numpy` has a function that can help with degree-radians conversions:

[3] Python is distributed with a tremendous variety of standard modules. Documentation on the python Libarary is available from http://docs.python.org/.

[4] The `numpy` module is part of the "SciPy" (Scientific Python) project at http://www.scipy.org/.

```
>>> numpy.radians(90)
1.5707963267948966
```

`numpy.radians()` returns a value that can be used as an argument to `numpy.sin()`:

```
>>> numpy.sin( numpy.radians(90) )
1.0
```

Documentation on `numpy` can be found at http://numpy.scipy.org/. Look in particular at the the categorized list of numpy functions, with examples, found at http://www.scipy.org/Numpy_Functions_by_Category.

1.1.3 Variables

We can *assign* value (perhaps resulting from an expression) to a *variable* using an *assignment statement*. For example:

```
>>> angle = numpy.radians(90)
```

The variable angle can be used in subsequent calculations:

```
>>> numpy.sin( angle )
1.0
```

Just like in mathematics, this is a helpful way to manage complexity and save information for later use. A single equals sign (=) is used for assignment, and a double equals sign (==) is used for comparison. Don't confuse the two!

Variable names must begin with a character, and can contain both uppercase, lowercase, numbers, and the underscore character (_). Nothing else! Also, the following are reserved words in python, so you can't use them as variables:

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

So, `x`, `v0`, and `my_favourite_number` are valid variable names, but `1st_name` and `my favourite number` are not.

The value assigned to a variable can be displayed using the print statement. For example:

```
>>> x = 1
>>> print x
1
```

The value assigned to a variable may be changed during the execution of a program – that's why we call it a variable! Continuing on from the example above, we could change the value of `x` and display it by writing:

```
>>> x = 2
>>> print x
2
```

We can also follow with something strange like:

```
>>> x = x + 1
>>> print x
3
```

Writing $x = x + 1$ would get you kicked out of any math class. However, in computer programming, the equals sign does not imply equivalence. It is the assignment operator, and the value of the expression on the right-hand-side is assigned to the variable name given on the left. So in the above with the variable x originally 2, the expression $x + 1$ evaluates to 3, and this value is (re)assigned to the variable x.

Finally, modules often have variables defined in them. They are usually called *constants* instead, because you are not meant to change them. For example, the `numpy` module defines the key constants pi and e.

```
>>> numpy.pi
3.1415926535897931
>>> numpy.e
2.7182818284590451
```

1.1.4 Arrays

To plot a sinc curve we will need a handful of x and $sin()$ values. For example:

```
>>> x0 = -10
>>> y0 = numpy.sin(x0)
>>> x1 = numpy.radians(-9.9)
>>> y1 = numpy.sin(x1)
>>> x2 = numpy.radians(-9.8)
>>> y2 = numpy.sin(x2)
```

This is very tedious, and it would take a long time to get from -10 to +10 this way. What we need is a way to create a series, or *array*, of values in one command. An ordered array of ten integers is constructed using the `numpy.arange()` function as follows:

```
>>> x = numpy.arange(10)
>>> print x
[0 1 2 3 4 5 6 7 8 9]
```

Notice that the ten integers start at 0 and don't reach the number 10.

More generally, we can provide the start, end, and step numbers to `numpy.arange()`:

```
>>> x = numpy.arange(-10,10.5,0.5)
>>> print x
[-10.   -9.5  -9.   -8.5  -8.   -7.5  -7.   -6.5  -6.   -5.5  -5.   -4.5
  -4.   -3.5  -3.   -2.5  -2.   -1.5  -1.   -0.5   0.    0.5   1.    1.5
   2.    2.5   3.    3.5   4.    4.5   5.    5.5   6.    6.5   7.    7.5
   8.    8.5   9.    9.5  10. ]
```

The numbers spill onto multiple lines because there are so many of them. Notice that +10 is included in the sequence because we defined the stop limit to be 10.5 and the step size to be 0.5. This behaviour is consistent with what we saw for integers earlier.

The type of the data assigned to variable `x` can be determined as usual:

```
>>> type(x)
<type 'numpy.ndarray'>
```

The type this time is `ndarray` (for multi-dimensional array), a new type for us that is defined in the numpy module. Arrays are *container types* that hold a collection of *elements*. In the example above, the elements are all numbers, but it doesn't have to be so. They may hold anything in python that can be referred to as an *object*. An object is any entity in a program that has a type. For example, both arrays and numbers are objects, but operators are not. [5]

We can determine how many elements are in an array by using the built-in `len()` function:

```
>>> len(x)
41
```

The len function can be used with many different types of objects. Use it where it seems appropriate.

Arrays can be used in computations in much the same way as numbers. You can multiply an array by a number

```
>>> x = numpy.arange(5)
>>> print x
[0 1 2 3 4]
>>> print 2*x
[0 2 4 6 8]
```

or even multiply two arrays together

```
>>> y = numpy.arange(1,6)
>>> print x
[0 1 2 3 4]
>>> print y
[1 2 3 4 5]
>>> print x*y
[ 0  2  6 12 20]
```

so long as the arrays both contain the same number of values. Notice that the multiplication is done element-wise; i.e.

$$[0,1,2,3,4] \times [1,2,3,4,5] = [0 \times 1, 1 \times 2, 2 \times 3, 3 \times 4, 4 \times 5] = [0,2,6,12,20]$$

To create an array of sine values corresponding to `x` values between -10 and +10, we do the following:

```
>>> x = numpy.arange(-10,10.5,0.5)
>>> y = numpy.sin(x)
>>> print y
[ 0.54402111  0.07515112 -0.41211849 -0.79848711 -0.98935825 -0.93799998
 -0.6569866  -0.21511999  0.2794155   0.70554033  0.95892427  0.97753012
  0.7568025   0.35078323 -0.14112001 -0.59847214 -0.90929743 -0.99749499
 -0.84147098 -0.47942554  0.          0.47942554  0.84147098  0.99749499
  0.90929743  0.59847214  0.14112001 -0.35078323 -0.7568025  -0.97753012
 -0.95892427 -0.70554033 -0.2794155   0.21511999  0.6569866   0.93799998
  0.98935825  0.79848711  0.41211849 -0.07515112 -0.54402111]
```

There are as many sine values as `x` values:

```
>>> len(x)
41
```

[5] Curiously enough, in python functions are objects too. They may be manipulated like any other object, and so are often described as "first class functions".

The sine values oscillate between -1 and 1... almost. Our x values are never exactly equal to $\pm\pi/2$, $\pm 3\pi/2$, etc, and so the sine values never hit -1 or 1. Data resolution is an issue we must always be concerned about in numeric computations.

Numpy has all the trigonometric (and hyperbolic trig) functions you would expect: `sin()`, `cos()`, `tan()`, `arcsin()`, `arccos()`, and `arctan()` (and `sinh()`, `cosh()`, `tanh()`, `arcsinh()`, `arccosh()`, and `arctanh()`). It also has the exponential function `exp()`, the natural logarithm `log()`, the base-10 logarithm `log10()`, the base-2 logarithm `log2()`.

A huge number of other special functions are contained in the `scipy.special`. Here, `scipy` is called *package*, and `special` is one of the modules it contains. You can import and use it the same as before. For example, to calculate the Bessel function of order zero:

```
>>> import scipy.special
>>> x = numpy.arange(0.,10.,0.1)
>>> print scipy.special.j0(x)
[ 1.          0.99750156  0.99002497  0.97762625  0.96039823  0.93846981
  0.91200486  0.88120089  0.84628735  0.8075238   0.76519769  0.71962202
  0.67113274  0.62008599  0.56685512  0.51182767  0.45540217  0.39798486
  0.33998641  0.28181856  0.22389078  0.16660698  0.11036227  0.05553978
  0.00250768 -0.04838378 -0.09680495 -0.14244937 -0.18503603 -0.22431155
 -0.26005195 -0.29206435 -0.32018817 -0.34429626 -0.3642956  -0.38012774
 -0.39176898 -0.3992302  -0.40255641 -0.40182601 -0.39714981 -0.38866968
 -0.37655705 -0.36101112 -0.34225679 -0.32054251 -0.29613782 -0.26933079
 -0.24042533 -0.20973833 -0.17759677 -0.14433475 -0.11029044 -0.07580311
 -0.0412101  -0.00684387  0.02697088  0.05992001  0.09170257  0.12203335
  0.15064526  0.17729142  0.20174722  0.22381201  0.2433106   0.26009461
  0.27404336  0.28506474  0.2930956   0.29810204  0.30007927  0.29905138
  0.29507069  0.28821695  0.27859623  0.26633966  0.25160183  0.23455914
  0.21540781  0.19436184  0.17165081  0.14751745  0.1222153   0.0960061
  0.06915726  0.04193925  0.01462299 -0.01252273 -0.0392338  -0.06525325
 -0.09033361 -0.11423923 -0.13674837 -0.15765519 -0.17677157 -0.19392875
 -0.20897872 -0.22179548 -0.23227603 -0.24034111]
```

1.1.5 Function Definition and Conditional Statements

What if the function we need is not provided by either `numpy` or `scipy`? In this case we can define our own functions. For example, let's implement the sinc function, [6]

$$sinc(x) = \frac{\sin(\pi x)}{\pi x}$$

To begin, let's import what we need from numpy, but in a different way:

```
>>> from numpy import pi, sin
```

This directly imports both the constant `pi` and the function `sin()`. We don't need to keep writing `numpy` in front of everything any more. This isn't always a good idea, but here we will work with it. Continuing, our first implementation of the sinc function is

```
>>> def sinc(x):
...     "Returns sin(pi x)/(pi x)"
...     answer = sin(pi*x)/(pi*x)
```

[6] The `sinc()` function is provided by `numpy`. However, it is very instructive to implement our own version of it here.

```
...     return answer
...
>>>
```

On the first line, the *keyword* `def` indicates that we are defining a function, and the name of that function is `sinc()`. The function accepts one argument, and inside the function that argument will be assigned to the variable name `x`. This variable `x` exists only within the *scope* of the function. Other variables named `x` may exist outside the function, and they are not affected. The colon at the end indicates that what comes after gives the function definition.

Following the define statement, the prompt changes. The new prompt is python's way of saying that more code is needed to complete the statement. We follow with a *block* of code to *implement* the function. All lines in that block of code must be indented the same way. The standard is four spaces.

The second line of the sinc function is a *documentation string* that describes what the function is for. A `string` is a new type for us, and it provides a container for a sequence of characters. Single quotes and double quotes can be used to enclose strings on a single line. Triplicate quoting may be used for multi-line strings.

The function's documentation is displayed when we use the built-in `help()` function on it. e.g., try

```
>>> help(sinc)
```

and you will see something like:

```
Help on function sinc in module __main__:

sinc(x)
    Returns sin(pi x)/(pi x)
```

You can use the built-in `help()` function on just about anything in python. Whether or not the help is useful or not will depend on whether or not the function author took care enough to write good documentation (you should always take such care!). As an example, try

```
>>> help(numpy.lib)
```

to retrieve documentation for all of the basic functions in numpy (you may have to scroll down a fair ways to get to the good stuff). Some of the functions are not even documented on the Web pages!

Continuing on, the first line of the function implementation calculates the sinc function, and assigns it to the variable named `answer`. This is followed by a *return statement*, which indicates the value that the function should return. You may provide zero or more comma-separated values after the return keyword. If no values are to be returned, then you may skip the return statement.

You must leave one blank line after the function implementation to indicate to the interpreter that you have finished with it. Here is an example of the function in use:

```
>>> sinc(1)
3.8980430910514779e-17
```

That's very close to zero, which agrees with our expectations since $\sin(\pi) = 0$. The computation is not exact because numbers can often not be precisely represented on a computer. We will learn more about this issue later.

Let's try another value:

```
>>> sinc(0)
nan
```

This is a problem. A return value of `nan` means "not a number", but we expected 1. The error occurs because we tried to divide by zero. This is a special case that we must watch for and treat separately. Here's how we can adjust the function implementation:

```
>>> def sinc(x):
...     if x==0:
...         return 1.
...     else:
...         return numpy.sin(x)/x
...
>>>
```

We have introduced conditional statements, i.e., some logic, to the function implementation. The if-else statement provides *flow control* to our function: if `x` is zero it does one thing (as defined by an indented block of code), otherwise something else is done (as defined by a second block of code). The expression `x==0` is called the *condition*. The condition can be any (compound) expression that returns a boolean value. If the boolean value is `True`, then the block of code below the if clause is executed. Otherwise, the block of code below the else clause is executed.

Does it now work as expected?

```
>>> sinc(1)
3.8980430910514779e-17
>>> sinc(0)
1.0
```

Yes. Will it work for an array of `x` values?

```
>>> x = numpy.arange(-10,10.5,0.5)
>>> sinc(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in sinc
ValueError: The truth value of an array with more than one element
is ambiguous. Use a.any() or a.all()
```

No. Python is complaining because the result of the condition x==0 is not a boolean. e.g.,

```
>>> x = numpy.arange(-10,10.1,0.5)
>>> print x
[-10.   -9.5  -9.   -8.5  -8.   -7.5  -7.   -6.5  -6.   -5.5  -5.   -4.5
  -4.   -3.5  -3.   -2.5  -2.   -1.5  -1.   -0.5   0.    0.5   1.    1.5
   2.    2.5   3.    3.5   4.    4.5   5.    5.5   6.    6.5   7.    7.5
   8.    8.5   9.    9.5  10. ]
>>> x==0
array([False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False,  True, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False], dtype=bool)
>>> type(x==0)
<type 'numpy.ndarray'>
```

Conditional statements can act on a single boolean value, but we have provided an array of boolean values, so python doesn't know how to proceed. So, you can see we have to ensure that the instructions we provide in a program are very clear.

Here is what to do (assuming we have imported `numpy`, and both `pi` and `sin` from it):

```
>>> def sinc(x):
...     if numpy.isscalar(x):
...         if x==0:
...             return 1.
...         else:
...             return sin(pi*x)/(pi*x)
...     else:
...         return numpy.where(x==0,1,sin(pi*x)/(pi*x))
...
>>>
```

You can see above that we have used nested conditional statements. Python won't compain so long as we indent the blocks of code appropriately.

We have also used a few new numpy functions, `numpy.isscalar()` and `numpy.where()`. `numpy.isscalar()` returns `True` when applied to a numeric value, but `False` when applied to an array. So, we can use it in a conditional statement to do something different for numbers and arrays.

As for the `numpy.where()` function, let's break down what is happening there. First, x==0 returns an array of boolean values, as shown above. There is one boolean value for each element of x, and the only one that is True is the one associated with the 0 x-value. The `numpy.where()` returns an array with the same *shape* as the boolean array (in this case, an array 41 elements long). Wherever the boolean array is True, the value will come from the second argument, and wherever it is False it will come from the third argument. So, the return result from `numpy.where()` in this case is an array of sinc(x) values with the appropriate result where x is 0; i.e.,

```
>>> print sinc(x)
[ -3.89804309e-17  -3.35063038e-02   3.89804309e-17   3.74482219e-02
  -3.89804309e-17  -4.24413182e-02   3.89804309e-17   4.89707517e-02
  -3.89804309e-17  -5.78745248e-02   3.89804309e-17   7.07355303e-02
  -3.89804309e-17  -9.09456818e-02   3.89804309e-17   1.27323954e-01
  -3.89804309e-17  -2.12206591e-01   3.89804309e-17   6.36619772e-01
   1.00000000e+00   6.36619772e-01   3.89804309e-17  -2.12206591e-01
  -3.89804309e-17   1.27323954e-01   3.89804309e-17  -9.09456818e-02
  -3.89804309e-17   7.07355303e-02   3.89804309e-17  -5.78745248e-02
  -3.89804309e-17   4.89707517e-02   3.89804309e-17  -4.24413182e-02
  -3.89804309e-17   3.74482219e-02   3.89804309e-17  -3.35063038e-02
  -3.89804309e-17]
```

1.1.6 Plotting

For plotting we will use the `pylab` module. Documentation can be found at http://matplotlib.sourceforge.net/. Here is an example of how it is used:

```
>>> import pylab
>>> pylab.plot(x,y,'o-')
[<matplotlib.lines.Line2D object at 0x17204cb0>]
>>> pylab.xlabel('x')
<Matplotlib.text.Text object at 0x2f73bd0>
>>> pylab.ylabel('sinc(x)')
<matplotlib.text.Text object at 0x2f94bb0>
>>> pylab.show()
```

The pylab `pylab.plot()` function accepts two arguments which define the x and y coordinates of the data points that we wish to plot, and a third which in this case is a string that indicates we want circles at each data point with a line connecting them. The `pylab.xlabel()` and `pylab.ylabel()` functions plot labels on the x and y axes, respectively. Each of the commands return values that we will ignore.

The `pylab.show()` function causes the plot to be displayed (see Figure). Notice that `pylab.show()` takes no arguments.

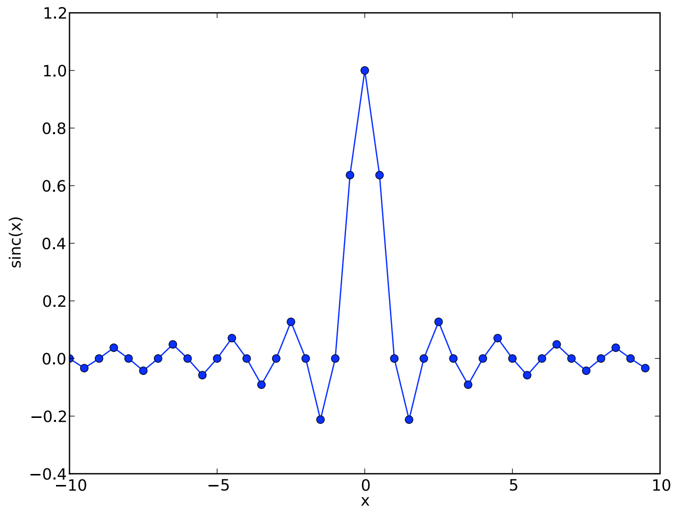

Figure 1.3: A sinc curve evaluated a low resolution.

Ugh. That is not what we expected. The problem is that we don't have enough data points to smoothly represent the line. We need to try again, starting with more values in the x array. Unfortunately, the `pylab.show()` function can only be called once per python session, and closing python (by typing <CTRL>-d) causes all of our work to be lost. Let's assume we have restarted python and retyped in all the necessary include statements and sinc function definition, and the follow with:

```
>>> x = numpy.arange(-10,10.1,0.1)
>>> y = sinc(x)
>>> pylab.plot(x,y,'o-')
[<matplotlib.lines.Line2D object at 0x1719a470>]
>>> pylab.xlabel('x')
<matplotlib.text.Text object at 0x2ef5390>
>>> pylab.ylabel('sinc(x)')
<matplotlib.text.Text object at 0x2f1e370>
>>> pylab.show()
```

That's much better. This demonstrates again that resolution in calculations is an important matter that we need to pay attention to.

Typing in all this code, and losing it when we close python is a major bother. Programs will need to be written into a text editor and saved if they are to be saved and reused.

1.2 Hello World

To create your first program, fire up your favourite text editor and write the following text on the first line:

```
print "Hello, World!"
```

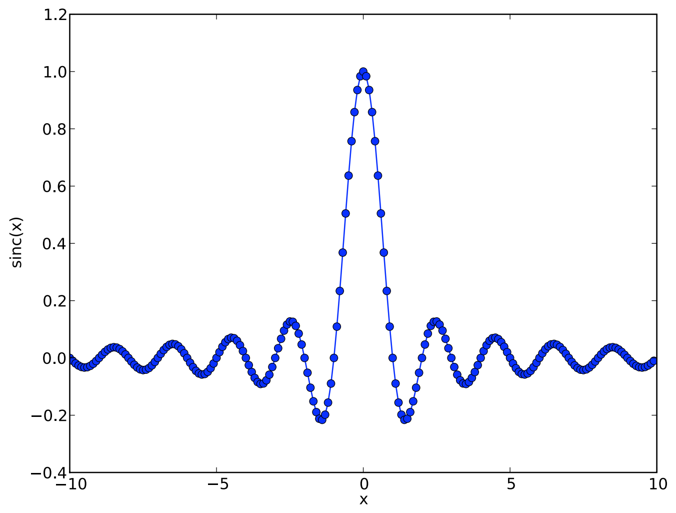

Figure 1.4: A sinc curve.

Save the program to `hello.py`. Python programs have a .py extension by convention. Run your program using the python interpreter from the bash prompt as follows:

```
$ python hello.py
Hello, World!
```

The program output is printed below the bash command.

1.3 Comparison to Other Programming Languages

Python is an *interpreted* language: python programs are read and acted on by the python interpreter, which is a *binary executable* (instruction set) native to the computer. You can write your own compiled programs as well. Here is the "Hello World!" program written in C:

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  puts("Hello, World!");
  return EXIT_SUCCESS;
}
```

The program looks different because C has a different syntax from python.

To *compile* the program into a binary executable, we use the *gcc compiler*:

```
$ gcc hello.c -o hello
```

You can execute the program and see its result as follows:

```
$ ./hello
Hello, World!
```

The execution time of a program can be determined:

```
$ time ./hello
Hello, World!

real    0m0.005s
user    0m0.001s
sys     0m0.003s
```

Our C program took only 5 ms to run. Compare this with the python program:

```
$ time python hello.py
Hello, World!

real    0m0.464s
user    0m0.013s
sys     0m0.039s
```

The python program took almost 100 times longer! To be fair, most of that time was due to loading the python interpreter into the computer's memory, a one-time-only expense. Nevertheless, python programs will in general take more time to run than the equivalent C program. On the flip side, the python program is much shorter and clearer.

To see the Hello World program in over 400 different programming languages, see The Hello World Collection at http://www.roesler-ac.de/wolfram/hello.htm. Even more fun is the 99 Bottles of Beer song written in over 1300 variations at http://99-bottles-of-beer.net/. Be sure to check out the perl version.

1.4 Fitting a Line to Data

Let's program something a little more advanced, like fitting a line to data. Start with a program that plots a series of points (see Figure):

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

# linefit1.py

import numpy, pylab

# Create a data series
x = numpy.arange(10)
y = numpy.array([6.1, 8.3, 8.8, 12.1, 13.8, 15.8, 16.5, 18.4, 20.0, 24.4])

# Plotting
pylab.plot(x,y,'o')
pylab.xlabel('x')
pylab.ylabel('y')

pylab.subplots_adjust(left=0.2,right=0.8,top=0.8,bottom=0.2)
pylab.show()
```

The first line [7] is a comment that indicates the file contains a python program. Comments are denoted by a hash symbol (#) at the beginning. You should include comments in your code that explain what you are doing in a block of

[7] Python programs can be made executable on Unix-like systems by changing the file mode; e.g. by executing **chmod +x hello.py** under bash. When the program is run (e.g., by entering **./hello.py** on the command line), the shell consults the first line of the file to determine the execution environment. If the first line is '`#! /usr/bin/env python`' then it will be python. In other cases we might use perl, or even bash. The first line written in this way is called a *shebang* line, due to the fact that the first two characters are hash (#) and bang (!).

code and why you are doing it.

The `numpy.array()` function returns an array constructed from a list of numbers that we specify. Go ahead – test the type of the argument to `numpy.array()` on your own. A list is a comma-separated series of values contained in square brackets as shown. Lists look and behave a lot like arrays (in fact, you can often use them in place of arrays with `numpy`, `scipy`, and `matplotlib`), but have somewhat different behaviours (like how integers and floating-point numbers are different) that are not important to us now.

The `pylab.plot()` command has an extra argument that we have not seen before: 'o'. This argument instructs pylab to plot the data points as circles instead of as lines between them. Note that the order of the arguments matters. There is also the `pylab.subplots_adjust()` function, which was used to size plots for inclusion in this document.

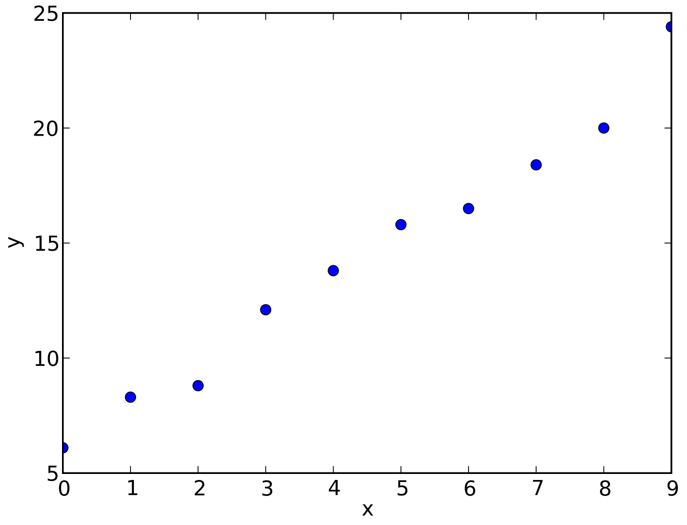

Figure 1.5: A sample data series to which we will fit a straight line.

How can we determine a "best-fit" line $y = A + Bx$ through the points? The "Introduction to Error Analysis" book by Taylor provides the equations for a linear least-squares fit:

$$A = \frac{\sum x^2 \sum y - \sum x \sum xy}{\Delta}$$

$$B = \frac{N \sum xy - \sum x \sum y}{\Delta}$$

$$\Delta = N \sum x^2 - (\sum x)^2$$

B is the slope and A is the y-axis intercept. The uncertainties in A and B are given by

$$\sigma_A = \sigma_y \sqrt{\frac{\sum x^2}{\Delta}}$$

and

$$\sigma_B = \sigma_y \sqrt{\frac{N}{\Delta}}$$

respectively, where N is the number of points in the series and

$$\sigma_y = \sqrt{\frac{1}{N-2} \sum (y - A - Bx)^2}$$

is an estimate for the uncertainty in the measurements of y.

The program is modified to implement the fit calculations (without errors) as follows:

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

# linefit2.py

import numpy, pylab

# Create a data series
x = numpy.arange(10)
y = numpy.array([6.1, 8.3, 8.8, 12.1, 13.8, 15.8, 16.5, 18.4, 20.0, 24.4])

# Least-squares calculations
N = len(x)
D = N * numpy.sum(x**2) - (numpy.sum(x))**2     # Delta computation
A = (numpy.sum(x**2) * numpy.sum(y) - numpy.sum(x) * numpy.sum(x*y)) / D
B = (N * numpy.sum(x*y) - numpy.sum(x) * numpy.sum(y)) / D

# Create points for the straight-line fit
y_fit = A + B*x

# Plotting
pylab.plot(x,y,'o')
pylab.plot(x,y_fit)
pylab.xlabel('x')
pylab.ylabel('y')

pylab.subplots_adjust(left=0.2,right=0.8,top=0.8,bottom=0.2)
pylab.show()
```

Notice that we are using the same operators with arrays that we used earlier for numbers. Arrays can be added, multiplied, etc. We can even add or multiply an array by a single numberical value. The `numpy.sum()` function used above is new to us. It returns the the sum of all values in an array.

Execution of the program fits a straight line to the data points, as required (see Figure).

Fitting a line to data points is a useful capability that we can expect to use again. We can define our own function for linear fits as follows:

```
import numpy

def fit(x,y):
    N = len(x)
    D = N * numpy.sum(x**2) - (numpy.sum(x))**2     # Delta computation
```

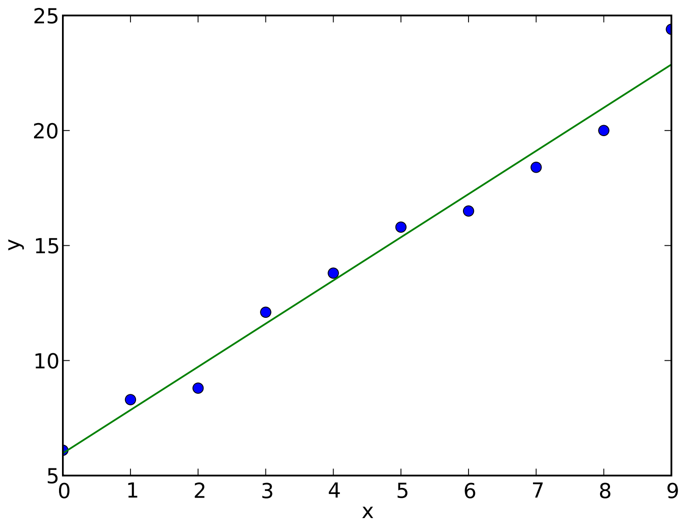

Figure 1.6: Straight-line fit to a data series.

```
    A = (numpy.sum(x**2) * numpy.sum(y) - numpy.sum(x) * numpy.sum(x*y)) / D
    B = (N * numpy.sum(x*y) - numpy.sum(x) * numpy.sum(y)) / D

    return A, B     # Returns two values rather than the normal one
```

Here is how the function could be included in our program:

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

# linefit3.py

import numpy, pylab

# Create a data series
x = numpy.arange(10)
y = numpy.array([6.1,8.3,8.8,12.1,13.8,15.8,16.5,18.4,20.0,24.4])

# Define a function to do the fit
def fit(x,y):
    """Returns the y-intercept and slope for a straigt-line fit"""

    # Determine the coefficients
    N = len(x)
    D = N * numpy.sum(x**2) - (numpy.sum(x))**2     # Delta computation
    A = (numpy.sum(x**2) * numpy.sum(y) - numpy.sum(x) * numpy.sum(x*y)) / D
    B = (N * numpy.sum(x*y) - numpy.sum(x) * numpy.sum(y)) / D

    return A, B

# Least-squares calculations
A, B = fit(x,y)     # Unpacks two values

# Create points for the straight-line fit
y_fit = A + B*x

# Plotting
```

```
pylab.plot(x,y,'o')
pylab.plot(x,y_fit)
pylab.xlabel('x')
pylab.ylabel('y')

pylab.subplots_adjust(left=0.2,right=0.8,top=0.8,bottom=0.2)
pylab.show()
```

We can put the code for the function anywhere before it is used. Notice that the function returns two values, and so we have provided two variables `A` and `B` on the left-hand-side of the assignment statement.

To make this fit function available to other programs, we can save it to a new module named `tools` in file `tools.py`. To use it, just import the `tools` module and modify the original program to call `tools.fit()` as follows:

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

# linefit4.py

import numpy, pylab

import tools     # Note that we drop the .py extension here

# Create a data series
x = numpy.arange(10)
y = numpy.array([6.1,8.3,8.8,12.1,13.8,15.8,16.5,18.4,20.0,24.4])

# Least-squares calculations
A, B = tools.fit(x,y)     # Unpacks two values

# Create points for the straight-line fit
y_fit = A + B*x

# Plotting
pylab.plot(x,y,'o')
pylab.plot(x,y_fit)
pylab.xlabel('x')
pylab.ylabel('y')

pylab.subplots_adjust(left=0.2,right=0.8,top=0.8,bottom=0.2)
pylab.show()
```