Chapter 1

Getting Started with Python

This chapter is not meant to be a comprehensive introduction to the Python language and all its features. It provides just enough Python to get you started and give you the tools to start doing significant and interesting computations. More advanced language constructs are introduced in later chapters, in the context of scientific and mathematical problems for which these constructs can form a useful part of the computational strategy.

1.1 Python 2 or Python 3?

1.2 Ways to interact with Python

You should read this chapter with your favorite Python interpreter at your side, typing in the examples to make sure you can get them to work. Then you should make up some similar examples of your own, and also do the suggested exercises as you read along. At the start, examples are simple enough that they can be done by just typing into the interpreter. As the examples get more complex, it becomes better to type the example into an editor and save it, then execute the file in a way that leaves you in the interpreter (e.g. `python -i`, or executing the file in an integrated development environment) so that you can do further experimentation with the things defined in the file.

1.3 Basic operations and data types

Let's start with a few operations with integers (whole numbers). In Python, a number without a decimal point is an integer. If you type an expression into the interpreter, Python will evaluate it and give you the result. The symbols used for addition and subtraction are the usual `+` and `-` signs. The `-` sign followed by a number (without any spaces) denotes a negative number. The symbol used for multiplication is `*`. Here is an example, just as you would see it when you type the expressions into the interpreter after each command prompt `>>>`.

```
>>> 2*3
6
```


DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

```
>>> 2-3
-1
>>> -1 + 3
2
>>> 1*2*3*4*5
120
>>> 1+2+3+4+5
15
```

In the last two examples, it doesn't matter in what order the computer carries out the operations, since multiplication of integers is *commutative* (i.e. $a * b = b * a$) as is addition ($a + b = b + a$). In cases where the order of operations matters, you need to be aware of the rules dictating the order in which operations are carried out. The operations are not done in sequential left-to-right order, but instead use *rules of precedence*. First all the multiplications are carried out, then all the additions. If you want some other order of operations, you specify the order by grouping terms with parentheses. The following example illustrates the general idea.

```
>>> 2*3 + 1
7
>>> 1+2*3
7
>>> 2*(3+1)
8
```

We'll introduce a few more rules of precedence as we introduce additional operators, but generally speaking it never hurts to group terms with parentheses, and you should do so if you are not absolutely sure in what order the computer will carry out your operations.

Python integers can have an arbitrary number of digits. No integer is too big, at least until you run out of memory on your computer. For example, we can find the product of the first 26 integers as follows:

```
>>> 1*2*3*4*5*6*7*8*9*10*11*12*13*14*15*16*17*18*19*20*21*22*23*24*25*26
403291461126605635584000000L
```

The `L` at the end serves as a reminder that the result is a "long" integer, which can have an arbitrary number of digits. All Python integers are treated as long (rather than fixed length) integers when they get long enough to need it. For the native integer data type, the distinction between fixed-length and long integers is removed in Python 3.

For integers, defining division is a bit tricker, since you can't divide any two integers and always get an integer back. Clearly, we want 4 divided by 2 to yield 2, but what should we do with 5 divided by 2? The actual answer is 2.5, which is not an integer, but we get a very useful integer to integer operation if we round the result to the next *lower* integer to the actual result (in this case 2). Note that this definition has some unintuitive implications when we do an integer divide into a negative number: the integer divide of -5 by 2 is -3, and not -2 as might have been thought. Python represents integer division of this sort with the operator `//`. Here are a few examples:

```
>>> 5//2
2
```

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

```
>>> -5//2
-3
>>> 5//-2
-3
>>> 5 - 2*(5//2)
1
```

The third example here illustrates the property that if m and n are integers, then the remainder $m*(n//m)-n$ is a positive integer in the sequence $0, 1, ...(n-1)$. In mathematics, this kind of remainder is represented by the expression $n(\text{mod}m)$, where "mod" is short for "modulo". Python uses the symbol `%` for this operation, as in

```
>>> 5%2
1
>>> 5%3
2
>>> -5%2
1
>>> 720%3
0
```

The mod operator is a very convenient way to tell if an integer n is divisible by another integer m, since divisibility is equivalent to the statement that $n(\text{mod}m)$ (implemented as `n%m` in Python) is zero. Testing divisibility is often used in number theory, but beyond that, in writing programs one often wants to perform some operation (such as writing out a result) every m steps, and the easiest way to do this is to keep some kind of counter and then do the required operation whenever the counter is divisible by m.

Exponentiation is represented by the operator `**`, as in:

```
>>> 2**100
1267650600228229401496703205376L
```

Exponentiation takes precedence over all the other arithmetic operators we have introduced so far. That includes the unary `-` operator, so that `-1**2` evaluates to `-1`, since the order of operations says first we exponentiate `1` and then we take its negative. If we wanted to square `-1`, we'd have to write `(-1)**2` instead.

You can store intermediate results in a named container using the assignment operator, which is the `=` sign in Python. Once a result is stored, it can be used in other operations by name later, and the contents of the container can be printed by simply typing the name of the container. Container names can be any text without spaces, so long as it doesn't begin with a digit, and is not one of the words reserved for Python commands or other elements of the Python language itself. Python will warn you if you use one of these by accidents. Here are some examples.

```
>>> x = 2 ; y=3
>>> z = x*x + y*y
>>> z
13
```

Here, we have also illustrated the fact that multiple Python statements can be put on a single line, separated by a semicolon. Containers fulfill a role loosely similar to "variables" in algebra,

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

in that they allow us to specify operations to be performed on some quantity whose value we may not know. Really containers just define a new name for whatever is on the right hand side of the `=` sign. As we start working with objects in Python other than simple numbers, it will become necessary to pay some attention to whether the container refers to a separate copy of the thing on the right hand side, or is referring directly to the very same object, though by an alternate name. The real power of containers will come later, when we introduce ways to define operations on containers whose values have not yet been assigned, and which will not be carried out until we plug in the values we want.

In mathematics, expressions that increment a variable by either adding a quantity or multiplying by a quantity are so common that Python has shorthand assignment operators of the form `+=` and `*=`. For example the assignment `x += 1` is equivalent to the assignment `x = x+1`, and the assignment `x *= 2` is equivalent to the assignment `x = x*2`. This works for all Python data types for which the respective operations are defined.

A great deal of mathematics, and probably most of physical science, is formulated in terms of *real numbers*, which characterize continuous quantities such as the length of a line or the mass of an object. Almost all real numbers require an infinite number of digits in order to be specified exactly. Nothing infinite can fit in a computer, so computations are carried out with an approximation to the reals known as *floating point numbers* (or *floats* for short). The floating point representation is based on scientific notation, in which a decimal fraction (like 1.25) is followed by the power of ten by which it should be multiplied. Thus, we could write 12000.25 as $1.200025 \cdot 10^4$, or .002 as $2 \cdot 10^{-3}$. In Python, the exponent is represented by the letter `e` immediately preceded by a decimal fraction or integer, and immediately followed by an integer exponent (with a minus sign as necessary; A plus sign is allowed but not necessary). Further, you can write the number preceding the exponent with the decimal point whereever you find convenient. Thus, `12000.` could be written `1.2e4`, `12e3` or `.12e5`, as well as any number of other ways. If the exponent is zero, you can leaave it out, so `12.` is the same number as `12e0` . Unlike Python integers, floating point numbers retain only a finite number of digits. Although numbers (both integer and floating point) are by default given in the decimal system for the convenience of the user, inside the computer they are stored in binary form. When dealing with a decimal fraction that cannot be represented exactly by a string of binary digits short enough for the computer to handle, this introduces a small amount of round-off error. There is also a maximum number of binary digits assigned to the exponent, which limits the largest and smallest floating point numbers that can be represented. The values of these limits are specific to the kind of computer on which you are running Python.

Python will treat a number as a float if it includes a decimal point, or if it includes an exponent specification (regardless of whether the preceding digits include a decimal point. Thus `1.`, `1.0` , `1e20` and 1.e20 are all floats. In displaying the value of a float, by default Python writes it out as a decimal fraction without the exponent if the number of digits is not too big, but puts it in standardized scientific notation if the number would get too long. For example:

```
>>> 15.5e10
155000000000.0
>>> 15.5e30
1.55e+31
```

This is just a matter of display. `12000` and `1.2e4` are the same number inside the computer and handled identically. In Section 1.14 we will discuss how to control the way a number is displayed.

The arithmetic operators `+` , `-` , `*` ,`**` work just the way they do for integers, except they now keep track of the decimal point. Floating point division is represented by `/` .

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

```
>>> 2.5e4*1.5 + 3.
37503.0
>>> 1./3.
0.3333333333333333
>>> 3.*0.33333333333333333
1.0
```

Division has equal precedence with multiplication, so in a term involving a sequence of `*` and `/` operations, the result is evaluated left-to-right unless a different order is specified with parentheses:

```
>>> 2.*3./4.*5.
7.5
>>> 2.*3./(4.*5.)
0.3
```

This can easily lead to unintended consequences, so it is usually best to be on the safe side and enclose numerator and denominator expressions in parentheses. The same remark applies to integer division `//` .

Though the modulo operator is conventionally defined just for integers in number theory, in Python, it is also defined for floats. An expression like `x%y` will subtract enough multiples of `y` from the value `x` to bring the result into the interval extending from zero to `y`. This can be very useful for periodic domains, where, for example, the point $x + nL$ is physically identical to the point x for any integer n.

Exponentiation can be carried out with negative or fractional exponents, e.g. `2.**.5` is the same as $\sqrt{2}$ and `2.**-1` is the same as `1./2.`. Negative floats can be raised to an integer power, but if we try to raise a negative float to a non-integer power Python responds with an error message:

```
>>> (-1.5)**3
-3.375
>>> (-1.5)**3.0
-3.375
>>> (-1.5)**-3
-0.2962962962962963
>>> (-1.5)**3.1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: negative number cannot be raised to a fractional power
```

This is because the result of raising a negative number to a fractional power is not a real number. We'll see shortly that this operation is perfectly well defined if we broaden our horizon to include complex numbers, but Python does not do complex arithmetic unless it is explicitly told to do so. Note that, as shown by the second example, Python knows when a number written as a float has an integer value, and acts accordingly.

In any operation involving two different types, all operands are promoted to the "highest" type before doing the operation, with a float being considered a higher type than an integer. Thus, the expression `1 + 1.5` yields the float result `2.5`. The exponentiation operator also promotes

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

operands to float when using negative exponents, even if everything is an integer. Thus `2**-2` evaluates to `0.25` but `2**2` yields the integer `4`.

What do you do if you want to divide two integers and have the result come out like a floating point divide? One way to do that is to just put a decimal point after at least one of the integers so that Python treats the division as floating point – you only really need to do this to one thing in the numerator or denominator, since everything else will be promoted to a float automatically. Thus, `1./2` or `1/2.` will both evaluate to `.5`. But suppose you have two variables `a` and `b` which might contain integers, but you want the floating-point quotient of them? One way to do that is to force a conversion to a float by multiplying either the numerator or denominator by `1.` There are other ways to do a type conversion, which you'll learn about shortly, but this will do for now. Then if `a=1` and `b=2` we get the desired result by typing `(1.*a)/b`. It is a bit silly to have to do this to force a float division when we would generally use `//` if we really wanted integer division, but in Python 2.x this is indeed what you need to do. It will still work in Python 3, but Python 3 made the sensible division to always treat `a/b` as float division even if `a` and `b` are integers.

Python has complex numbers as a native (i.e. built-in) data type. The symbol `j` is used for $\sqrt{-1}$, and the imaginary part of a complex constant is given by a floating point number immediatedly followed by `j`, without any spaces. Note that even if you want to represent $\sqrt{-1}$ by itself, you still need to write it as `1j`, since an unadorned `j` would be interpreted as a reference to a container named `j`. Here are some examples of the use of complex arithmetic:

```
>>> z1 = 7.5+3.1j ; z2 = 7.5-3.1j
>>> z1+z2
(15+0j)
>>> z1*z2
(65.86+0j)
>>> z1**.5
(2.7942277489593965+0.5547149836219465j)
>>> (-1+0j)**.5
(6.123233995736766e-17+1j)
```

It often happens that you need to create a complex number out of two floats that have previously been stored in containers. Here is one way you can do this:

```
>>> x=1.;y=2.
>>> z = x + y*1j
>>> z
(1+2j)
```

Note that simply writing `z=x+yj` will not work, since Python will try to interpret `yj` as the name of a container.

In the promotion hierarchy, `complex` is a higher type than `float`, so operations mixing `complex` and tt float values will yield a `complex` result. The final example above comes out a little bit differently from the exact answer `0 + 1j` because of the finite precision of computer arithmetic. Depending on the precision of the computer you are using, the answer you get may differ slightly from the example.This example also illustrates how Python guesses whether you wish an operation to be carried out using complex or real arithmetic – if there is any complex number involved, the whole operation is treated as complex. If you instead tried `(-1.)**.5` you would get

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

an error message, since it would be assumed the operation is meant to be carried out over the real numbers. Try evaluating `(-1.)**1j`, and check the answer against the expected result (which you can determine using the identity $e^{i\pi} = -1$). Also try evaluating `(-1.)**3.`, `(-1.+0j)**3.1` and `(-1.)**3.1`. Why do the first two work, whereas the last causes an error message?

The real and imaginary parts of native Python complex numbers are always interpreted as floats whether or not you include a decimal point. Complex integers (known to mathematicians as *Gaussian integers*) are not a native data type in Python, though in Chapter **??** you will learn how to define new data types of your own, including Gaussian integers.

Python also allows Boolean constants, which can take on only the values `True` or `False`. Various logical operations can be carried out on Boolean expressions, the most common of which are `and`, `or` and `not`. Here are a few examples of the use of Boolean expressions.

```
>>> ToBe = True
>>> ToBe or (not ToBe)
True
>>> ToBe and (not ToBe)
False
```

When applied to Boolean data, the ampersand (`&`) is a synonym for the keyword `and`, and the vertical bar (`|`) is a synonym for `or`.

Boolean expressions will prove useful when we need to make a script do one thing if some conditions are met, but other things if different conditions are met. They are also useful for selecting out subsets of data contingent on certain conditions being satisfied. Relational operators evaluate to Boolean constants, for example

```
>>> 1 < 2
True
>>> 1 > 2
False
>>> 1 == 2
False
```

Note the use of the `==` operator to test if two things are equal, as distinct from `=` which represents assignment of a value to a container. The compound relational operators are `>=` ("Greater than or equal to") and `<=` ("Less than or equal to). Since relational expressions evaluate to Boolean values, they can be combined using Boolean operators, as in

```
>>> (1>2) or not (6%5 == 0)
```

which evaluates to `True`

The final basic data type we will cover is the `string`. A string consists of a sequence of characters enclosed in quotes. You can use either single quotees (`'` or double quotes (`"`) to define a string, so long as you start and end with the same kind of quote; it makes no difference to Python which you use. Strings are useful for processing text. For strings, the `+` operator results in concatenation when used between two strings, and the `*` operator results in repetition when used with a string and a positive integer. Here are some examples:

```
>>> hobbit1 = 'Frodo'; hobbit2='Bilbo'
```

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

```
>>> hobbit1 +' ' + hobbit2
'Frodo Bilbo'
>>> hobbit1 + 2*hobbit2
'FrodoBilboBilbo'
```

This is a good example of the way operations can mean different things according to what kind of object they are dealing with. Python also defines the relational operator `in` for strongs. If `s1` and `s2` are strings, then `s1 in s2` returns `True` if the string `s1` appears as a substring in `s2`, and `False` otherwise. For example:

```
>>> crowd = 'Frodo Gandalf Samwise'
>>> 'Frodo' in crowd
True
>>> 'Bilbo' in crowd
False
>>> 'Bilbo' not in crowd
True
```

The relational operators `<` , `==` and `>` are also defined for strings, and are interpreted in terms of alphabetical ordering.

In some computer languages, called *typed* languages, the names of containers must be declared before they are used, and the kind of thing they are meant to contain (e.g. `float`, `int`, etc.) must be specified at the time of declaration. Python is far more free-wheeling: a container is created automatically whenever it is first used, and always takes on the type of whatever kind of thing has most recently been stored in it, as illustrated in the following:

```
>>> x = 2. ; y = 4.
>>> x+y
6.0
>>> x = "We were floats,"; y = " but now we are strings!"
>>> x+y
'We were floats, but now we are strings!'
```

Python's let-it-be approach to containers is very powerful, as it makes it possible to easily write commands that can deal with a wide variety of different types of inputs. This puts a lot of power in the hands of the user, but with this power comes a lot of responsibility, since Python itself isn't continually looking over your shoulder checking whether the type of thing put into a container is appropriate for the intended use. A well designed command (as we shall learn to write) should always do whatever checking of types is necessary to ensure a sensible result, though Python does not enforce this discipline.

1.4 A first look at objects and functions

Everything in Python is an object. Objects in the computer world are much like objects in the physical world. They have parts, and can perform various actions. Some of the parts (like the pistons in an automotive engine) are meant to be the concern of the designer of the object only and are normally hidden from the user. Others (like the steering wheel of a car) are meant to

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

interface with other objects (the driver) and cause the object to carry out its various purposes. Complex objects can be built from simpler objects, and one of the prime tenets in the art of object-oriented programming is to design objects that are versatile and interchangeable, so they can be put together in many ways without requiring major re-engineering of the components.

The parts of an object are called *attributes*. Attributes can be data, or they can be things that carry out operations, in which case they are called *methods*. in fact, the attributes of an object can be any object at all. In this section you will get a glimpse of how to use objects that somebody else has created for you. Later in this chapter you'll learn the basics needed to craft your own custom-designed objects, and later chapters will develop more facility with objects, through case-studies of how they can be used to solve problems.

The attributes of an object are indicated by appending the name of the attribute to the name of the object, separated by a period. For example, an object called `shirt` can have an attribute called `color`, which would be indicated by `shirt.color`. Since attributes can themselves be objects, this syntax can be nested to as many levels as needed. For example, if an object called `outfit` has `shirt` as an attribute, then the color of that `shirt` is `outfit.shirt.color`.

As an example, let's consider the complex number object `z` which is created for you when you write `z = 1+2j`. This has two data attributes, which give the real and imaginary parts as shown in the following example:

```
>>> z.real
1.0
>>> z.imag
2.0
>>> z.real**2 + z.imag**2
5.0
```

The last command in this example illustrates the fact that you can use attributes in any Python expression for which any other objects of their type (floats, in this case) can be used.

When an attribute is a method rather than simple data, then typing its name only gives you some information on the method. In order to get the method to actually do something, you need to follow its name with a pair of matched parentheses, `()`. For example, a complex number object has a method called `conjugate`, which computes the complex conjugate of the number as illustrated in the following example

```
>>> z.conjugate
<built-in method conjugate of complex object at 0x2c3590>
>>> z.conjugate()
(1-2j)
>>> z*z.conjugate()
(5+0j)
>>> (z*z.conjugate()).real
5.0
```

In this example, the `conjugate` method returns a value, which is itself a complex number, though in general methods can return any kind of object. Some methods do not return anything at all, but only change the internal state of their object, e.g the values of the attributes. The final part of the example shows that you can get an attribute of an object without assigning a name to the object.

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

Many methods require additional information in order to carry out their action. The additional information is passed to the method by putting it inside the parentheses, and takes the form of one or more objects placed within the parentheses, separated by commas if there are more than one. These items are called *arguments*, following the mathematical terminology in which one refers to an argument of a function. The type of object that can be passed as an argument depends on what the method expects, and a well-designed method will complain (by giving an error message) if you hand it an argument of a type it doesn't know how to handle.

For example, string objects have a method `upper` which takes no arguments and returns an all-capitalized form of the string. They also have a method `count` which takes a string as its argument and returns an integer giving the number of times that string occurs in string to which the method belongs. The `replace` method takes two strings as arguments, and replaces all occurrences of the first argument with the second. For example, if `s = 'Frodo Bilbo Gandalf'`, then

```
>>> s.upper()
'FRODO BILBO GANDALF'
>>> s
'Frodo Bilbo Gandalf'
>>> s.count('o')
3
>>> s.count('Gan')
1
>>> s.replace('Bilbo','Samwise')
'Frodo Samwise Gandalf'
```

Note that in in the `upper` and `replace` examples, the original string is left unchanged, while the method returns the transformed string.

Some arguments are optional, and take on default values if they are not explicitly specified. For example, the `strip` method of a string object, by default, strips off leading and trailing blanks if it is not given any arguments. However, if you can optionally give it a string containing the characters you want to strip off. Here's an example of how that works if `s = ' ---Frodo--- '`:

```
>>> s.strip()
'---Frodo---'
>>> s.strip('- ')
'Frodo'
```

Causing a method to do something is known as *invoking* the method, rather like invoking a demon by calling its name. This is also referred to as *calling* the method. Objects can themselves be callable, in which case one calls them by putting the parentheses right after the object name, without the need to refer to any of the object's methods.

In computer science, a callable object is often referred to as a *function*, because one of the things a callable object can do is define a mapping between an input (the argument list) and an output (the value or object *returned* by the function). This is pretty much the way mathematicians define functions. However, functions in a computer language are not precisely analogous to functions in mathematics, since in addition to defining a mapping, they can change the internal state of an object, or even change the state of objects contained in the argument list. Because of the generality of what they can do, functions in Python need not return a value, and for that matter need not have any argument list within the parentheses.

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

Nonetheless, many functions in Python do behave very much like their counterparts in mathematics. For example, the function `abs` implements the mathematical function which returns the absolute value of an argument x, namely $|x|$. Just as in mathematics, the definition of the function depends on the type of argument, the allowable types being real (called `float` in Python) and complex. For a complex argment z with real part x and imaginary part y, the absolute value is defined as $\sqrt{x^2+y^2}$, whereas for a real argument, the absolute value of x returns x itself if $x \geq 0$ and $-x$ if $x < 0$. A well behaved computer function checks for the type of its argument list and behaves accordingly; it also makes sure the number of arguments is appropriate.

```
>>> abs(-1.)
1.0
>>> abs(1+1j)
1.4142135623730951
>>> abs(1,2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: abs() takes exactly one argument (2 given)
```

Python includes a small number of predefined functions (like `abs`) which are available as soon as the interpreter is started. Many of these convert data items from one type to another. For example, the function `int` converts its argument to an integer. If the argument is a float, the conversion is done by truncating the number after the decimal point. Thus, `int(2.1)` returns the integer `2`, and `int(-2.1)` returns the integer `-2`. Similarly, the function `float` converts its argument to a float. The function `complex`, when called with two integer or float arguments, returns a `complex` number whose real part is the first argument and whose imaginary part is the second argument. This function provides a further indication of the versatility of Python functions, as the function examines its argument list and acts accordingly: you don't need a different function to convert pair of `int` data types vs. a pair of `float` data types, or even one of each. For that matter, if `complex` is called with just a single argument, it produces a `complex` number whose real part is the `float` of the sole argument and whose imaginary part is zero. Thus, `complex(1,0)` and `complex(1.)` both return the complex number `1.+0.j`. The ability to deal with a variety of different arguments goes even further than this, since the type conversion functions can even convert a string into the corresponding numerical value, as illustrated in the following example:

```
>>> x = '1.7 ';y='  1.e-3'
>>> x+y
'1.7   1.e-3'
>>> float(x)+float(y)
1.7009999999999998
```

(note the effect of roundoff error when the addition is performed). This kind of conversion is very useful when reading in data from a text file, which is always read in as a set of strings, and must be converted before arithmetical manipulations can be performed. Going the other way, when writing out results to a text file, it is useful to be able to convert numerical values to strings. This is done by the function `str`, whose use is illustrated here:

```
>>> x=2;y=3
>>> str(x) + ' plus ' + str(y) + ' equals ' + str(x+y)
'2 plus 3 equals 5'
```

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

Python functions and methods can have optional arguments of two types: *positional* arguments, which follow the obligatory arguments but can be left off if not needed, and *keyword* arguments, which come at the end of the argument list and specify arguments by name, in the form *name* = *value*. For example, the function `round(...)` can be called with a single argument and in that case rounds the argument to the nearest integer. If called with an optional integer second argument, the second argument specifies the number of digits to round to:

```
>>> round(1.666)
2.0
>>> round(1.666,2)
1.67
```

A function `f` with an obligatory argument `x` and a keyword argument called `power` would be called using a statement like `f(0.,power= 2)`. We will encounter specific examples of such functions a bit later.

You will learn how to access more extensive libraries of mathematical functions in Section 1.5, and how to define your own functions in Section 1.10.

Unlike objects in the physical world, software objects, if well designed come equipped with a built-in manual that can't be lost. Python has extensive built-in help functions, which allow the user to find out what is in an object and how it can be used. Given that so much of Python is found in various language extensions the Python community has written, the availability of embedded documentation is beholden to the good behavior of the programmer. Python fosters a culture of good behavior, and tries to make it easy for developers to provide ample help and documentation integrated with the tools they have developed.

The main ways of getting help in Python are the `help()` and `dir()` functions. For example, you have learned about the `strip()` method that is one of the methods available to strings. Suppose you didn't know what methods or data attributes went along with a string, though? Rather than going to a handbook, you can use the `dir()` function to find out this sort of thing. For example, if `a` is a string, you can type `dir(a)` to get a list of all its methods, and also all its data attributes (e.g. its length). Then, if you want to know more about the `strip()` method you can type `help(a.strip)` (Warning: don't type `help(a.strip())`, which would look for help items on the words in the content of the stripped string!). Both strings and lists have many useful and powerful methods attached to them. Many of these will be illustrated in the course of the examples given in the rest of this Workbook, but you are encouraged to explore them on your own, by finding out about them using `dir()` and `help()`, and then trying them out.

Typically, when you do a `dir` on an object, several of the attributes returned will have names the begin and end with a double underscore, e.g. `'__gt__'` in the case of strings, which the string object uses in order to determine whether one string is considered greater than another. This is the naming convention Python uses for *private* attributes. Private attributes are mainly for the object's own internal use, and are not generally meant to be accessed directly by the user. Some object-oriented languages actually completely hide private methods from the user, but in Python the distinction is purely informative, and the naming convention is just meant to provide a hint to the user as to which methods are likely to be of most interest.

So when in doubt, try *help* and *dir*. One or the other will give you some useful information about just about anything in Python.

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

1.5 Modules

To do almost any useful science or mathematics with Python, you will need to load various libraries, known as *modules*. Actually, a module can be just an ordinary Python script, which defines various functions and other things that are not provided by the core language. A module can also provide access to high-performance extensions written using compiled languages.

To make use of a module with name `myModule`, you just type: `import myModule`. Objects in the module are accessed by prepending the module name to the object name, separated by a period, just the way you refer to attributes of an object. In fact, a module is itself a Python object, and the various things defined in it are its attributes. The attributes of a module can be constants, functions, or indeed any object that can be defined in Python.

For example, the standard math functions are in the module `math`, and you make them available by typing `import math`. To see what's there, type `dir(math)`, as you would for any other object. Now, to compute $sin(\pi/7.)$ for example, you type `math.sin(math.pi/7.)`. To find out more about the function `math.sin`, just type `help(math.sin)`. If you don't like typing `math.sin`, you can import the module using `from math import *` instead, and then you can just use `sin`,`cos`, etc. without the prefix. Python also allows you to refer to a module with a name of your choosing. For example, if you import the `math` module using the command `import math as m`, you can refer to the functions in the module as `m.sin` and so forth. This way of importing is often used to provide a shorthand for a module that has a long or unwieldy name.

The `math` module contains all the common trigonometric functions (`sin`, `cos`, `tan`) and their inverses (`asin`, `acos`, `atan`), as well as the exponential function `exp` and the natural logarithm `log`. The trigonometric functions work with radians rather than degrees. `math` also defines the constants `math.pi` and `math.e`. For a full listing of the functions in `math`, do `help(math)` and then do `help` on any function for which you require additional information, e.g. `help(math.sinh)`.

The math functions in the `math` module are meant to work over the real numbers. They will return an error message if given a complex argument, or if the result of the operation would be a complex number. All of the common math functions can be extended to work over the complex numbers, though. If you want to use the complex versions, you need to import the `cmath` module instead of `math`. The following example illustrates Euler's Identity, $e^{i\pi} = -1$ (within roundoff error):

```
>>> cmath.exp(1j*math.pi)
(-1+1.2246467991473532e-16j)
>>> cmath.log(-1.)
3.141592653589793j
```

Or, to take an example using the inverse sine function, `math.asin(2.)` returns an error message since $|\sin(x)| \leq 1$ for any real x. However, `cmath.asin(2.)` happily returns `-1.3169578969248166j`.

1.6 Lists and tuples

1.6.1 List Basics:Creation and indexing

Lists are among the most widely used and versatile data structures in Python. Lists contain any kind of data at all, and the elements can be of different types (floats, int, strings, even other tuples

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

or lists, indeed any kind of object at all).

Lists are defined by putting the items of the list between square brackets, separated by commas. Here is an example showing how to define a list and get at an element:

```
>>> a = [1,'two',3.0e4]
>>> a[0]
1
>>> a[1]
'two'
>>> a[2]
30000.0
>>> a[-1]
30000.0
>>> a[-2]
'two'
```

Note that the first item has index zero, and the indices of the other count from there. Negative indices count backwards from the last item in the list, as illustrated in the final two index references in the example.

Sublists can be made by specifying a *cross section* in place of an integer for an index. A cross section has the general form `i:j:k` which specifies the collection of indices starting with `i`, up through `j-1`, by increments of `k`. For example if `a` is `['first','second','third','fourth']` then

```
>>> a[0:4:2]
['first', 'third']
>>> a[1:3:1]
['second', 'third']
>>> a[1:4:1]
['second', 'third', 'fourth']
>>> a[0:4:2]
['first', 'third']
```

Cross sections of lists return a new list which copies the selected data from the original list. If you modify an element of a list cross section, the original list is unaffected. Similarly, modifying an element of the original list after a cross-section is created does not affect the cross-section. Thus,

```
>>> b = a[0:4:2]
>>> b[0] = 'Gotcha!'
>>> b
['Gotcha!', 'third']
>>> a
['first', 'second', 'third', 'fourth']
>>> a[0] = 'No you did not!'
>>> b
['Gotcha!', 'third']
```

When using cross-sections it is important to keep the distinction between copy and reference in mind. Lists are just one of many kinds of indexable objects in Python that can accept cross-sections as indices, and some of these objects behave differently from lists with regard to the issue

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

of whether the cross section is a new object or an alternate way to refer to the data in the original object. The mathematical `numpy` arrays you will learn about in Section 1.8, treat cross sections as references rather than copies.

The cross section notation allows for default values that allow you to save some typing in common situations. In `i:j:k` the default for the increment `k` is `1`, so that `i:j:1`, `i:j:` and `i:j` all mean the same thing. The default for `i` is to start from the beginning of the list and the default for `j` is to start from the end of the list, so that `:3:2` is the same as `0:3:2`. If the list has length `n`, then `1:n+1` is the same as tt 1:; both represent the elements starting fron the second one in the list and continuing through the end of the list. The cross section `::3` would represent every third element of the list starting with the first and continuing through to the end.

A list is an example of an *indexable object*. Indexing provides a systematic way of retrieving an item with some specified characteristics from a collection of items. In Python, indexing is a general concept, and any object can be indexable if the designer of the object has provided a way to associate an index with an item. The key used to index the collection is always enclosed in square brackets after the name of the object. For lists, the index is an integer or collection of integers, but Python has a very general notion of indexing, and it is common to find objects which are indexed by names (given as strings), arrays of boolean values, or indeed any object that the designer finds convenient.

1.6.2 List methods and functions that return lists

What if you want to append an item to the end of an existing list? Lists are objects, and have an `append` method that does this. If `a` is the list `[1,'two']` then the following shows how to append the float `3.5` to the list:

```
>>> a.append(3.5)
>>> a
[1, 'two', 3.5]
```

The `extend` method works just like `append` except that it takes a list as an argument and appends all the elements, e.g. typing `a.append([4,5])` after the above example turns `a` into `[1, 'two', 3.5, 4, 5]`. What if we want to put a new item someplace other than at the end of the list? The `insert` method does this for us. It takes two arguments. The first is an integer which gives the position (counting from zero) before which the new item is to be inserted, and the second argument is the item to be inserted. Thus, typing `a.insert(1,'one')` turns `a` into `[1, 'one', 'two', 3.5, 4, 5]`. That takes care of the basic ways to add a new item to a list. How do you remove an item? The `pop` method is the most generally useful way to do this. When invoked without an argument, `pop()` *removes* an item from the end of the list, but also returns the item in case you want to do something with it rather than just discarding it. It "pops" and item off the end of the list. Here's an example illustrating the use of `pop()` with the list `a` we have been working with:

```
>>> a.pop()
5
>>> a
[1, 'one', 'two', 3.5, 4]
>>> a.pop()
4
```

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

```
>>> a
[1, 'one', 'two', 3.5]
```

When used with an integer argument, `pop(i)` pops off an item from position `i` in the list instead of the last position. For example, `a.pop(0` would pop off the first item of list `a`.

The `pop` method is useful when an item in a list can be discarded after it is processed; if you wanted to keep the original data in the list, you would access the item using `a[i]` rather than `a.pop(i)`. Using `pop` saves you the trouble of keeping track of which items have already been processed, and also automatically frees up storage by getting rid of raw data that is no longer needed.

Python also provides the `remove` method for pruning an item from a list without returning its value. It takes a mandatory argument, which is the item to be removed, and it removes the *first* occurrence of this item in the list. Note that the argument of `remove` is the item itself, and not the index of the item; the item passed to `remove` will only be removed if it is an exact match to something in the list. The way this works is best explained by example. If `Hobbits` is the list `['frodo','bilbo','gandalf','bilbo']` then `Hobbits.remove('bilbo')` is the list `['frodo', 'gandalf', 'bilbo']`, but `Hobbits.remove('Bilbo'` would return an error message.

Lists have a number of other useful methods. Suppose `L` is the list `['a','b','c','a','a']`. The `count` method returns the number of times the argument appears in the list. For example `L.count('a')` returns the value `3`. The `index` method returns the index value of the *first* exact occurence of the argument. For example `L.index('a')` returns`0`, and `L.index('c')` returns `2`. Other list methods transform a list in-place instead of returning a value. The `sort` method sorts the list in ascending order, and the `reverse` method reverses the order in the list. The following examples illustrate the use of these methods:

```
>>> L.sort()
>>> L
['a', 'a', 'a', 'b', 'c']
>>> L.reverse()
>>> L
['c', 'b', 'a', 'a', 'a']
```

Remember, just as for any other object, you can find out a list's methods by calling `dir(...)` with the list as it's argument, and then find out what any of the methods does by calling `help(...)` on the method. For example, if `L` is a list, then calling `dir(L)` will tell us (among other things) that the list has a method called `extend`. Then, calling `help(L.extend)` will describe what that method does. Note that the name of the method you are calling for help on always needs to be attached to its object, since a method of the same name could well do something completely different when used with a different kind of object.

Python's handling of lists allows multiple assignments of values to variables with a single statement. Specifically, if a list of variables appears on the left hand side of an assignment and a list of values of equal length appears on the right hand side, then values are assigned in order to the items on the left. In this construction, enclosing the items of a newly created list in brackets is optional. Thus, if `L` is the list `[1,2,3]` the following statements all assign the value `1` to `x`, `2` to `y`, and `3` to `z`:

```
x,y,z = L
[x,y,z] = L
```

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

```
x,y,z = 1,2,3
x,y,z = [1,2,3]
```

and so forth. In doing computations, it is often necessary to exchange the values of two variables, for example in preparing for the next step of an iteration. Python's multiple-assignment feature provides a compact way of doing this, as illustrated in the following:

```
>>> next=1;previous=2
>>> next,previous = previous,next
>>> next
2
>>> previous
1
```

As always in Python, the values being put into the containers can be objects of any types, not just numbers.

Many methods and functions return lists. For example, if `S` is the string `'First,Second,Third'` , then `S.split(',')` returns the list `['First', 'Second', 'Third']`. Indeed, returning a list is one of the common ways for a function or method to return multiple results. This can be quite handy when used together with Python's multiple-assignment feature. For example, the statement `x,y,z = S.split(',')` would set `x` to `'First'` and so forth.

Another useful built-in function which returns a list is `range`, which returns lists of integers. `range(n)` returns a list of integers up to but not including `n`, starting with zero, e.g. `range(3)` returns `[0, 1, 2]`. `range(m,n)` does the same, except it starts with `m` (which can be negative). When used in the form `range(m,n,inc)` the integers in the list are incremented by `inc` (which must be an integer), e.g. `range(0,5,2)` returns `[0, 2, 4]`. `range(0,6,2)` returns the very same list, because the list returned always stops one short of the specified endpoint `n`. The increment can be negative, in which case the list still stops one item short of the endpoint `n`, e.g. `range(6,0,-2)` returns `[6, 4, 2]` . The way the endpoint is handled in `range` can be confusing, but if you keep in mind that the specified endpoint `n` is always excluded from the list returned, you'll rarely make a mistake. Remember, too, that since Python is an interpreted language, if you ever get confused about whether a call to `range` does what you want it to do, you only need to try out an example in the interpreter. Note that `range` produces only lists of integers, and requires that its arguments be integers. Later you will learn various ways to conveniently create lists of floats or other numeric types.

1.6.3 Operators on lists

Arithmetic operators like `+` and `*` are called *binary operators*, not because they operate on binary digits, but because they take two items as input and produce a third item. For example the statement `1 + 2` implements a function that takes the arguments `1` and `2` and maps them to the sum, i.e. `3`. In object-oriented languages like Python, binary operations can be defined for any object, and a number of binary operators are defined for lists. The same arithmetic operator symbols are also used for other objects, but they can mean very different things, depending on what the designer of the object had in mind. For lists, the most useful binary operator is `+`, which is used to concatenate lists, as in:

```
>>> a = [1,2];b=['one','two']
```

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

```
>>> a+b
[1, 2, 'one', 'two']
```

For lists, the * operator is defined only if the other item in the pair is an integer, and in that case creates a list consisting of multiple repetitions of the contents of the original list, as in:

```
>>> 3*[1,2]
[1, 2, 1, 2, 1, 2]
```

The * operator is not by default defined for pairs of lists, and attempting to use it on a pair of lists will raise an error message.

There are also a number of binary relational operators defined for lists, which take a list and another object as inputs, and produce a Boolean truth value as output. Of these, the `in` operator is one of the most useful. The `in` operator allows you to find out if a thing is in a list, It is used as follows:

```
>>> town = ['frodo','bilbo','gandalf']
>>> 'frodo' in town
True
>>> 'samwise' in town
False
>>> 'frod' in town
False
```

The final example shows that the `in` operator requires exact equality of the item being matched. The operator works with Python objects of any type, including integers and other arithmetic data types, but because of round-off error its utility for floats and similar data types is somewhat limited. The boolean values produced by `in` can be used in any boolean expression. For example, `('frodo' in town)or('samwise' in town)` evaluates to `True`.

The equality relational operator == returns `True` if each item in the first list is identical to the corresponding item in the second list. Here are some examples:

```
>>> [1,'frodo'] == [1,'frodo']
True
>>> [0,'frodo'] == [1,'frodo']
False
>>> [1] == [1,'frodo']
False
>>> [1,'frodo'] == ['frodo',1]
False
```

The other relational operators, e.g. < and > are also defined for lists, but their meanings are a bit obscure and they are less commonly used.

1.6.4 List comprehension: Building new lists from other lists

List comprehension refers to the ability to create new lists by processing elements of old lists. The operations on lists we have just seen are a simple form of list comprehension, but Python also

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

supports a much more general form. Many programming tasks that would require multiline loops and conditionals in other programming languages can be carried out compactly in Python using list comprehension. Adept use of list comprehension can make a program easier to write, easier to get working, and easier for others to understand.

The most basic form of a list comprehension is uses the expression

[*expression* `for` *variable* `in` *list* [

to create a new list, where *expression* is any Python expression, generally dependent on the dummy variable *variable*, and *variable* loops through all the elements of the list *list* . List comprehension in Python is especially powerful because the processing can loop over the elements of any list or list-like object. Here is an example computing the squares of a list of integers:

```
>>> X = [1,2,3]
>>> [val*val for val in X]
[1, 4, 9]
```

and here is an example converting a list of numbers in text form into floating point numbers and squaring them:

```
>>> TextNums = ['1.5','2.5','3.5']
>>> [float(s)**2 for s in TextNums]
[2.25, 6.25, 12.25]
```

You can even loop over lists of functions:

```
>>> [f(.5) for f in [math.sin,math.cos,math.tan] ]
[0.479425538604203, 0.8775825618903728, 0.5463024898437905]
```

Indeed you can loop over lists of any objects you like.

Actually, Python allows you to loop over a much broader class of objects than just lists. Just as lists are an example of an indexable object, they are also an example of a much broader class of objects called *iterable* objects. An iterable object is an object that can specify a start, a procedure to get from the current item in a sequence to the next, and a procedure for determining when to stop. Any indexable object is iterable, but an iterable need not be indexable. Iterable objects can be more efficient than lists, because they can compute items in sequence without ever storing all the items in the sequence.

The `range(...)` function is often used with list comprehension to create a regularly spaced list of floating point numbers, which can then be processed further as in:

```
>>> xList = [2.*math.pi*i/5. for i in range(6)]
>>> xList
[0.0, 1.2566370614359172, 2.5132741228718345, 3.7699111843077517,
   5.026548245743669, 6.283185307179586]
>>> [math.sin(x) for x in L]
[0.0, 0.9510565162951535, 0.5877852522924732, -0.587785252292473,
   -0.9510565162951536, -2.4492935982947064e-16]
```

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

The basic list comprehension construct can be qualified by appending a conditional `if ...` clause after the `for` clause, as in:

```
>>> xList = [-2.,-1.,0.,1.,2.]
>>> [x**.5 for x in xList if x >= 0.]
[0.0, 1.0, 1.4142135623730951]
```

The phrase following the `if` keyword can be any boolean expression. One can also provide for an alternative action when the conditional is not satisfied, by using a `if ... else ...` clause, but in that case, the conditional clause must be placed *before* the `for` clause, as in:

```
[x**.5 if x>= 0. else 'Imaginary' for x in xList]
['Imaginary', 'Imaginary', 0.0, 1.0, 1.4142135623730951]
```

Python has a number of built-in functions that operate on lists. `len(...)` returns the length of its argument, `max(...)` returns the maximum of the items in the list, `min(...)` returns the minimum and `sum(...)` returns the sum. The latter three work only if all the items in the list are object for which the respective operations (comparison or addition) are defined. Theses functions make it easy to compactly compute statistics of lists using list comprehension. For example, the following computes the average and variance of a list of numbers contained in `L`:

```
avg = sum(L)/float(len(L))
var = sum( [x*x for x in L])/float(len(L)) - avg*avg
```

Make up a list of numbers yourself and try this out.

Loops can be nested within list comprehensions, and nesting can also be used to create list of lists. Try out the following for some short list of numbers `xList` to see if you can understand what they do and why:

```
[x+y for x in xList for y in xList]
[ [i*x for x in xList] for i in range(3)]
```

Python provides the built-in functions `reversed(...)`, `enumerate(...)` and `zip(...)` to help make list comprehension more versatile. These functions work with any iterable object, and produce iterators rather than lists, so they are efficient. The function `reversed(...)` loops allows one to loop over a list or other iterable object in reverse order, from last element to first. For example, in order to compute the reciprocals of an ascending list of numbers but have them come out in ascending order, we could do

```
>>> data = [1.,5.,10.,100.]
>>> [1./x for x in reversed(data)]
[0.01, 0.1, 0.2, 1.0]
```

This could also be done by executing `data.reverse()` before the list comprehension to reverse the list in-place, but aside from being less efficient and requiring an extra line of code, using the `data.reverse()` method alters the list `data`, which you might want to use unreversed at some later point. Alternately, one could give the list being created a name and then use the `reverse()` method of this list afterwards to reverse it in place, but at the cost of using two statements instead

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

of one. Further, you might not need to put the list into a container for future use (e.g. you might want to pass the list directly as the argument of a function).

The function `enumerate(...)` is useful when one needs to do a calculation that depends on the index of an element in the list (i.e. its position in the list). It does this by creating a sequence of pairs in which the first element is the index (starting from zero) and the second is the corresponding item in the list,e.g.

```
>>> L = ['a','b','c']
>>> [pair for pair in enumerate(L)]
[(0, 'a'), (1, 'b'), (2, 'c')]
```

`enumerate(...)` is typically used together with Pythons multiple assignment feature, as in the following example which appends an index number to each name in the list

```
>>> [name+str(i) for i,name in enumerate(L)]
['a0', 'b1', 'c2']
```

Or, to give an arithmetic example, suppose `xList` is a list of floats and we want to create a list containing each item raised to the power of its index. This can be done as follows:

```
>>> xList = [2.,5.,11.]
>>> [x**i for i,x in enumerate(xList)]
[1.0, 5.0, 121.0]
```

`zip(...)` is like `enumerate`, except that it can take an arbitrary number of iterables as arguments and instead of creating index-item pairs creates tuples consisting of the corresponding items of the inputs – it "zips" together items from a collection of lists. To compute a list consisting of the sums of the squares of corresponding elements of lists `A`, `B` and `C`, we could do

```
[a**2 +b**2 + c**2 for a,b,c in zip(A,B,C)]
```

Here is an example computing the covariance statistic of data contained in lists `xList` and `yList`:

```
covariance = sum([x*y for x,y in zip(xList,yList)])/float(len(xList))
```

and here is an example using `zip` and list cross sections to compute the correlation between successive elements in a list of data `dataList`

```
sum( [val*nextval for val,nextval in zip(data[:-1],data[1:])] )/(len(data) - 1.)
```

If you are having trouble understanding how this example works, try making up a data array and printing out the two cross sections and the output of `zip(...)` to the screen. Note that since the output of `zip(...)` is an iterator and not a list, you need to use it in a simple list comprehension to build a list in order to see what it puts out; simply typing the function into the interpreter won't give you the results you want. Alternately, you can use the `print` statement to print out the results.

Almost anything you can do using `enumerate` and `zip` could also be done by looping over an integer index and using the index directly. The main rationale for these functions is to eliminate the introduction of unnecessary indices, which makes code less cluttered and easier to follow.

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

1.6.5 References or Copy?

Lists illustrate a fundamental aspect of objects, common to all object-oriented languages. You must always keep straight when you are making a fresh copy of an object, as opposed to just creating an alternate name by which the same object may be referenced. Here is an example that creates an alternate reference:

```
>>> A = ['one','two','three']
>>> B=A
>>> B[0] = 1
>>> A
[1, 'two', 'three']
```

Changing an element of `B` changes the original list, because `B` is just another reference to the same thing. The assignment operator in Python, when applied to an object name, always works this way: it creates a reference rather than a copy (except when the object on the right hand side is a basic data type such as a number – you cannot change the value of `2`) If you really wanted `B` to be a copy rather than a reference, you would need to deliberately create a fresh copy. This illustrates one way of doing it, using a cross-section:

```
>>> A = ['one','two','three']
>>> B = A[:]
>>> B[0] = 1
>>> A
['one', 'two', 'three']
>>> B
[1, 'two', 'three']
```

However, not all indexable objects in Python treat cross-sectons as fresh copies; that is something that is left in the hands of the designer of the object. A copy of the list could also be created using the statement `B = list(A)`. There is also a module `copy` which handles various kinds of copy operations for objects of arbitrary complexity, but for lists the cross-section method works fine. Some objects also have their own built-in methods for providing copies.

1.6.6 Lists vs Tuples

Python distinguishes between lists and tuples. Tuples are a lot like lists, except that lists can be modified but tuples cannot. Lists are denoted by square brackets, whereas tuples are denoted by parentheses. You can define a tuple, but once defined you cannot modify it in any way, either by appending to it or changing one of its elements – tuples are *immutable* objects. Certain kinds of operations can be done much more efficiently if Python knows that the objects in use are immutable, which is the main reason for introducing tuples. In cases where Python commands specifically require a tuple rather than a list, in which case you can turn a list (say, `mylist`) to a tuple by using the function `tuple(mylist)`. Likewise you can turn `mytuple` into a list by using `list(mytuple)`; then you can change it, and turn it back to a tuple if you need to.

Note that a string is not a list, but a string is also an indexable object which behaves a lot like a list. If `s` is the string `d`, and `s[1:3]` evaluates to `'abcde'` , then `s[3]` evaluates to the string `'d'` , and `s[1:3]` evaluates to `'bc'` . However, lists, like tuples are immutable, so an assignment

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

like `s[3] = 'X'` will return an error message. All the string methods, like `replace` that change the string leave the original string intact but return a *new* string with the desired changes made.

1.7 Dictionaries

Dictionaries are another class of fundamental Python data structure. Dictionaries are a lot like lists, and are indexable objects, but the index can be any immutable Python object, which is referred to as a *key*. Python dictionaries are a generalization of the everyday use of the word, in that they are used to look up the data corresponding to a key. For an everyday dictionary, the key would be a word, and the data would be the word's definition, and while this can be directly implemented as a Python dictionary, and indeed often strings are used as keys. Compared to an everyday dictionary, a Python dictionary can use a wider variety of objects as keys and can store any kind of object as the corresponding data.

A dictionary consists of a sequence of items separated by commas, enclosed in curly brackets: { *item1*, *item2*, ... } . Each item consists of a pair separated by a semicolon, in which the first item is the key and the second is the value, i.e. *key*:*value* . The following dictionary would associate molecular weights with some common compounds:

```
D = {'H2O':18.,'CH4':16.,'CO2':44.,'H2':2}
```

and then `D['CO2']` would return the value 44. and so forth. A more common way to create a dictionary would be to first create an empty dictionary and then add items to it by setting the values of keys.

```
D = {}
D['H2O'] = 18.
D['H2'] = 2.
...
```

and so forth. The keys need not first be created – they are automatically added to the dictionary the first time they are used. Subsequent uses of a key reset the value associated with that key. There is always just one value associated with a given key in a dictionary.

Part of the power of dictionaries is that dictionary lookups are extremely fast, even if the dictionary is huge.

An attempt to look up a key whose value has not been set will raise an error message, so it is frequently necessary to check if the key is present before looking it up. This can be done using the `in` operator as in the following:

```
>>>'CO2' in D
True
>>>'C2H6' in D
False
```

Dictionaries, like lists, are iterable, which means you can loop over them in comprehensions and other kinds of loops you will learn about in Section **??**. Loops iterate over the keys, as in the following example:

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

```
>>>['MolWt of ' + key + ' is '+str(D[key]) for key in D]
['MolWt of H2 is 2',
 'MolWt of H2O is 18.0',
 'MolWt of CO2 is 44',
 'MolWt of CH4 is 16.0']
```

Note that the keys are not processed in the same order as they appear in the dictionary; this has to do with the way dictionaries are organized so as to allow fast lookup.

Dictionaries provide a very powerful tool for organizing computations, but are relatively underutilized in computational science. In subsequent chapters we will encounter many examples of how dictionaries can be used to streamline the implementation of computations.

1.8 The numpy array module

1.8.1 numpy array basics

Python lists look a little bit like the objects known to mathematicians and physicists as *vectors*, but in reality the only point of similarity is that both kinds of objects are indexable by integers. Vectors are special, in that they have a fixed dimension and consist of objects of identical type which permit arithmetic operations such as addition and multiplication (e.g. 3 real numbers in the case of a vector describing a point in space). Two vectors can be added to yield a third vector of the same dimension, and a vector can be multiplied by a *scalar* – a number of the same type that makes up the contents of the vector. Vectors can be transformed linearly through multiplication by a matrix of suitable dimensions. Vector spaces are so ubiquitous in mathematics and computational science that one needs a class of Python objects that implement something much closer to vector spaces.

This is where the real power of the extensibility feature of Python comes in. When the language is missing some feature some community really needs, the community gets together and writes a module which fills the bill. Sometimes this is a cooperative process. Sometimes it is an evolutionary process, with many competing extensions co-existing until one comes to dominate. For scientific arrays, the solution that has come to the fore is the `numpy` module, which provides highly efficient array objects. The `numpy` module actually implements a class of objects somewhat more general than a vector space as conventionally defined in mathematics, but it contains all the usual vector and linear algebra concepts as a subset.

`numpy` is not written in Python. It is written in a very highly optimized compiled language, which is why it is so efficient. The general strategy for high performance computation in Python is to do as little as possible at the compiled level, building tools there that are very general and of broad applicability. One seeks to isolate the computationally intensive work in a few compiled toolkits, and build more complex models out of these building blocks at the Python level.

`numpy` provides one of the most fundamental building blocks for scientific programming in Python, and most other Python modules doing numerical analysis or data analysis deal with `numpy` arrays. `numpy` not part of the core Python language, but so universally useful that it is included with most modern Python distributions. The `numpy` module is imported like any other module, using `import numpy`.

A `numpy` array is characterized by three properties:

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

- A *data type*, which is the type of data contained in each item of the array. All the items of the array must be of the same type. Typical data types used for arrays are integer, float and complex, but a number of other types are available. Note that the integers stored in `numpy` arrays are fixed-length integers, not arbitrary length Python long integers.
- A *rank*, which is the number of indices used to index elements of the array. A rank zero array is called a scalar, and is just a number referred to without any indices at all. A rank 1 array is a vector, indexed by a single integer, e.g. `a[i]` where `i` is some integer. A rank 2 array is a matrix, indexed by a pair of integers within brackets, e.g. `a[i,j]`. Arrays can have as high a rank as the application demands. In this book, we will generally refer to each index as belonging to an *axis* of the array. For example a rank 2 array has two axes, the first of which is axis 0 and the second of which is axis 1. As for lists, the indices of `numpy` arrays start at zero.
- A *shape*, which specifies the length of the array along each axis. For example, for a rank 2 array `a` having shape `(5,3)` the index `i` in a reference like `a[i,j]` ranges over the integers `0, 1,2,3,4` and the index `j` ranges over `0,1,2`. Negative indices are interpreted the same way as they are for lists. For example, if `a` is a rank 2 array with shape `(n,m)`, then `a[-1,-2]` would be shorthand for `a[n-1,m-2]` .

Arrays of rank 1 are often called "one-dimensional arrays" and arrays of higher rank are often called "multidimensional arrays" (e.g. a matrix being a "two-dimensional array") but this common terminology risks confusion with other uses of the term "dimension" in mathematics.

All the examples in this section presume you have first imported `numpy` in your current interpreter session, before entering any other commands.

1.8.2 Creating and initializing a `numpy` array

Before you can use a `numpy` array, you first need to create it. There are many ways to create an array. One common way is to create an array filled with zeros and then fill in the data you want. For example to create a rank 1 array called `a`, having length 3 and containing floats, and then set the values to what you want, you would do

```
>>> a = numpy.zeros(3,'float')
>>> a
array([ 0.,  0.,  0.])
>>> a[0] = -1. ; a[2] = 1.
>>> a
array([-1.,  0.,  1.])
```

The first argument of `numpy.zeros` is the shape of the array, and the second is the data type. Actually, `'float'` is the default data type for `numpy.zeros`, so the second argument is optional. The shape would ordinarily be specified as a tuple of integers giving the length of each axis, i.e. `(3)` for the example above, but for rank zero arrays the parentheses can be left out. To create a 2x2 matrix (rank 2 array) of float zeros, we would instead do

```
>>> M = numpy.zeros((2,2),'float')
>>> M
array([[ 0.,  0.],
       [ 0.,  0.]])
```

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

`numpy.ones(...)` works just like `numpy.zeros(...)` except that it fills the array with ones instead of zeros.

You can determine the shape and data type of an array from the `shape` and `dtype` attributes. e.g for the array defined above,

```
>>> M.shape
(2, 2)
>>> M.dtype
dtype('float64')
```

The latter tells us that the specific kind of float this kind of array contains is a 64-bit float. Unless you have a rather poor memory, it wouldn't be necessary to recover the information about the array from these attributes if you have created the array yourself, but the attributes are very useful when writing functions that process arrays, since the necessary information can be recovered from the attributes. If we want to create an array with the same shape and data type as `M`, but containing all ones, we could just use the statement `numpy.ones(M.shape,M.dtype)` .

Unless the array is very small, it is cumbersome to set the elements individually. `numpy.array(...)`, which converts lists to arrays, can be used with list comprehension to conveniently set the values of arrays of arbitrary shape. To create a rank 1 array with evenly spaced values, we could use

```
>>> a = numpy.array([.1*i for i in range(5)])
>>> a
array([ 0. ,  0.1,  0.2,  0.3,  0.4])
```

Higher rank arrays are defined by using lists of lists, as in

```
>>> numpy.array([ [1.,2.] , [3.,4.] ] )
array([[ 1.,  2.],
       [ 3.,  4.]])
```

This, too, can be used with list comprehension to create higher rank arrays. The following line creates a 5x10 integer rank 2 array (a matrix) whose rows contain powers of a list of numbers:

```
numpy.array( [ [ i**j for i in range(5)] for j in range(10)] )
```

Try it out and see how it works. Note that `numpy.array(...)` by default chooses the data type from the data type of the the input lists. If you want to force a conversion to a specific data type you can use the optional `dtype` keyword argument. If you wanted the previous example to create an array of 128 bit floats, you would use

```
numpy.array( [ [ i**j for i in range(5)] for j in range(10)] ,dtype = 'float128')
```

Another useful way to create an array is the function `numpy.arange(...)`, which works just like the Python built-in function `range(...)` except that it creates a rank 1 numpy array instead of a list. By default, it creates an array of integers, and while it can also accept float arguments and create arrays with non-integer increments, the typical use of `arange` is to create `numpy` integer arrays, which are subsequently processed by various other operations into float arrays. `numpy.linspace(...)` is useful for creating a float array of equally spaced values with specified starting point (the first argument) ending point (the second argument) and number of points. The statement

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

```
x = numpy.linspace(0.,2.,n)
```

produces the same array as the statement

```
numpy.arange(5)*(2./(n-1))
```

Try this out and verify that it works. Remember that you will need to set `n` to some integer value before executing either one of these statements.

1.8.3 numpy self-documentation features

`numpy` is a very large and versatile module, and the results of the standard Python `help(...)` function are not always very illuminating. `numpy` provides a special documentation function, `numpy.info(...)`, which provides more detailed and user-friendly information on any `numpy` routine or object passed to it as an argument, generally providing some examples of usage. Try `numpy.info(numpy.linspace)` and a few other queries to get a feel for the kind of information that is available.

1.8.4 Operations on numpy arrays

Once an array is created, various operations performed with existing arrays automatically create new arrays to store the results. This is the real power of `numpy` arrays, since the array operations are extremely fast, even for large arrays. `numpy` arrays support all the operations that will be familiar to those who have dealt with linear algebra or vector Two arrays of the same shape can be added to yield a new array of the same shape, whose elements are the sum of the corresponding elements of the arrays added. Also, a `numpy` array can be multiplied by a scalar, which yields a new array of the same shape, each of whose elements are multiplied by that scalar. You needn't create a special `numpy` scalar for operations of this sort, though that is possible.

Adding a scalar to an array is not a standard vector space operation, but this kind of operation is so common in other uses of arrays in computational science that it is implemented for `numpy` arrays, in the following way. If you add a scalar to an array, it will be "broadcast" to an array of identical values of the same type, with the same shape as the array to which you are adding the scalar. Thus, if `A` is some `numpy` array the statement

```
A1 = A + .5
```

is essentially shorthand for the statement

```
A1 = A + .5*numpy.ones(A.shape,A.dtype)
```

Make up an array `A` and try this out yourself to see how it works.

The modulo operation `%` also works pointwise between pairs of arrays of the same shape, and either array can be replaced by a scalar.

Multiplication of arrays, indicated by the usual `*` operator, is not defined as matrix multiplication. Instead, it acts much like array addition. Two arrays can be multiplied if they have identical shapes, and results in an array of the same shape whose elements are the products of the

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

corresponding elements of the input arrays. Similarly, array division, indicated by `/`, is carried out pointwise for the elements of the arrays. There are ways of carrying out the standard linear algebra operations on arrays, such as dot product, matrix multiplication and matrix inversion. These are introduced in Section 1.8.10, and developed further in Chapters **??** and **??**.

In summary, the array arithmetic operators work just like scalar operators, except the operations are performed "pointwise" for each pair of corresponding elements from the input arrays, with the result put in the corresponding place in the output array. Array arithmetic provides a compact and very fast way to perform the same operation on many different numbers.

Array arithmetic can be done between arrays of different types, if the operation makes sense. The result is promoted to the higher of the two operands. For example, adding an integer and a complex results in a complex, or adding a 64 bit float to a 128 bit float yields a 128 bit float. You can determint the default float type on your system by just creating an array using `numpy.zeros(10,'float')` and looking at the resulting `dtype` attribute. Operations between float arrays and Python scalars do not change the type of the resulting array, though operations between an integer array and a Python float will promote the resulting array to a default float type.

`numpy` provides a number of math functions which are similar to those in the `math` module, but operate on arrays and return an array of the same shape resulting from applying the the appropriate math function to each element of the input array. The following example shows how this works.

```
>>> a = numpy.array([-1.,0.,1.])
>>> b = numpy.exp(a)
>>> b
array([ 0.36787944,  1.        ,  2.71828183])
>>> numpy.log(b)
array([-1.,  0.,  1.])
>>> numpy.sin(a)
array([-0.84147098,  0.        ,  0.84147098])
>>> numpy.arcsin(a)
array([-1.57079633,  0.        ,  1.57079633])
```

Note that, as in the `math` module, `log(...)` computes the natural logarithm. Although `math.log(...)` takes an optional second argument that gives the base to which the logarithm is to be computed, `numpy.log(...)` does not provide for computation of logarithms to an arbitrary base in this way. However, one can do the computation using the usual simple mathematical formula for conversion between bases. For example, to compute the log of an array `b` to the base 3, one would simply do `numpy.log(b)/numpy.log(3.)`. `numpy` does, however, provide functions `numpy.log2(...)` and `numpy.log10(...)` for the most commonly used bases. `numpy` provides array versions of all the usual elementary functions, trigonometric functions and hyperbolic functions under the usual mathematical names. The inverse trigonometric and hyperbolic functions use the prefix `arc` as in the example above. `numpy.info(...)` will provide detailed definitions and usage examples for any of the math functions.

When applied to a higher rank array, the array math functions work similarly to the rank 1 example, returning an array of the same rank and shape as the input array. Try this with some rank 2 arrays to see how it works.

All the `numpy` math functions can operate on complex as well as float arguments, and unlike the functions in `math` will detect the type of the argument and act accordingly. Consider, however,

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

the function $\log(x)$, which is defined even for negative x if one allows the result to be complex (e.g. $\log(-1) = \pi$), but which is undefined for negative x if one insists that the result be a real number. If a negative real is handed to `numpy.log(...)`, how should the function behave? The way `numpy` deals with this situation is to interpret the function as a function that is intended to be real-valued if the input is real (i.e. `float`) but to interpret it as a complex-valued function if the input has a complex data type – even if the imaginary part happens to be zero. Here is an illustration of this behavior in action:

```
>>> a =numpy.array([-1.,1.])
>>> b = numpy.array([-1.+0j,1.+0j])
>>> numpy.log(a)
array([ nan,   0.])
>>> numpy.log(b)
array([ 0.+3.14159265j,  0.+0.j])
```

`nan` is a special Python constant that means "not a number." Make up some similar examples using `numpy.arcsin(...)` and try them out.

Besides the pointwise array math functions like `numpy.sin(...)`, the `numpy` module provides a number of other pointwise functions that operate on one or more arrays having a common shape and produce an array of the same shape. For example, the function `numpy.maximum(...)` takes any number of arrays as arguments, and produces an array consisting of the maxima of the corresponding elements of the input arrays, as in:

```
>>> a,b
(array([ 3.,  2.,  1.]), array([ 1.,  2.,  3.]))
>>> numpy.maximum(a,b)
array([ 3.,  2.,  3.])
>>> numpy.maximum(a,2.)
array([ 3.,  2.,  2.])
```

Scalars are broadcast to an array, as usual. `numpy.minimum(...)` does the same for minima. There are many more of these, which you can explore by using the numpy help functions. A selection of them will be introduced in later chapters as needed

1.8.5 Computing arrays on a grid

`numpy` arrays of rank 2 or higher are often used in scientific computation to represent fields of values, e.g. temperature on a grid of latitude-longitude points. Array operations together with the function `numpy.meshgrid(...)` provide a versatile way to create and initialize such fields. Suppose we want to set up a rank 2 array to represent the value of a function on a patch of the plane with horizontal coordinate x running from -1 to 1 and vertical coordinate y also running from -1 to 1. Suppose further we wish to represent the function on a grid of values, with 5 equally spaced x values in the horizontal and 3 equally spaced y values in the vertical, making up 15 pairs of points. We can set up arrays for x and y using `numpy.linspace(...)`, but these are rank 1 arrays, so one cannot directly use them to create the required rank-2 array representing the function on the grid. This problem is neatly solved by the function `numpy.meshgrid(...)` which takes any number of rank 1 arrays and returns a set of arrays with rank equal to the number of input arrays. The number of arrays returned is equal to the number of axes passed to `meshgrid(...)`, and

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

each array represents the corresponding axis *broadcast* (i.e. duplicated) to the grid. This is best explained by example. If two axis arrays, say `x` and `y` are input as arguments to `meshgrid`, it will return a pair of arrays, the first of which contains the array `x` duplicated to all rows of a rank 2 array, and the second of which contains the array `y` duplicated to all columns. If `x` has length `n` and `y` has length `m`, the resulting arrays will have shape `(m,n)`. The shape is defined this way because the convention for `numpy` arrays is that the second index ranges over all items in a row and the first ranges over all items in a column, and the `x` axis is taken to correspond to the horizontal (row) dimension. Here is an example of how this works for our grid with 5 points in x and 3 points in y (a 3x5 grid, using the `numpy` indexing convention).

```
>>> x = numpy.linspace(-1.,1.,5)
>>> y = numpy.linspace(-1.,1.,3)
>>> xg,yg = numpy.meshgrid(x,y)
>>> xg
array([[-1. , -0.5,  0. ,  0.5,  1. ],
       [-1. , -0.5,  0. ,  0.5,  1. ],
       [-1. , -0.5,  0. ,  0.5,  1. ]])
>>> yg
array([[-1., -1., -1., -1., -1.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 1.,  1.,  1.,  1.,  1.]])
>>> xg.shape ; yg.shape
(3, 5)
(3, 5)
```

Then, if we want to represent the function $f(x, y) = x^2 + y^2$ on the grid, we can simply write

```
>>> f = xg**2 + yg**2
>>> f
array([[ 2.  ,  1.25,  1.  ,  1.25,  2.  ],
       [ 1.  ,  0.25,  0.  ,  0.25,  1.  ],
       [ 2.  ,  1.25,  1.  ,  1.25,  2.  ]])
```

The same can be done for any expression built up of `numpy` array operations and array functions. `meshgrid` can also be used with more arguments to create meshes of higher rank, for representing fields defined on a three dimensional, four dimensional or even higher dimensional space.

Array operations are extremely fast. To get an appreciation of just how fast they are, compute the function $f(x, y) = \sin(x)\cos(y)$ on a 1000x1000 grid with each of x and y ranging from 0 to π.

1.8.6 `numpy` array reduction functions

`numpy` provides several *array reduction* functions which operate on `numpy` arrays and produce lower-rank arrays (or scalars, which are rank zero). The function `numpy.sum(...)` is typical of the array reduction functions. It is like the built-in Python `sum` function, but works much faster when used with `numpy` arrays, and also can work with arrays of arbitrary shape. For a rank 1 array, `numpy.sum(...)` works just like the built-in `sum(...)` does for any list or other iterable. For higher rank arrays, `numpy.sum(...)` by default sums all the elements to produce a single value. Here is an example for a rank-2 array:

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

```
>>> A
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
>>> numpy.sum(A)
36
```

For arrays of higher than rank 1, `numpy.sum(...)` and other array reduction functions can take an optional `axis` keyword argument, which specifies which axis (index) of the array the reduction operation is to be performed over. If this argument is specified, the operation produces a rank $n-$ array from a rank n array. For example, this is how we can add up the columns of the array `A` of the previous example:

```
>>> numpy.sum(A,axis=0)
array([ 6,  8, 10, 12])
```

To add up the rows, we would use `axis = 1` instead. the `axis` keyword argument can also be a tuple of integers, in which case the reduction is carried out over all the corresponding axes.

There are a variety of other array reduction functions that work similarly to `numpy.sum(...)`. For example `numpy.average(...)` works basically the same way, but computes an average instead of a sum; it can also take an optional keyword argument `weights` that allows for non-uniform weighting of the data going into the average. Other reduction functions in `numpy` include `cumsum, prod,amax, amin` and `argmax`. These will be introduced in future chapters as needed, but if you are curious about them right now, you can use the `numpy.info(...)` function to get descriptions and examples of usage.

1.8.7 numpy array methods

1.8.8 numpy array cross-sections

`numpy` array cross sections work much the same way as the cross-sections already describe for list indices. The syntax for describing the cross section desired is identical to that used for lists, but a cross section can be used in place of an index in any axis of a higher rank array. Cross sections allow you to easily manipulate subarrays. Whereas the use of an unadorned semicolon as an array cross-section has limited utility for a list or rank 1 array, for higher rank arrays it can be very valuable for extracting slices (e.g. rows or columns) of an array. The following example shows how cross-sections can be used to refer to the first row or column of a 3x3 array that has been previously defined:

```
>>> M
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> M[0,:]
array([0, 1, 2])
>>> M[:,0]
array([0, 3, 6])
```

To pick out a 2x2 subarray in the upper left corner of `M` we could do

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

```
>>> M[0:-1,0:-1]
array([[0, 1],
       [3, 4]])
```

In one important regard, cross sections for numpy arrays are different from cross sections for lists. `numpy` cross-sections create `views` (or references) rather than new copies. This is a deliberate design feature, since in many mathematical operations it is convenient to use array cross-sections to manipulate some portion of a larger array. The following example illustrates that `numpy` array cross sections just create an alternate way to refer to the original data:

```
>>> a = numpy.zeros((2,2))
>>> a
array([[ 0.,  0.],
       [ 0.,  0.]])
>>> b = a[0,:]
>>> b[0] = 1.
>>> a
array([[ 1.,  0.],
       [ 0.,  0.]])
```

However, for cases in which one really wants a cross section to be a new copy which will not affect the original data, `numpy` provides an array method `.copy()`, which creates a copy of the array for which the method is invoked. If, in the above example, the array cross-section were created using the line `b = a[0,:].copy()`, then setting `b[0]` would not affect `a`. Try this out and see. The `.copy()` method can be used on any `numpy` array, cross-section or not, whenever a copy is needed so that operations can be performed without affecting the original array.

Cross sections can be used with array arithmetic and array functions to compute many useful derived quantities without the need for writing loops. For example, suppose the array `T` is a rank-3 array representing a ten year time series of monthly mean surface temperature fields on a latitude-longitude grid, extending over many years, with the first index representing the month number (ranging from 0 to 119) and the next two indices representing the positions in the latitude-longitude grid. Thus, `T[0,:,:]` would be a rank-2 array yielding a temperature map for the first January, `T[1,:,:]` would be the first February, `T[12,:,:]` would be the second January, and so forth. We could then compute the mean temperature map for all the March months in the data set with the single line

```
MarchMeanT = numpy.average(T[2::12,:,:], axis = 0)
```

As another example, suppose we are given a function `f(...)` which (like `numpy.exp(...)`) operates element-wise on numpy arrays, and we wish to tabulate it on a grid of unevenly spaced points from $x = 0$ to $x = 1$ and estimate the derivative on that grid. This could be done as follows

```
n=10
x = (numpy.linspace(0.,1.,n))**2 #Yields an unevenly spaced grid from 0 to 1
F = f(x)
dx = x[1:n]-x[0:(n-1)]
dF = F[1:n]-F[0:(n-1)]
dFdx = dF/dx
xmid = (x[1:n]+x[0:(n-1)])/2.
```

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

The final line defines the array of midpoints, where the derivative has been estimated. Try this out for some function whose derivative you know, and see how well the computation converges to the right answer as n is increased.

Array cross sections can also be used to do matrix row and column operations efficiently. For example, suppose A is a matrix. Then a row reduction and column reduction for row i and column reduction for column j can be done using the statement

```
A[i,:] = A[i,:] - 2.*A[0,:]
A[:,j] = A[:,j] - 3.*A[:,0]
```

1.8.9 Boolean and conditional operations with numpy arrays

All the usual Boolean operations can be carried out on one or more numpy arrays having identical shape and produce numpy Boolean arrays of the same shape corresponding to the Boolean operation being carried out for corresponding elements of the input arrays. The usual comparison operators < ,>,== and so forth can be used with arrays, but for Boolean conjunctions on arrays one must use & instead of the keyword and, | instead of the keyword or, and instead of the keyword not.

The following provides an example of a Boolean array operation, carried out with a 3x3 float array A:

```
>A
array([[ -4. ,  -8. , -12. ],
       [  0.5,   1. ,   1.5],
       [  5. ,  10. ,  15. ]])
>mask = (A>0.)&(A<7.)
>mask
array([[False, False, False],
       [ True,  True,  True],
       [ True, False, False]], dtype=bool)
```

The Boolean operation has created a boolean array of the same shape as the original array A.

Boolean array operations provide a powerful means of carrying out various conditional operations involving arrays. A boolean array can be used as the index of a numpy array (i.e. the thing put inside the square brackets), and if so used creates an array cross-section consisting of all the elements of the indexed array for which the corresponding element of the boolean array are True. The boolean array used as the index must have the same shape as the array being indexed. The ability to use a boolean array as the index of a numpy array is a good example of the general philosophy of object-oriented programming that is at the heart and soul of Python. An index is not regarded as a static inflexible sort of thing that must be done using sequences of integers. Rather, "indexing" is an abstract action that any object can potentially perform, and apart from the specification that indexing is represented by something inside square brackets following the name of an object, it is up to the object's designer to determine how the indexing is done, and if indeed the object is to be indexable at all.

Building on our previous example, if we wanted to reset all of the elements of A satisfying the conditional to the value 1e5, we could do

```
>A[mask] = 1.e5
```

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

```
>print A
[[ -4.00000000e+00  -8.00000000e+00  -1.20000000e+01]
 [  1.00000000e+05   1.00000000e+05   1.00000000e+05]
 [  1.00000000e+05   1.00000000e+01   1.50000000e+01]]
```

It is not necessary to first create a mask array before carrying out an operation of this sort. Expressions like `A[A<0.] = 0.`, in which the boolean array is created in-place, also work. The boolean expression used for the index can involve arrays other than the one being indexed, so long as they all have the correct shape.

The use of boolean cross-sections to modify selected elements of an array works in a fairly intuitive way, but it takes a bit of thinking to understand what is actually going on. If `mask` is a boolean array, then the array cross-section `A[mask]` created by the indexing operation is actually a rank-1 array consisting of the selected elements of `A`, regardless of the rank of `A`. (Try printing out `A[mask]` in the preceding example to verify this.) The reason an expression like `A[mask] = 0.` or `A[mask] = B[mask]` works the way we want it to is that the array cross-section created is a *reference* (also called a *view*) into the original data in its place in the array `A`, rather than a copy. Because it is a reference, not a copy, when we modify the cross-section, we modify the original data as well. The ability to do operations like this is one of the rationales for the default behavior of `numpy` cross-sections being references rather than copies.

Boolean cross-sections can be used together with `numpy` array reduction functions to perform complex computations in a single step. For example the sum of all integers $n < 1000$ such that $n^2 + 1$ is divisible by 5 can be computed by the statements

```
Ns = numpy.arange(1000,dtype='int')
numpy.sum( Ns[(Ns*Ns+1)%5 == 0] )
```

As another example, let's suppose we have a rank-2 array `T` giving the global surface temperature on a latitude-longitude grid, and that the boolean array `LandSea` contains the value `True` for points that are on land and `False` points that are on the ocean. This is typical of a class of computations that often arises in data analysis. Then, the mean temperature of all ocean points is just

```
numpy.average(T[~LandSea])
```

This average is not actually the most physically relevant one, since it does not take into account the fact that a latitude-longitude cell has less area near the poles than near the equator, but that could easily be fixed by specifying the appropriate area-dependent weighting using the optional `weights` keyword argument of `numpy.average(...)`.

You can also do a broad class of conditional operations on arrays using the function:

`numpy.where(`*BoolArray*`,`*Array1*`,`*Array2*`)`.

This function takes three arguments all having the same shape, and returns an array of the same shape as the arguments. The first argument, *BoolArray* is a Boolean array. For each element of this array, `numpy.where(...)` returns the corresponding element of `Array1` if the Boolean value is `True`, or the corresponding element of `Array2` if the Boolean value is `False`. Either *Array1* or *Array2* can be replaced by a scalar value, which will be broadcast to the correct shape of array. Here is a simple example that zeroes out the value of a function being tabulated when one gets too close to a singularity

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

```
>>> x = numpy.linspace(-1.,1.,11)
>>> x
array([-1. , -0.8, -0.6, -0.4, -0.2,  0. ,  0.2,  0.4,  0.6,  0.8,  1. ])
>>> f = numpy.where(numpy.abs(x)<.3, 0. , 1./x )
__main__:1: RuntimeWarning: divide by zero encountered in divide
>>> f
array([-1.        , -1.25      , -1.66666667, -2.5       ,  0.        ,
        0.        ,  0.        ,  2.5       ,  1.66666667,  1.25      ,  1.])
```

Note that a warning error is generated, because the entire array `1/x` is computed before it is decided which elements are actually used to make the output array `f`. The operation nonetheless completes successfully with the desired output, since the error message is only a warning, and the computation of `1/x` just generates a `nan` ("not a number") value when the argument is zero, and that value is discarded by `where(...)`.

The features of `numpy.where(...)` that distinguish it from the use of boolean cross-sections are that it creates a new array and fills in each and every element with one of two alternatives, based on the mask value. If neither of these features is needed in a conditional operation, it is usually better to use a boolean cross-section instead.

1.8.10 Linear algebra with `numpy` arrays

1.9 Conditional Blocks

We have already seen some examples of conditionals in the context of list comprehension and array processing. It is very common that programs need to do different things when various different conditions are encountered – a process known as *branching*. Many of these circumstances require a more general form of branching than we have encountered so far. This is handled by Python's conditional block construction. The general structure of a conditional block is illustrated in Fig. 1.1. In it's simplest form, the conditional block consists of an *if* statement which includes a Boolean expression (i.e. one that evaluates to `True` or `False`) after the `if` keyword, followed by a block of code to be executed if the conditional is `True`. If the conditional is `False`, the block of code is ignored, and program execution proceeds to the next executable statement following the skipped block.

How does Python know what lines are to be associated with the `if` block ? Here we see an important general feature of Python for the first time: Indentation is not just an optional feature you use to make your code look pretty. Rather, it is an essential part of the syntax, and is used by Python to group together statements that belong to a given program unit. Either tabs or spaces can be used for indentation, but it is better not to mix the two and in either case all the statements in a given program block must line up at the same indentation. Program blocks may, however, contain subunits demarcated by further indentations, and levels of indentation can be nested to any depth required.

Conditional blocks are of most use within loops or functions, so most of our examples will be deferred to the next two sections, but still you can get a feel for how conditional blocks work by typing the following examples into a file and executing them for various values of `x`. The following is an example of a conditional block in its simplest form:

```
import math
```

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

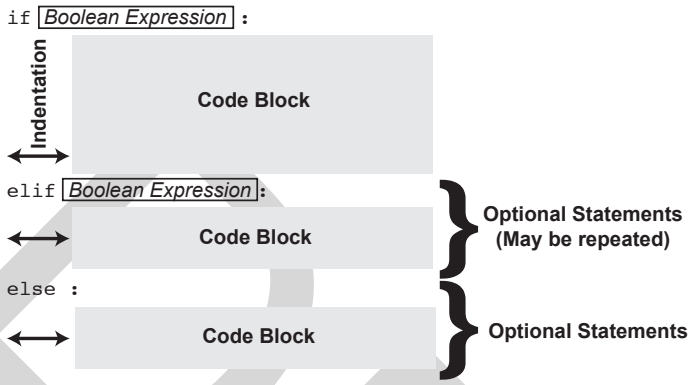

Figure 1.1: General structure of a conditional block

```
x = 2.
answer = 'Not Computed'
if x >= 0. :
     answer = math.sqrt(x)
print answer
```

If you execute this file with the value of `x` as written, it will output `1.4142...`, but if you change the value of `x` to a negative number, it will instead output the phrase `Not Computed`.

An arbitrary number of `elif` (short for "else if") clauses can optionally be added after the `if` clause, and these conditionals will be queried in sequence if the first conditional fails, and the first `elif` block for which the conditional is `True` will be executed, after which control will pass to the next Python statement following the entire conditional block. If no conditional is `True` in any block, control passes to the subsequent Python statement without any conditional block being executed. Type the following into a file and execute it with various values of `x`:

```
x = 1.5
if x < 0.:
     print "It's negative"
elif (x>0.) & (x <= 1.):
     print "It's positive but less than unity"
elif x > 1.:
     print "x is greater than unity"
print "Done"
```

What value of `x` do you have to put in to get the `Done` statement alone?

With or without intervening `elif` clauses, a conditional block can optionally be terminated by an `else` clause, which is executed if the `if` and all `elif` clauses evaluate to `False`, but is skipped if any of them evaluate to `True`. Try executing the following for various different values of `x`:

```
x = 1.5
if (x>=0.) & (x<=1.):
     ResultString = "In Unit Interval"
```

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

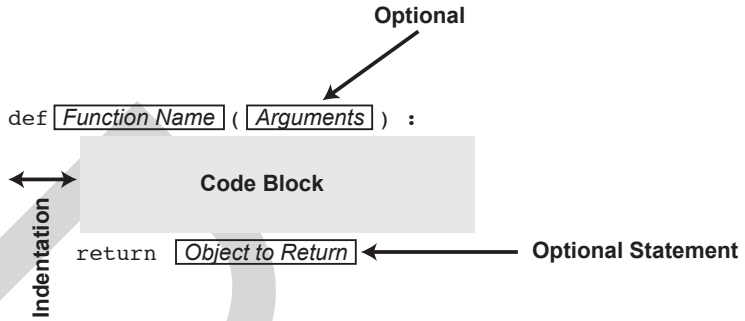

Figure 1.2: General structure of a function definition

```
else:
     ResultString = "Not in Unit Interval"
print ResultString
```

None of these examples do anything particularly useful, but they serve to illustrate the basic operation of conditional blocks. We will see more useful applications of conditional blocks in the context of functions and loops.

1.10 Defining Functions

1.10.1 Basic structure of a function definition

You have already learned how to call a function that somebody else has provided. In this section, you will learn how to define your own functions. This is done using the `def` statement. The general structure of a function definition is given in Fig. 1.2. The function name is the name you choose for the function, and will be the name used to call it. The argument specification shown within parentheses is optional, since there are many cases in which a function might not need arguments; we have seen some of examples of this already, in connection with string and list methods. When the argument specification is included, it consists of a sequence of Python object names separated by commas. There are ways to specify default values for arguments, and more general ways to specify argument names, but we'll get to that later. The `return` statement is optional. If it is present, the expression following the `return` keyword defines the object to be returned as the value of the function when it is called; a Python function can return any Python object. If the `return` statement is absent, the function returns the default value `None`. Just as with conditional blocks, Python uses indentation to group together lines of code that belong to the function definition.

Here is a simple example of a function definition. To define a function called `square`, which returns the square of its argument, you would write

```
def square(x):
     return x*x
```

Here is how the example above would look when defined and then used in the plain Python command line:

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

```
>>> def square(x):
...    return x*x
...
>>> square(2.)
4.0
```

When typing the function definition into the command line interpreter, the function body is terminated by entering a blank line, with no spaces or tabs. The blank line is not needed if the function definition is stored in a file which you execute later. In the plain Python command interpreter you must remember to put in the required indentations yourself, but the `ipython` command interpreter auto-indents for you, and knows when indentation is needed. Note that executing the function definition only defines the function; it does not cause any of the functions operations to be carried out.

Functions can return multiple results, as in:

```
def powers(x):
      return x,x*x,x*x*x
```

This returns a tuple containing the three values. It can be very nicely used with Python's ability to set multiple items to corresponding items of a tuple or list, using constructions of the form:

```
x1,x2,x3 = powers(2)
```

If instead, you wanted the function to return a list that could be modified, the return statement would have to be written as

```
     return [x,x*x,x*x*x]
```

Functions can have as many arguments as you like, the arguments can be any Python objects that the function is set up to handle, and the body of the function can contain multiple lines, which can be any valid Python statements. The following function creates an x-axis ranging from `xmin` to `xmax` sampled with `npts` equally spaced points, and returns both the axis and the values of a function with name `MyFun` evaluated on the axis. A function like this would be useful for making a graph of the input function `MyFun`

```
def FunctionTable(xmin,xmax,npts,MyFun):
     x = numpy.linspace(xmin, xmax,npts)
     return x,MyFun(x)
```

Try it out with the function `square` defined earlier, and also with some `numpy` array functions like `numpy.sin(...)`.

1.10.2 Functions have a private namespace

Functions have their own private *namespace*, and variables defined within a function, and which do not appear in the argument list, do not affect the values of similarly named variables outside the function. Consider the following function, which uses temporary variables, `A` and `B` within the body of the function.

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

```
def f(x,y):
   A = 1. + 1./x
   B = 1. + 1./y
   return A/B
```

The following illustrates how the private namespace works for the function `f`.

```
>>> A = 'A string'
>>> B = 'Another string'
>>> f(1.,2.)
1.3333333333333333
>>> A
'A string'
>>> B
'Another string'
```

Setting the values of `A` and `B` within the function has no effect on the values of objects of the same name outside the function.

1.10.3 Use of conditionals in functions

Conditionals have many uses within functions. The following example shows how conditionals can be used to make a continuous piecewise linear function $f(x)$ with $f(x) = 0$ if $x < 0$, with $f(x) = 1$ if $x > 1$, and with linearly interpreted values in between:

```
def f(x):
    if x<0.:
        return 0.
    elif x>1.:
        return 1.
    else:
        return x
```

This also illustrates the use of multiple returns within a function.

Since Python functions do not themselves impose any constraints on the types of the arguments that are passed to the function, another common use of conditionals is to check for the types of the arguments and act accordingly. This allows functions to be very versatile, since it avoids the need to keep track of a different function for each type of argument that one might want to operate on. Let's suppose, for example, that we want a function to return a `numpy` array that is the sin of the input array, but that we want to allow the input to be either a list of floats, or a `numpy` array of real numbers. The function needs to not only check if the input is a list, but also needs to check the contents of the list to see if they are of the right type. The following function definition does the trick.

```
def f(X):
    #Check to see if X is a list
    if type(X) == list:
        #Check to see if all elements are floats
```

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

```
        if not (False in [type(x)==float for x in X]):
            return numpy.sin(numpy.array(X))
        else:
            return 'Error: Wrong type'
    #Check to see if input is a numpy array
    elif type(X) == numpy.ndarray:
            #Check to see if it is float
            if (X.dtype == 'float64')|(X.dtype == 'float128'):
                return numpy.sin(X)
            else:
                return 'Error: Wrong type'
    else:
        return 'Error: Wrong type'
```

If the arguments are of the wrong type, the function returns the string `'Error: Wrong type'`. Note the use of list comprehension with Boolean expressions, to do the check to see if all the elements of the list are of the right type. This example also illustrates that conditionals can be nested.

There are many other was to use conditionals in functions, which will be developed in subsequent chapters.

1.10.4 More about arguments

In a function definition, a default value can be given for some arguments, in which case the corresponding argument is optional when the function is called, and the default value will be used if the argument is omitted. For example, given the following function definition

```
def f(x,coeff = 1., power = 2.):
     return coeff*(x**power)
```

calling the function as `f(2.)` will return `2**2`, calling it as `f(2.,5.)` will return `5.* 2.**2` and calling it as `f(2.,5.,3.)` will return `5.*2.**3`. The only way a function knows which optional argument you have included is by its position in the argument list, so optional arguments must come at the end of the argument list. Similarly, if one wants to specify a non-default value for some optional argument towards the end of the list, all the intervening values must be specified whether or not one wanted the default value. For example, to compute `2**3` one would need to call the function as `f(2,1,3)` even though the default value for `coeff` was wanted.

Sometimes one wants a function to be able to handle an arbitrary number of arguments, as in the built-in function `max(...)` . This is handled by using the token `*`*ArgList* in the argument list, where *ArgList* can be any name you want. The sequence of arguments will then be accessible within the function as a list with that name. For example, the function

```
def f(x,*powers):
     return [x**n for n in powers]
```

will return a list of however many powers of `x` are specified. The function could be called as `f(2.,1.)`, `f(2.,1.,1.5,2.,2.5)` and so forth. Try out this function, and make up a one of your own.

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

A more flexible way to handle optional arguments is to use *keyword* arguments. These are specified by name when the function is called, as described in Section 1.4. To allow a function to handle keyword arguments, you use the token ***kwargs* in the argument list of the function definition, where *kwargs* can be any name you want. In this case, the object referred to by *kwargs* will be a dictionary rather than a list, and the dictionary keys are the names of the arguments passed as keyword arguments and the corresponding entries are the values specified. The function in our example of positional optional arguments could be handled using keyword arguments as follows:

```
def f(x,**OptArgs):
    a,n = 1., 2. #Set default coefficient and power
    if 'power' in OptArgs:
        n = OptArgs['power']
    if 'coeff' in OptArgs:
        a = OptArgs['coeff']
    return a*(x**n)
```

Note that in order to make the keyword arguments optional, we have used a conditional to check if the keyword is in the set of dictionary keys before looking up the corresponding value, since a dictionary lookup with a nonexistent key raises an error message. With the function set up this way, it could be called in any of the following ways.

```
f(2.)
f(2.,power=5) ,f(2.,coeff = -1.)
f(2.,coeff=-1.,power=5),f(2.,power = 5, coeff = -1.)
```

Note that although the keyword names given in the argument list are turned into strings to be used as keys in the dictionary the function gets, the keyword names given in the argument list are not themselves enclosed in quotes. Define the function and try these calls out, and make sure you understand the results returned by the function.

1.10.5 Side-effecting arguments (Proceed with caution)

Functions can return an output in exchange for an input (the arguments), but they can also process or change things in the argument list, a process known as "side-effecting" the arguments. Generally speaking, it is good programming style to avoid side effecting arguments, in favor of returning results, because promiscuous use of side-effecting makes it harder to keep track of what is an input and what is an output. However, there are times when side-effecting is more convenient than returning multiple results, which must be then put into the right places. If you do write a function that is intended to side-effect its arguments, it is important to keep in mind that Python functions work only on a *copy* of the arguments. In consequence, any changes made to these arguments ("side-effects") do not affect the variable's value in the calling program. Consider the two function defined as follows:

```
def f(myValue):
     myValue = 0
def g(myList):
     myList[0] = 'Changed!'
```

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

The first function does not actually change the value of the argument called in the main program, because it is only a *copy* of the argument which is set to zero; the copy is then discarded when the function exits, for example,

```
>>> A = 1.
>>> f(A)
>>> A
1.0
```

Such a function is useless, and does not achieve the desired side-effect in Python (there are other computer languages for which such a function would work). But consider the following example of the use of the function `g` to side-effect a list passed as an argument:

```
>>> L = [0,1,2]
>>> g(L)
>>> L
['Changed!', 1, 2]
```

In this case the argument is a name which points to the location of some data. The data pointed to can be modified in the function, even though the name is only a copy of the original name passed as an argument, it still points to the same data. The general lesson here is that if you want a function to side-effect an argument, the argument must be the name of a mutable Python object, and the function must modify the data contained in (i.e. pointed to by) the object, rather than modifying the name of the object itself.

Here's a somewhat more useful example, which illustrates the way side-effecting can be used, but which also illustrates some of the pitfalls. Let's suppose we are implementing an algorithm which frequently requires that all the counts in a list of integers be incremented. To do this, it is handy to write a function that does the increment. If we tried to write the function this way:

```
def bump(L):
     L = [n+1 for n in L]
```

it would fail to work, since the body of the function attempts to replace the argument by a new list,but in fact only succeeds in replacing a copy of the argument, which is then discarded when the function exits. (Try defining this function in the interpreter, the execute it on a list of integers to convince yourself that it really fails to change the input list). To do the side-effecting properly, we would need to modify the *contents* of the input list instead, as in:

```
def bump(L):
     L[:] = [n+1 for n in L]
```

which would be equivalent to writing `L[0] = L[0] + 1`, and so forth. The reader would be quite right in concluding that it would be better to just avoid the confusion altogether by using a `return` rather than a side-effect, as in:

```
def bump(L):
     return [n+1 for n in L]
```

which could then be used to replace the original list through an assignment statement `L = bump(L)`. It is generally good programming practice to avoid side-effecting when a `return` would do as well, but there are cases when side-effects can be more convenient, and sometimes even more efficient

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

1.10.6 Lambda functions

1.10.7 Oh,the things you can do with functions!

Being able to define your own functions greatly increases the versatility of list processing using Python's built-in functions `filter`, `reduce`, and `map` which operate on lists.

To be continued ...

The organization of almost any program can benefit from the subdivision of the labor of the program into a number of functions. This makes the program easier to debug, since functions can be tested individually. It also allows the re-use of code that is needed in many different places in the program. Because each function has its own private namespace, functions allow you to introduce variables needed for a calculation without accidentally altering the value of those variables used elsewhere. Finally, since an objects methods are essentially functions, writing functions is one of the critical tasks carried out when designing an object.

1.11 Loops

A *loop* is a programming construction that tells the computer to do something over and over again until some condition is met and it's time to stop. Most scientific simulations – indeed most software of any type – are built around loops. List comprehension involves a form of loop, and while many programming tasks can be performed with list comprehension or array arithmetic, sometimes you need something more versatile and general. Python provides two kinds of loops that serve this purpose: the `for` loop and the `while` loop. Their general structure is given in Fig. 1.3.

1.11.1 `for` loops

Like a list comprehension, a `for` loop loops over elements drawn in sequence from a list or any other iterable object (such as a dictionary, string, or numpy array). In many programming languages, typical practice is to loop over a range of integer values, and this can be done in Python by looping over `range(...)` , but it is seldom necessary to do this in Python. Usually, it's better to just loop directly over the elements you want to deal with. Suppose we want to evaluate the polynomial

$$p(x) = a_0 + a_1 x + a_2 x^2 + ... + a_n x^n \tag{1.1}$$

This can be efficiently done using the iteration

$$p_0 = a_n, p_{j+1} = x \cdot p_j + a_{n-j-1} \tag{1.2}$$

run until p_n has been computed, which is the value of the polynomial. `[a0,a1,...,an]` then the iterative evaluation can be performed using the loop

```
p = A[-1]
for a in reversed(A[:-1]):
    p = x*p + a
```

When this loop completes, `p` contains the value of the polynomial. Note the use of the built-in function `reversed(...)` to make the loop run through the items in the list in reverse order. This

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

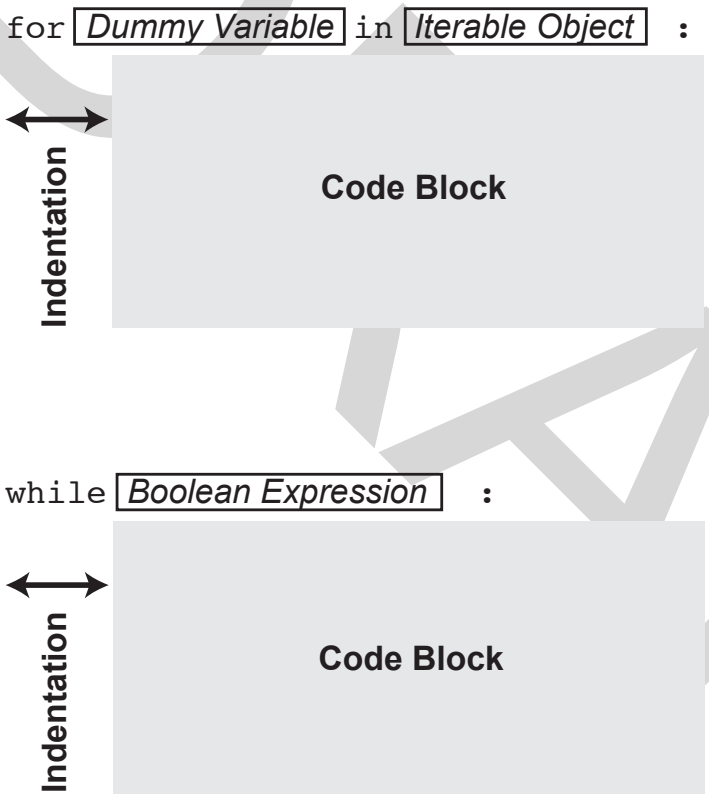

Figure 1.3: Structure of a `for` loop (top) and a `while` loop (bottom).

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

is better than using the list method `A.reverse()` which would reverse the list `A` in place, so that we would need to make a copy of `A` first so as not to affect the original list; moreover the array would have to be reversed *before* using it in the loop, since the `.reverse()` method reverses the array in place rather than returning a reversed array. This example also illustrates the use of a list cross-section to drop out the last coefficient, which need not be processed, since it has already been used to set the inital value of `p`.

When a mathematical expression explicitly involves an integer index, it makes sense to loop over the index. Suppose we want to compute the following n-term Taylor series approximation to e^x

$$f_n(x) = \sum_{i=0}^{n-1} \frac{x^i}{i!} \tag{1.3}$$

where $i!$ is the factorial of i, i.e. the product of the integers up to and including i. This can be computed very compactly using a list comprehension using the statement

```
fn = sum( [x**i/math.factorial(i) for i in range(n)] )
```

which is nice because it corresponds very closely to the mathematical definition of the expression. If efficiency is not an issue, this is a reasonable solution since it produces easily understandable code. However a lot of objections can be raised to this way of doing the computation, on efficiency grounds. It requires all the terms to be stored before the summation is done, and also requires i multiplications of x and i multiplications for the factorial to compute each term, whereas each term can be computed from the previous using just a single multiplication. There are actually ways to make the list comprehension more efficient using the general *iterator* construction described in Chapter **??**, but let's look at a way to do an efficient computation using a `for` loop. The following `for` loop makes use of a recursion inside the loop to eliminate the storage overhead and reduce the amount of computation needed:

```
fn = 0.
term = 1.
for i in range(n):
    fn += term
    term *= (x/(i+1))
```

This assumes `x` and `n` have been previously defined somewhere. Recall that an expression like `term *= a` is shorthand for `term = term*a`. You should avoid the temptation to use the name `sum` for the container used to accumulate the result (`fn` in the example) since that will redefine the Python built-in function `sum`, preventing you from using it later. Python will not prevent you from doing that.

It is generally not good Python practice to introduce an integer index if it is not really needed. As with list comprehensions, the built-in functions `enumerate(...)` and `zip(...)` can often be used to avoid the introduction of an index in circumstances when an index would make the code more cluttered and harder to follow. However there are times when you basically just want to tell Python to "do it `n` times", and the cleanest way to do this is to simply use a loop of the form `for i in range(n):` even if the thing being done in the body of the loop does not need to refer to the index `i`.

`for` loops can also be useful for processing lists of non-numeric data. For example, suppose `L` is a list of strings, each of which contains a molecule name followed by its molecular weight

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

separated from the name by a space, e.g. 'H2O 18'. A list of this sort might be the result of reading in information from a text file, as described in Section 1.14. Then the following loop would set up a dictionary mapping molecule names to the corresponding molecular weight

```
MolDict = {} #Create an empty dictionary
for data in L:
    MolName,MolWt = data.split() #Split the string at the space
    MolDict[MolName] = float(MolWt) #Convert to float and put in dict
```

After running this loop `MolDict['H2O']` would return `18.0` and so forth. Strings are fairly simple objects, but the ability of Python to loop over lists of arbitrary objects is very powerful, since objects can have many attributes performing complex calculations, which can be invoked within the loop. Recall also that loops are not restricted to lists. You can loop over anything that's iterable, and that includes `numpy` arrays, dictionaries, strings and a variety of other objects.

1.11.2 `while` loops

A `while` loop is used to carry out an operation as long as specified following the `while` keyword is satisfied (i.e. evaluates to `True`). As an example, let's write a loop to compute a list of Fibonacci numbers the first two Fibonacci numbers are defined to be `0` and `1`,and each subsequent Fibonacci number is obtained by adding the previous two. The following loop does the trick:

```
FibNums = [0,1] #Initialize the list with the first two
while FibNums[-1] < 1000:
    FibNums.append(FibNums[-1]+FibNums[-2]) #Add the last two to get the next one
```

If you run this code fragment, the last element of the list of numbers computed, `FibNums[-1]` will be `1597`, because the loop terminates the first time that the Fibonacci number `exceeds` 1000. If you wanted the list to only contain values less than 1000, the easiest thing to do would be to discard the final value using `FibNums.pop()` after the loop

`while` loops are very commonly used in scientific simulations of the way a system evolves in time. The general scheme is to set some initial values, repeatedly carry out a procedure specifying how to change the state of the system from one time to the next, and then stop when some condition is met. Sometimes one only is interested in the final state of the system, but more often, it is desirable to save the full time series of results along the way. The following example uses a loop to approximately compute the altitude of an object thrown upward with a specified initial velocity, subject to a constant (negative) acceleration a due to gravity. It saves the results in lists for further use (e.g. plotting). The computation stops when the object reaches the ground.

```
#Initializations
a = -10. #Value of acceleration (constant)
v = 100. #Initial velocity
y = 0.   #Initial altitude
t = 0.   #Initial time
dt = .01 #Time step (constant)
tList = [ ] #List to save time values
vList = [ ] #List to save velocity values
yList = []  #List to save altitude values
```

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

```
#Initializations done. Now do the computation
while y>=0.:
    vList.append(v)  #Save velocity
    yList.append(y)  #Save altitude
    tList.append(t)  #Save time
    v += a*dt #Update velocity
    y += v*dt #Update altitude
    t += dt   #Update time
```

As is typical for `while` loops, the loop computes one more value than you really want, since it stops the first time `y` goes negative. Without use of additional conditionals, this is unavoidable, since a `while` loop cannot test the condition for terminating until the calculation needed to carry out the test has been completed. Most of the time this is harmless, since one can just discard the final value. In this example, the final value is automatically discarded, since the data is saved to the output lists *before* the values are updated

1.11.3 Uses of conditionals within loops

Conditionals are often needed within loops to perform some operations only if the right conditions are met, or to branch to different versions of a calculation depending on the values of some of the quantities used in the calculation. In our simple simulation of the trajectory of an object thrown upward, we might for example, only want to save every hundredth result. This could be done by introducing a counter and conditional as follows

```
#Initializations
...
count = 0 #Initialize a counter
while y>=0.:
    if count%100 == 0:
         vList.append(v)  #Save velocity
         yList.append(y)  #Save altitude
         tList.append(t)  #Save time
    count += 1
    ... #Do the calculation
```

As another variant of this simulation, we might suppose that the object was thrown upward over water, that the object is positively buoyant when under water (yielding positive acceleration when $y < 0$) and that we want to track the object's position while underwater as well as above water. In this case, we not only need a conditional in the loop, we also need to set a different condition for terminating. Let's say we want to track the object until it crosses the surface 5 times. This can be done by modifying the loop as follows:

```
#Initializations
aAir = -10.
aWater = 1.
...
count = 0 #Initialize a counter
SurfaceCrossCount = 0 #Count surface crossings
```

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

```
while SurfaceCrossCount <= 5:
    if count%100 == 0:
         ... #Save results
    count += 1
    if y >= 0:
         a = aAir
    else:
         a = aWater
    yLast = y #Save the previous value of y before calculating the new one
    ... #Do the calculation
    # Determine if we have crossed the surface
    if ((yLast >= 0.) and (y < = 0.) ) or ( (yLast <=0.) and (y>0.)):
        SurfaceCrossCount += 1
```

Try out this loop and have a look at the list of results.

The above examples use `while` loops, but conditionals can be used to good advantage in `for` loops as well. The following example sorts a list of complex numbers into separate lists containing those in the upper half plane, those in the lower half plane, and those on the real axis:

```
zList = [(1+1j)**m for m in range(20)]
Upper= []
Lower = []
RealLine = []
for z in zList:
    if z.imag >0.:
        Upper.append(z)
    elif z.imag <0.:
        Lower.append(z)
    else:
        RealLine.append(z)
```

Try this out and look at which values appear on the real line. Can you come up with a conjecture about which values appear on the real line? Can you prove the conjecture?

Conditionals can be used with the `break` statement to provide multiple pathways for terminating a `for` or `while` loop. When a `break` statement is encountered, control immediately passes to the next statement following the body of the loop, without any further statements in the loop or any further iterations of the loop being executed. The following `while` loop carries out an iteration and stops either when 100 iterations have been done, or when the differences between subsequent iterates are less than 10^{-6} in magnitude, whichever comes first. In either case, it puts a message as to what has happened into the string `message`. In an iteration of this sort, it is necessary to put in a guaranteed exit by capping the number of iterations, otherwise the loop would run forever if the iteration fails to converge.

```
a = 2.
x = 2. #Initial guess
count = 0 #Initialize the counter
while True:
    if count > 100:
        message = 'Max iterations exceeded'
```

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

```
        break
    if count > 0:
        xlast = x #Save previous value
        x -= (x*x -a)/(2.*x) #Update x
        if abs(x-xlast) < 1.e-6:
            message = 'Converged'
            break
    count += 1 #Update count
#Control passes to the following statement after a break
print x,abs(x-xlast),message
```

This example also illustrates how to use a `while` loop to make, in effect, a "do until ... " construction. By using `true` in the `while` clause, the loop would run however long it took until one of the `break` conditions were triggered. The loop implements the Newton's method iteration to solve $x^2 - a = 0$ given an initial guess. Try out the loop as written, then try it with $a = -2.$. Why don't you get convergence in the latter case? Then, with the same value $a = -2.$, try using the initial guess `2. + .1j` for x. Why does the iteration converge in this case?

1.12 When bad things happen to good programs: raise, try and except

Stuff happens, and a well written program should be prepared to handle errors gracefully when they occur, and give the user some informative feedback as to what has happened.

1.13 Classes: Defining your own objects

You have already learned how to use objects that others have defined for you. In this section you will learn some of the basics of how to define and create your own objects.

A new kind of object is defined by the `class` statement. Here is an example of a `class` statement that defines a trivial-looking class of objects with no data or methods:

```
class Dummy:
    pass
```

The `pass` statement is a statement that tells Python to "do nothing." It is used in cases like this where the syntax requires that a statement be present, but we don't actually need the statement to do anything. Every object is an *instance* of a class. To create two objects with the characteristics of the class `Dummy` we would do

```
Instance1 = Dummy()
Instance2 = Dummy()
```

Note that an object is created by "calling" the class as if it were a function. In this case, the creator does not require any arguments, but in other cases the creator is called with arguments. The function `numpy.array(...)` is actually a form of creator for `numpy` array objects.

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

These objects may look pretty useless, but in fact even a bare object like this is very useful in Python, since Python objects can be modified dynamically. That means that new attributes can be defined on the fly, unless the designer of the object has expressly forbidden that. For instance, we can store the radius of a circle and the units in which it is measured in `Instance1` as follows:

```
Instance1.radius = 5.
Instance1.units = 'meters'
```

whereafter the values can be retrieved by name as in the following example:

```
>>>r, units = Instance1.radius,Instance1.units
>>>print 'The area is',3.14159*r**2 ,units
The area is 78.53975 meters
```

This technique is very handy for bundling together data into a single object, to be passed to a function for processing. It is especially powerful since the attributes can contain any Python objects whatever (e.g. `numpy` arrays or functions). Bundling together data into objects helps avoid long, incomprehensible argument lists for your functions.

That's already pretty useful, but objects can do much more than act as bundles.

To be continued ...

1.14 Input and Output

1.14.1 String operations useful for I/O

Text files are read in or written out as sequences of strings, so I/O of text very commonly involves manipulating strings. Even if you are just putting some results out to the screen using the `print` statement, it is often useful to display a string that you have built from the results, since this gives you more control over how the results are presented.

It is very often useful to be able to build strings from numerical values in your script. This need often arises in formatting printout of results to look nice, and also sometimes arises when building the names of files you wish to open. Suppose `a = 2` and `b = 3`. Then, the following example show how you can insert the values into a string:

```
>>> s = '%d + %d = %d'%(a,b,a+b)
>>> s
'2 + 3 = 5'
```

The string contains a number of format codes consisting of a letter (`d` in this case) following a `%` sign, which convert the numbers 'input' to the string to text according to the format specification (decimal integer in this case). The 'input' to the string is a tuple following the string, separated from the string by a `%` sign. The first item in the tuple is given to the first format code in the string, the second is given to the second format code, and so forth. Note that the "input" to the format string must be a `tuple`, not a list; recall, however, that if `L` is a list, the function call `tuple(L)` will return a tuple whose elements are those of the list input as the argument. If the tuple has only one element, you can leave off the parentheses. The format code `%d` (or equivalently `%i`) converts an

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

integer into a string. You use `%f` for floating point numbers, and `%e` for floats in scientific notation. There are other options to these format codes which give you more control over the appearance of the output, and also several additional format codes. For example the code `%10.2f` would desplay the float input with two figures to the right of the decimal point, and would take up a total of 10 characters for the display of the number, padding the result with blanks on the left if necessary. (Try out a few conversions like this with `f` and `e` format codes). Format strings of this type are typically used to build lines which would later be written out to a text file.

When text is read in from a file, each line of the file is converted to a string. If the string contains numeric data to be processed, then the items need to be converted to Python numbers in order to do computations with them. The string method `split(...)` is handy for this, and can be used with list comprehension to do the needed conversion very simply. Suppose that the string `Line` has been read in from a file and consists of the characters `'1.0 1.2 -1.5 3.25'`, i.e. several floating point numbers separated by spaces. Then the following code fragment shows how the values can be extracted and used in subsequent computations

```
Values = [float(xChar) for xChar in Line.split()]
mean = sum(Values)/float(len(Values)) #Compute average
x,y,z,w = Values
print 'Average of %f %f %f %f is %f'%(x,y,z,w,mean)
```

Generally speaking, you need to know how the contents of the file are arranged in order to extract the data you want, but it is possible to put in conditionals and `try ... except ...` blocks to allow for some flexibiity in the form of the file.

1.14.2 Writing text data to a file

First you need to open the file. To do this, you use the `open(...)` statement as in the following:

```
MyFile = open('test.out','w')
```

where `MyFile` is the object by which you will refer to the file in your Python program and `test.out` is the name you want the file to have your computer. You can use any variable name you want for this. Similarly, you can replace `test.out` with whatever you want the file to be called. The second argument of `open(...)` says that we want to create a new file, if necessary, and allow the program to write to it.

You can only write a string to a file, so first you have to convert your data to a string, as illustrated in the preceding section. Let's suppose you want to write two numbers `a` and `b` to a line of the file you have opened. You first build the string, and then write it out using the file method `.write(...)`, which is a method of the `MyFile` object you created when you opened the file:

```
outstring = '%f %f\n'%(a,b)
MyFile.write(outstring)
```

The
`n` character put in at the end of the string has the effect of skipping to a new line before any subsquent writes are done to the file. Without that end-line character, subsequent writes to the file would instead go onto the same line. This can be a useful feature if you wanted to build each

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

line of the file with several different `MyFile.write(...)` statements, since you could then just write out the parts of the line one at a time and finish by writing out the string `'
n'`

Now you have everything you need to write space-delimited data to a file. You simply repeat lines like the preceding in a loop until you are done writing out the data you want. When you are done, you need to close the file by executing `myfile.close()`. If you wanted the numbers in each line of the file to be separated by some character other than a space, you would simply change the format string used to build `outstring`. To do tab-delimited data, you would use the special character
`t` instead of a space, between the format codes.

Using what you have learned write a table of the functions $x/(1+x)$ and $x^2/(1+x^2)$ vs. x to a file. You can take a look at the resulting file in any text editor, or load the file into the program of your choice for plotting.

If you need to open several different files for writing, you can do this very compactly using list comprehension and Python's multiple assignment feature. Suppose we want to open three files, with names `data0`, `data1`, and `data2`. This can be done on a single line using

```
file0,file1,file2 = [open(name) for name in ['data%d'%i for i in range(3)] ]
```

1.14.3 Reading text data from a file

To read data from a text file, you open the file as before (but without the `'w'` argument), and then read in the lines of the file as strings, processing each one according to the kind of data it contains. To read a file, the file must, of course, already exist on your computer. The directory in which the file resides is defined as part of the file name you use to open the file, according to the directory naming conventions on the computer you are using. If no directory specifies are given, the file will be looked for in the current working directory.

A single line can be read from a file with identifier `MyFile` using the `.readline()` method, which is called without arguments. The following reads in one line of a file contining space-separated floats, and converts them to a list of floats. It will return an error if the line contains anything that cannot be converted to a float.

```
MyFile = open("test.txt")
buff = MyFile.readline()
items = buff.split()
values = [float(item) for item in items]
```

To read all the lines of a file, you could put the last three lines of the example into a `while` loop which terminates when the string returned by `.readline()` is empty (has length zero). Alternatively, you could read all the lines of the file at once into a list, using `.readlines()`, and then process the resulting list in a `for` loop:

```
NumbersList = [ ]
for line in MyFile.readlines():
     values = [float(item) for item in line.split()]
     NumbersList.append(values)
```

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

If you went to the trouble of reading in data, you probably want to save it in a form convenient for further computations. The above example stores the converted data in a list `NumbersList`, each item of which consists of a list of the values converted from the corresponding line of the file. When reading in data from a file, it is convenient to accumulate results using lists in this way, since you needn't know in advance how many lines are in the file, or how many items in each line. Once the data is read in, it can be converted to `numpy` arrays for further processing if necessary.

1.14.4 Interactive input

1.14.5 Reading and writing binary data

1.15 Graphics in Python

1.16 Organizing your Python projects

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author

DRAFT

Copyright 2014 R.T. Pierrehumbert
All rights reserved. Do not reproduce or distribute without permission of the author