The Guerrilla Guide to Python

by Che Pinto

This Guerrilla Guide® is one of many written for this course through the years, first on using Fortran, and followed by ones covering the use of C, C++, and now Python. Now that you have some experience writing Python programs, it should help you to better understand what is going on "under the hood" when Python interprets your program.

This is *not* intended to be a comprehensive guide to the Python programming language. There are very many such guides available on the web, as a simple web search for any Python language element will show. Instead, this is an introduction to the *structure* of Python – to how it works and to some language concepts which it helps to keep in mind while programming.[1]

1 Expressions and Statements

As a convenient abstraction (one which is provided to us by the operating system) we will consider the memory in our computer as simply an enormous list of binary digits. To keep track of data stored in this list we need to know the *location* of any given datum and it's *contents* (the datum itself). If one had to refer to every location in memory by its numerical address ("add the number stored at location 8283747723 to the number stored at 4239128797 and put the result in location 7512398794") using memory would be an almost impossibly cumbrous process.

A high-level programming language simplifies the bookkeeping by providing a mnemonic name for a location in memory: a *variable*.[2] When we say in Python `foo = 4`, the compiler chooses an unused location in memory and associates it with the text `foo` – much easier for us to remember. `foo` then refers to the contents of that location, wherever it may be. More often than not, people speak of the datum "4" as being stored *in* `foo`, as if `foo` itself was a container.[3] This is a useful mental picture, though we will see later on that it is also useful to remember (in your subconscious) that `foo` represents an actual location in memory where something is stored.

You can use any name you like for a variable as long as it is composed of strings of upper- and lower-case letters, numbers, and the underscore character (and most importantly for program clarity, no blanks). Unlike in mathematics, one frequently does not use a single symbol for a variable name but rather a descriptive word, *e.g.* `Dipole_moment`.

A program is a set of *instructions* for the computer to take some action. Thus, if we write `my_num = foo`, this is not the mathematical statement that `my_num` is equal to `foo`. Instead, the statement will be translated by the compiler into instructions for the CPU which say, roughly:

[1] You can try out various Python features as you read this guide by using the `ipython` interpreter. Type `ipython` to start it up. Any python expression you type will then be evaluated and you can see the result immediately. Try it! Start `ipython` and type `9+12`. Try executing the code examples given here in ipython.

[2] You may have heard that, in Python, everything is an *object*, and this is indeed the case. An object is a somewhat more sophisticated idea than a variable, and we will postpone discussing objects for a few pages. For now, let us consider a variable simply to be a mnemonic for a location in memory, as it is in many other computer languages.

[3] Again, in Python this is perhaps more true than in some other languages, but we defer discussing things at that level.

1. Take the datum stored in the location in memory associated with foo and stick it in a temporary location within the CPU (known as a *register*).
2. Take the contents of that register, and store it in the location in memory associated with `my_num`.

(Of course, once these instructions have taken place, the contents of `my_num` will equal, in the mathematical sense, the contents of `foo`, but you should consider that a by-product of the operation of transferring the datum). The high-level language thus achieves a significant economy of expression.

The symbol `=` in the above statement is an *operator*; specifically, it is the assignment operator, assigning the value appearing to its right to the variable appearing on its left. The expression to the right can be more complex. We can write `x = foo + 24`. To the machine this might be:

1. Take the contents of `foo` and stick it in a register.
2. Take the number 24 and store it in another register.
3. Add the contents of the first register to that of the second and leave the result in the second register.
4. Take the contents of the second register and store it in the variable `x`.

Our high-level notation seems ever more economical!

Other familiar operators are `-` (subtraction), `*` (multiplication), and `/` (division). Parentheses, `(` and `)`, have just the same function they have in mathematics: grouping operations. Now we can really go to town and write statements like

```
factor = (2*n-1)/(2*n+1)
```

to put the value of the mathematical expression

$$\frac{2n-1}{2n+1}$$

(evaluated with the value stored for n) in the variable `factor`.

2 Expressions and Statements

The group of symbols `(2*n+1)/(2*n-1)` is an example of an *expression*. An expression is a set of instructions which evaluates (perhaps after some considerable amount of work) to a value, a number we haven't yet stored anywhere. A *statement* is an instruction which may or may not leave a value but which is executed for its effects. Thus, if we had previously stored `n = 1`, the *expression* `(2*n+1)/(2*n-1)` evaluates to the value 3. If we execute the *statement* `foo = (2*n+1)/(2*n-1)`, the effect is to store the value 3 in the variable `foo`.

All of the operators we have seen so far are *binary* operators. This is not to say they operate on binary data; rather, they take two values and evaluate to one value. Thus, with 3 previously stored in `n`, the expression `(2*n+1)/(2*n-1)` triggers a sequence of evaluations we can view as

```
(2*n+1)/(2*n-1)
(2*3+1)/(2*n-1)
(6+1)/(2*n-1)
```

```
7/(2*n-1)
7/(2*3-1)
7/(6-1)
7/5
1.4
```

Notice the *precedence* of the operations; things in parentheses are done first, and multiplication is done before addition or subtraction, just as you are used to from mathematics. The assignment operator = has the lowest precedence, so that in `foo = (2*n+1)/(2*n-1)` the expression is evaluated first and the result is finally stored in `foo`. Another useful binary operator is the modulus operator `%`. The expression `a % b` evaluates to the remainder after `a` is divided by `b`. Thus, `13 % 5` evaluates to 3.

Python also has some *unary* operators. The most common is - (negation). It has the obvious action of replacing a value with its additive inverse. It has higher precedence than all of the above binary operators, so that upon executing `foo = 2*-4` the result in `foo` is -8. This is ugly, however, and prone to misinterpretation (if not by the computer then by someone reading the program). Much better to write `foo = 2*(-4)` or even better, `foo = -4*2`. When in doubt about precedence, use parentheses liberally!

Another unary operator is `not`. Since in Python the logical values `True` and `False` are represented by "not zero" and "zero", respectively, the expression `not 3` evaluates to `False` and the expression `not 0` evaluates to `True`. In Python, you don't easily get access to the numerical values representing `True` and `False`, but under the hood this is how they are stored.

3 Types

There are many kinds of data we may wish to store in a variable, and each may require a different amount of memory to hold it. The Python language is a *weakly typed* language. This means that we do not need to specify to the Python interpreter the kind of data we expect to store in each variable we use; Python figures it out (or tries to) for us.[4] For example, if we execute `x=4`, `x` becomes an integer (of type `int`) and holds the integer value 4 in four bytes of memory. If instead we execute `x=4.0`, `x` becomes a floating-point number (of type `float`) which is stored as an 8-byte value.[5]

Python has rules for *type promotion*. For example, if we add two integer values *e.g.* `4+2` the result is an integer, `6`. If we add an integer and a float, the result is a float (even if the float value is, mathematically, an integer value). The same is true for multiplication: an `int` times an `int` is an `int`, and an `int` times a `float` is a `float`. Division is an exception to this pattern; division *always* results in a float, even when both operands are of type `int`.

If you want the usual integer division,[6] Python provides an operator for that, too: `//`. Thus, `8/3` gives the `float` 2.6666666666666665, while `8//3` gives the `int` 2. Occasionally these type

[4] This is an example of where "everything is an object" comes in. Since everything is an object, everything "knows" its own type. Thus, when we use the assignment operator, not only does a value get transferred to the new object, the new object is assigned the appropriate type as well.

[5] Specifically, an IEEE-754 double precision value.

[6] Integer division is defined as producing a result which is the largest integer not greater than the result in floating point. For example, in floating point, $5 \div 2 = 2.5$ while in integer division, $5 \div 2 = 2$. Note, however, that $-5 \div 2 = -3$: -3 is the largest integer not greater than -2.5

promotion rules can cause problems. Often in mathematical formulæ one does in fact want integer division; you need to remember to use the `//` operator in these cases.

You can find the type of an object in Python by using the function `type()`; for example,

```
a = 4; print(type(a))
```

prints "<class 'int'>" Note that we have used the semicolon to "stack" two statements on the same program line.

4 Conditionals

It is useful to be able to specify the conditions under which a certain set of instructions are to be performed. In Python, the syntax for this is the conditional construct `if: else:` Typical usage is

```
if <expression>:
   <set of statements 1>
else:
   <set of statements 2>
```

When the `if` statement is executed, the expression following is evaluated. If it evaluates to `True` (non-zero) then `<statements 1>` are executed, but not `<statements 2>`. If the expression evaluates to `False` (zero) then only `<statements 2>` is executed.

Python does not use punctuation to indicate that a group of statements (often called a "block of code") go together. Instead it uses indentation, so the code blocks which are conditionally executed are indented in the `if: else:` statement one level further than the `if` statement itself

```
if a==b:
    print('probably not')
else:
    print('no')
    print('still no')
print('ok ok yes')

# is easier to read than
if a==b:
 print('probably not')
else:
 print('no')
 print('still no')
print('ok ok yes')
```

It doesn't matter how much indentation is used, but all code at a given level must line up.[7]

There is another set of binary operators which are useful in combination with conditionals: `>`, `<`, `>=`, `<=`, `==`, and `!=`. These all have their obvious meaning, so that the expressions `4 < 7` and `5 <= 5` both evaluate to `True`. Indeed, these are just like any other binary operators, so we can

[7] In the Emacs editor, you can set the offset between indentation levels by setting the variable `python-indent-offset`. You can do this interactively in python-mode by typing `M-x set-variable` and then typing `python-indent-offset`, and then giving the desired value. You can do this once and for all in your `.emacs` file by adding the line:
`(add-hook 'python-mode-hook '(lambda () (setq python-indent 4)))`

write `foo = N < 7`; if the value of `N` is less than 7, then `foo` becomes a variable of type `boolean` and is set to `True`, while if `N` is greater than or equal to 7, `foo` is set to `False`.

The operator == deserves special mention. Why couldn't we simply use "="? Let us examine what happens when we write

```
if foo = 5:          # don't do this!
   <statements 1>
else:
   <statements 2>
```

The = is the assignment operator, and `foo=5` is a *statement*. In Python, a statement doesn't evaluate to anything, so there is nothing for the `if` to operate upon, and we get a syntax error from the interpreter. If, instead, we use the operator ==, then `foo==5` is an *expression* which evaluates to `True` or `False`, values which `if` understands

```
if foo == 5:
   <statements 1>
else:
   <statements 2>
```

If 5 was stored in `foo`, the expression in the conditional evaluates to `True`; if not, it evaluates to `False`, as desired. Note that the `else:` clause is not required. If you simply want to execute something when the conditional expression is true, it may be omitted:

```
if foo == 5:
   <statements 1>
```

For more complex conditional expressions, we have the `elif:` clause

```
if x == 0:
   <statements 1>
elif x>0:
   <statements 2>
else:
   <statements 3>
```

This is equivalent to

```
if x == 0:
   <statements 1>
else:
   if x>0:
      <statements 2>
   else:
      <statements 3>
```

but easier to read. You can have as many `elif`'s as you wish, with or without an `else` at the end.

Python also supports *conditional expressions*

```
a if <condition> else b
```

This expression evaluates to `a` if the condition evaluates to `True` and to `b` if not. We can use this to write statements like

```
y = 1/x if x!=0 else 0
```

Note that in a conditional expression there must be an `else` clause – otherwise, what value would be given to `y` if `x` was zero?!

5 Loops

Another common task is repetition, or *iteration*: repeating a set of instructions one or more times. There are several ways of expressing iteration in Python. The simplest is known as a `for` loop:

```
for <variable> in <sequence>:
   <statements>
```

This code sets the variable to each of the values in the sequence in turn, executing the statements in the indented block once for each value. The sequence must be a Python object known as an *iterable*. For now, let's just say that an iterable is a function which can yield a set of values. One such function is `range([start], stop [, step])`.[8] The code

```
for a in range(3):
   print(a)
# produces
1
2
3
```

(remember, counting in Python begins at zero), while

```
for a in range(2,10,2):
   print(a)
# produces
2
4
6
8
```

We shall see below that many of the Python container objects (like lists) can also function as iterables.

Another way of expressing iteration is `while`:

```
while <expression>:
   <statements>
```

In this case, the expression is evaluated. If `True`, the indented block of statements is executed, and control passes back to evaluating the expression once again. This reads just as in English: while `<expression>` is true, keep executing `<statements>`. Just remember that, on each iteration, the expression is evaluated *before* the statements; if `<expression>` initially evaluates to `False`, then

[8] In descriptions like this, it is common notation to put optional arguments in square brackets. Thus, we can write `range(4)` where `stop=4` and there is no start or step; when not present, `start` defaults to zero, and `step` defaults to one. This yields the sequence `0,1,2,3`. `range(1,4)` yields `1,2,3`, and `range(1,4,2)` yields `1,3`.

`<statements>` will never be executed. A common idiom to keep doing something forever is thus `while True:`

6 Functions

Functions in Python are a shorthand notation for a set of instructions. They are a good way to encapsulate code to allow easy re-use and to make the program you are writing more comprehensible to a human reader. Let us say we want to sum a few terms of the Taylor expansion of e^x around zero,

$$e^x \sim \sum_{i=0}^{\infty} \frac{x^i}{i!}$$

Here is a function to evaluate the first 6 terms ($i = 0, ..., 5$) given a value of x

```
def approx_exp(x):
   """
   Evaluate Taylor series of exp(x) to order 5
   """
   order = 5       # the order of the approximation

   # this loop is the sum over powers of x
   taylor = 0;
   for i in range(0,order+1):

      # this loop computes the factorial of i
      factorial = 1
      for j in range(2, i+1):
         factorial = factorial * j

      # this loop computes the power of x
      power_of_x = 1
      for j in range(1,i+1):
         power_of_x = power_of_x * x

      taylor = taylor + power_of_x/factorial

   return taylor
```

We can use such a function by writing

```
ans = approx_exp(0.05)
print(ans)
# produces
1.0512710963541667
```

The first thing you will notice reading the code is that there are *comments*, either enclosed within triple quotes, `"""` or following the "hash" symbol (`#`). Anything on a line after `#` is ignored by Python; this syntax is provided as a mechanism to annotate the text for the human reader. Likewise, anything between enclosing `"""` and `"""`, even spanning multiple lines, is treated by the compiler as a comment.

If you stare at this code long enough and read the comments, you should be able to see how it works. But see how much easier this would be to read if we write it using some other functions:

```
def approx_exp(x):
   """
   Evaluate Taylor series of exp(x) to order 5
   """
   order = 5

   taylor = 0
   for i in range(0, order):
      taylor = taylor + pow(x,i)/factorial(i)

   return taylor
```

Here `factorial(n)` is an expression which returns the value of the factorial of the value stored in `n`. Likewise, `pow(x,b)` is an expression evaluating to x^b . The `pow(a,b)` function, it turns out, is provided by Python in the `math` module. `factorial` is also provided by `math`, but let us see how to write our own:

```
def factorial(n):
   sum = 1;
   for i in range(2, n+1):
      sum = sum * i
   return sum
```

The first line defines this as a function called `factorial` which will take as an argument a value `n`. To evaluate the expression `factorial(n)`, the computer will go off and perform the instructions contained in the indented block of code, and finally evaluate to the value specified in the `return` statement.

We can write functions with as many arguments as we like. Here is a function which evaluates to the binomial coefficient

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

It is a function of two arguments, and will use the `factorial(n)` function we have already written:

```
def binom(n, k):
   return factorial(n)/(factorial(k)*factorial(n-k))
```

Once again, these are the instructions for how the expression `binom(a,b)` should be evaluated. These instructions require evaluating three factorials; we have provided these instructions by our definition of the `factorial(n)` function above. We can even write functions which take no arguments and/or which evaluate to nothing. Consider

```
void Die():
   exit()
```

This code is executed entrely for its *side effects*; in this case, to terminate the program using the built-in function `exit()`.

Note that the correspondence between names for variables given in the definition of a function and the names used for variables when calling the function is purely positional. As long as the correct values are put in the correct positions, they names of the variables needn't (and rarely will) correspond.

Functions in Python can call themselves, leading to a programming style called *recursion*. A factorial function written recursively might look like

```
def factorial(n):
    return n*factorial(n-1) if n>1 else n
```

We will have more to say about recursion in class.

7 Importing Functions

Most of the definitions for objects in Python are not present in the interpreter by default. They must be *imported* from their respective *modules*. The simplest form a module can take is just another file with Python code in it.

One can import all the functions from a given module, for example those in the `math` module (functions like `sin` and `sqrt`), with the line `import math` before any math functions are used. To use these functions, you need to call them as members of the `math` object: `math.cos(x)`.

If you find the extra effort of typing `math.` before each function annoying, you can instead import functions individually as `from math import sin, sqrt, pi`, or if you want them all, you can write `from math import *` This way, you can just refer to them as `sin` or `sqrt`, without the module name.

Often, however, it makes for more comprehensible code to leave in the module name; this way it is obvious where the function comes from. Some Python module names are just too long for this; having to type `matplotlib.pylab.` before every use of the `plot` function is too much typing and too cluttered. For just such instances, Python allows one to import a module and give it a more useful label: `import matplotlib.pylab as plt` Now, one can simply type `plt.plot` and have the best of both worlds.

Your Python installation has modules stored in a large number of directories and keeps a *search path*, a list of directories to search in when the importation of a module is requested. Normally, this is all set up for you by default, and by default your current working directory is included in this path. After you have been programming for a while, however, you will have developed modules of your own which you will wish to include, and you will not want to have to copy them into whatever directory you are using.

The simplest way to get convenient access to your often-used modules[9] is to create a directory somewhere (I use the hidden directory `~/.Python`) and copy the modules you use frequently into that directory. When you run the Python interpreter, it looks for an environment variable `PYTHONPATH`. If it exists, Python adds the directories listed in it to the default search path. To get Python to search in your directory, then, you need to define `PYTHONPATH` in your Bash shell environment. At the end of the `.bashrc` file in your home directory, add the line

[9]Warning: the instructions given here are for the Unix/Linux operating system. While an analogous process exists for all OS's, the details may well be different. The web search is your friend.

`export PYTHONPATH='/path/to/your/dir'` Now, whenever you start a new shell (a new terminal window), this variable will be defined, and you will be able to import your own code.

To import code, create a file with the desired module name and ending with `.py` For example, suppose you have a file `p305.py` somewhere in Python's search path which contains various function definitions. You can then treat this just as one of Python's built-in classes

```
import p305
z = p305.magic()
```

You can also add to the path within a Python program using the statements

```
import sys
sys.path.import(0, '/my/useful/stuff/directory')
import myusefulstuff
```

8 Objects

We now come to the subject of objects. An *object* is an entity which can store data within itself, just like the notion of a variable above. In addition, however, an object can also contain functions (also called *methods*) which can operate on the object's data. In Python, everything is an object. Even the lowly integer is an object. It not only stores its value, but it knows how convert itself into a character string representation for printing, knows how to adjust its internal representation when it magnitude becomes very large or its type changes, *etc.*

We can think of an "object" like `int` in two ways. The first is the generic object itself – this is a piece of code which can be used many times to make different versions of itself. The second is a specific *instance* of an object. When we type "a = 4" into a Python program, we are *instantiating* a new `int` object, which has 4 as its stored value. Thus, the object `int` is single piece of code of which there can be many instances within your program, each with its own value. Generally, when we speak of "an object" in what follows, we usually mean an instance of an object.

Because everything is an object, Python doesn't really have variables in the way that many other computer languages do. Instead, the "variables" we have been describing above are symbolic names which are bound to specific instances of objects. Indeed, we shall see below that the same object can have more than one name bound to it.

All this may seem unnecessarily complex and formal for someone writing a simple program, but keeping these concepts in mind will help you to understand precisely what is going on in your code, and make it easier to understand more advanced aspects of the Python language.

9 Container Objects

The `int` and `float` objects we have encountered so far each store a single value, but a computer language would not be very useful without the ability to store multiple items of data grouped into collections of some form which can represent abstract objects like lists, vectors, or sets. Python provides a variety of containers for common forms of collections. Since everything in Python is an object, containers are themselves objects for storing and manipulation collections of other objects. This is a very powerful idea as it can easily be generalized to represent quite general and flexible data structures.

Lists and Tuples

One of the most useful containers in Python is the `list`. A list is an ordered collection of objects.[10] Since it can store more than one object, a `list` must be able to keep track of where those objects are stored in the computer's memory, and of how many are stored. A `list` object must, therefore, have internal variables in order to store its data (the set of objects and the number of objects), but we are not granted access to those variables directly. Instead, we can use various functions to manipulate `list` objects.

The simplest form of `list` is an empty one, denoted by the expression `[]` (a list is enclosed by square brackets). We can bind an empty list to a symbolic name as in `foo = []`; generally, we speak of `foo` as *being* that list. We can find the number of elements in a list with the expression `len(foo)` which, in this case, evaluates to 0.

A non-empty list is denoted by a sequence of objects separated by commas and enclosed in square brackets: `letters = ['a','b','c','d']`. We can add an element to the end of a list using the `append` method: `a.append(42)` `append` is a special case of the `insert` method, which takes as arguments the index of the element before which to insert a value, and the value itself. Thus, `a.insert(0,42)` inserts 42 at the beginning of the `list a`, and `a.insert(len(a),'q')` inserts the character 'q' at the end of the list and is thus a synonym for `append('q')`.[11] Lists have additional methods such as finding the maximum and minimum elements, sorting, comparisons, and other functions which take them as arguments, such as `sum` for numeric lists

```
a = [1,2,3,4]
print(sum(a), max(a))
# produces
10 4
```

Another way to manipulate lists is to use the usual mathematical operators on them. This clearly implies redefining the meaning of these operators when applied to lists, an illustration of the concept of *operator overloading*. Operators are, after all, themselves just functions. When we type `2+2` we are in some sense causing Python to run a function which takes two arguments and returns their sum. In the simplest case, this function is designed to take two numerical values of a given type. Overloading the `+` operator is really just generalizing the corresponding function so that it understands what to do with other kinds of input.

We have already seen this in the case of type promotion. The `+` operator invokes a function which, given two `int`s adds them and produces another `int` object as a result. When given an `int` and a `float`, however, the function converts the `int` to a `float`, adds the two floats, and then makes a new `float` object into which it stores the result.

When applied to two lists `a` and `b`, the expression `a+b` *concatenates* the lists, returning a new list whose first `len(a)` elements are the contents of `a` and whose remaining `len(b)` elements are the contents of `b`. Clearly this makes some sense, but as we shall see below (with Numpy arrays), the

[10] An "ordered" list means that the order in which elements appear is important; the list `[a,b]` is not the same as the list `[b,a]`. A *set* is an example of an un-ordered collection. All that matters in a set is membership; the set $\{a,b\}$ is thus identical to the set $\{b,a\}$.

[11] Again, remember that Python counts from zero. Thus, the list `foo=[4,5,3,1,3]` has 5 elements; these are elements 0 through 4, and `len(foo)` evaluates to 5. If we insert before element 5, then, we are inserting at the end of the list.

+ operator can be given a completely different meaning when applied to some other container.[12] Likewise, the multiplication operator $\star$, between a list and an integer n, creates a list containing n copies of the original list: `[1,2,3]*3` evaluates to `[1,2,3,1,2,3,1,2,3]`.

The square-bracket operator is used to access a particular element of a list. If we have the list `a = [1,2,3,4,5]`, then the expression `a[3]` evaluates to 4. Note that we have two different overloadings for the operator `[]`. The first is used to specify a list, the second to extract elements from a list; which one is chosen by the interpreter depends upon how it is used.

As mentioned above, the elements of lists can themselves be any kind of Python object; one particularly useful concept is a list of lists of lists... to represent an N-dimensional collection. If we have a list of lists, *e.g.* `lol = [[1,2,3],[4,5,6],[7,8]]`, then the expression `lol[1]` evaluates to the list `[4,5,6]`. Since `lol[1]` is itself a list, we can apply the square bracket operator again to access an individual element: `lol[1][2]` evaluates to 6. Thus we have a convenient way to store matrices; each row of the matrix is a list, and the matrix itself is stored as a list of these row lists. Thus, if we have the 2x3 matrix `M=[[1,2,3],[4,5,6]]`, the second row-vector is `M[1]`, and the lower-right element is `M[1][2]`.

This illustrates an important idea which we have seen before in mathematical expressions. The expression `1 + (2*3+4)` consists of several sub-expressions, each of which is evaluated in turn, so that the expression `2*3` evaluates to 6, and then `6+4` evaluates to 10, and then `1+10` evaluates to 11. So too, in the example above `lol[1]` is a sub-expression which can then be evaluated by a second application of `[]`. With `a=[1,2,3]` and `b=[4,5,6]`, what would be the result of evaluating `min(a+b)`? Of `sum(2*b)`? Of `sum(2*a) == 2*sum(a)`? Or of `[a,b,a]`?

The indexing operator `[]` also allows for *slicing*; we can specify a range of indices and retrieve a subset of the list. For example, if `a=[7,8,9,10,11]`, then `a[1:3]` evaluates to the list `[8,9]`. Omitting the first index in a slice defaults to zero, and omitting the last defaults to the length of the list. A negative offset for the ending value moves in that far from the end. Specifying a third value gives a *stride* (which may be negative, in which case the slice starts from the end and works backward)

```
a = [1,2,3,4,5,6,7,8]
a[:] -> [1,2,3,4,5,6,7,8]
a[1:3] -> [2,3]
a[:3] -> [1,2,3]
a[4:] -> [5,6,7,8]
a[2:-2] -> [3,4,5,6]
a[::3] -> [1,4,7]
a[::-1] -> [8,7,6,5,4,3,2,1]
a[::-3] -> [8, 5, 2]
```

Slices may also be used for assignment to specific ranges of elements

```
a = [1,2,3,4,5,6,7,8]
a[3:4] = [0,0]      # a is now [1,2,3,0,0,6,7,8]
```

[12] You can even overload operators yourself to make them operate upon your own container objects. We will see a simple example of this below.

```
a[::2] = [0,0,0,0] # a is now  [0,2,0,0,0,6,0,8]
```

The Numpy container (below) will provide even more flexible array slicing.

An important feature of container objects like `list` is that they can be used as *iterables*[13] – as functions which return their contents one-by-one when used in *e.g.* a `for` loop. Thus, we can print out the contents of a list by using the indexing operator `[]`

```
for i in range(len(mylist)):
   print(mylist[i])
```

or, equivalently, we can make use of the iterable property of a list object

```
for x in mylist:
   print(x)
```

When using lists, and especially when returning lists from a function, one can *unpack* the list by specifying a list of names to assign to the elements in the list

```
def bounds(lo, hi):
   delta = 0.1*(hi-lo)
   return lo - delta, hi + delta

xlo, xhi = bounds(0, 10)

el0, el1, el2 = [1,2,3]
```

One can also return a list explicitly, and still unpack the results

```
def returnMany(x):
   return x, x**2, x**3

value, square, cube = returnMany(2)
```

One can even write statements like

```
x, y, z = 0, 0, 1
```

This is more compact on the page than the three lines this would otherwise occupy.

Tuples

The `tuple` is another container object, an immutable (read-only) form of list. Tuples are sequences enclosed by parentheses: `(1,2,3)`. Because of this use of parentheses, we cannot write a 1-tuple (a tuple of length 1) as `(1)`, as this would evaluate simply to the `int` 1. Instead, one must include a comma: `onetuple=(42,)`. An example of this comes from saving a `plot` object in Matplotlib

```
line, = plt.plot(x,y)
```

[13] of which more details will be given below

The comma is necessary since `plot` returns a 1-tuple; putting in the comma unpacks that tuple into a scalar object.

Tuples exist principally because they are faster and consume less memory; they are extensively used in Python for temporary objects. In the unpacking examples, above, Python is actually making tuples, not lists, except where we make a list explicitly.

Strings

A string (the Python `str` object) is a list of characters, with many methods useful for manipulating character strings. The elements of strings are single characters, and are not separated by commas as in regular lists: `foo='hello'`. Because they are like lists, two strings may be concatenated using the `+` operator

```
a = 'hello '
b = 'there'
print(a+b)
# produces
hello there
```

A commonly useful string method is `split()`, which splits a string into a list of sub-strings at a given character (whitespace, by default)

```
sentence = 'this is a sentence'
words = sentence.split()
for w in words:
   print(w)
# produces
this
is
a
sentence
```

One can split on other characters; for example, to parse an IP address, we can write

```
IP = '128.196.208.1'
w = IP.split('.')
```

and `w` will be the list of strings `['128', '196', '208', '1']`.

To print out all of the words in a file (assuming that words are separated by whitespace), each on a separate line

```
for word in open('file.txt').read().split():
   print(word)
```

Here the `open()` function evaluates to an object which represents an open file, its `read()` method acting on it evaluates to a string object, and then `split()` operating on that string object in turn evaluates to a list of strings containing the individual words. We can then use that list as an iterable to obtain the elements of the list (the words in the file) one-by-one.

There are methods to search within strings. For example, `'this is a line'.find('is')` returns 2, the beginning of the first 'is' in the string, while the same line with `rfind('is')` would find the last occurrence.

The inverse of `split` is `join`

```
"-".join(['tra','la','la'])
# produces
'tra-la-la'
```

the sting to the left is inserted between each element of the list argument to `join`, and the result returned as a single string.

Often we will want to convert other types of data into strings, and Python provides several ways to do this. The first is by the `str` function; `str(42)` evaluates to the string `'42'`. The trouble with this is that one has no control over how the string is formatted.

The second way uses the syntax of the C language's `printf()` function (see any web page describing the syntax for printf string formatting)

```
mystring = "The answer to %s is %d" % ('the meaning of life', 42)
```

This was the method introduced in Python2.

Python3 introduced the new string method, `format()`

```
mystring = "The answer to {:s} is {:d}".format('the meaning of life', 42)
```

Curly braces surrounding the format specification are now the placeholders for the value to be formatted, and the format method is given the data to be formatted as arguments. Again, see any web page describing the Python `format` function for more details; Google 'list of Python format characters' to find a list of the possible format specifications.

Sets

The `set` object is a container for collections of objects in which the order does not matter but in which distinct elements are represented only once. The curly-brace operator is overloaded to create sets, so that the RHS of the statement `a={1, 1, 1, 3}` evaluates to the set `{1,3}`. One use for a set would be to count the number of unique words in a text file:

```
setofwords = set()
for word in open('file.txt').read().split():
   setofwords.add(word)
print("Number of unique words:", len(setofwords))
```

While curly brackets are used to specify sets, they are also used for dictionaries (see below), so there is no notation for an empty set; hence one has to use the object name itself to define an empty set. Sets support the usual set operations such as union and intersection, but they do not support an indexing operation – *i.e.* the operator `[]` is not overloaded to support sets. One can add and remove components to/from a set with the `(`add) and `(`remove) methods. Sets also support unions `(`a | b) and intersection `(`a & b). One can test for membership in a set using the `in` operator; *e.g.* `2 in {1,3,5}` evaluates to `False`.

Dictionaries

Python's `dict` object is a somewhat fancier version of `set`, an implementation of *associative memory*. The sequence container objects we have seen so far (lists, tuples, and strings) are addressed by the order in which the elements are stored using the `[]` operator. A `dict` instead stores pairs of objects, a *key* and a *value*: `ages ={'Alexander':26, 'Philip':59, 'Robin':62}`. One can then access the value associated with a given key by using the `[]` operator on the key. In the example, `ages['Philip']` evaluates to 59. Entries can be added to an existing `dict` by declaration: `ages['Ajax']=13`.

This is a bit like having a database indexed on the keys. Drivers license data might be stored in a `dict` as

```
stuff = {}
stuff['Doe'] = ['John', 'NMI', 24, 321321387987421]
stuff['Thomas'] = ['Dylan', 'M', 102, 123213923487213]
stuff['Elliott'] = ['Thomas', 'S', 128, 987456423423429]
stuff['Milne'] = ['Alan', 'A', 134, 029340290325237]
```

We can then look up the drivers license number using the last name: the expression `stuff['Elliott'][3]` evaluates to `987456423423429`.

Dictionaries find frequent use in Python programs. For example, if we wish not only to find all of the words in a text file but also to count their frequency of occurrence, we could modify the code from a previous example to use a `dict`

```
count = {}
for word in open('file.txt').read().split():
   if word in count:
      count[word] += 1
   else:
      count[word] = 1
for word, times in sorted(count.items(), key=lambda x: x[1], reverse=True):
   print("\'%s\' was found %d times" % (word, times))
```

`count` is first defined as an empty `dict`. Then, for each word, we use the `in` operator to test if the dictionary already contains that word as a key. If so, we increment its value; if not, we define a new (key,value) pair with the word as the key and a value of 1. When we have counted all of the words, we then produce a new list, sorted in reverse order by the value in the `dict` of (key, value) pairs. Note the use of `lambda`, which will be covered later in the section on functional programming.

10 Functions, II

Now that we know more about objects, and that everything *is* an object, it should be clear that we can pass functions to other functions. A trivial example is

```
from math import cos, pi, factorial
def tryitout(func, x):
   print(func(x))
```

```
tryitout(cos, pi)
tryitout(atan, 0)
```

Here we pass a function object in the argument `func`, and `tryitout` then executes the function using the parameter `x`.

Another useful bit of function syntax is that function arguments may specify defaults

```
def func( x, y, z, q=47, u=18):
   print(x,y,z)
   print(q, u)
```

In this example, the first three arguments are *mandatory*; to call `fancyfunc` without specifying at least three arguments will cause an error. The next two arguments are optional. If they are omitted, their values default to those given in the function definition. If they are given, then they supersede these values.

The call to a function can also specify arguments by name (known as *keyword arguments*), in which case the arguments don't have to appear in any particular order. Thus, `func(z=c, y=b, x=a)` will produce the same result as `func(a,b,c)`. All keyword arguments, however, must follow all *positional* (*i.e.* non-keyword) arguments, if any. Thus, `func(a, z=c, y=b)` still produces the same result, while `func(x=a, b, z=c)` produces an error.

Sometimes it is convenient to write a function with a variable number of arguments. The `print` function is an example of this; you can give it as many arguments as you like. By placing an asterisk before the last argument in the function definition, it becomes a list which receives any extra arguments beyond the manditory ones (if any)

```
from math import pi, cos, sin, tan
def tryThemOut(x, *funcs):
   for f in funcs:
      print( str(f), f(x) )

tryThemOut(pi, cos, sin, tan)
```

Here, we give three functions to `tryThemOut`, and the argument `funcs` becomes a list of length three; we then loop over the contents of `funcs` to try out each function in turn.

When using extra arguments in this way, note that one cannot specify the mandatory arguments by name, since all positional arguments must come before keyword arguments and `*funcs` is a positional argument.

Another use of the operator `*` in this context of functions is to take a list and expand it into arguments when the function is called

```
def func( x, y, z, q=47, u=18):
   print(x,y,z)
   print(q, u)

foo = [x,y,z,stuff,nonsense]
func(*foo)
```

Note that we are using `*` when we *call* `func`, not in the definition of `func`. This is the same as giving five separate arguments to `func`, each of which is a single integer. It is not the same as

`func(foo)`, which tries to pass one argument, a single list with five elements. This would, of course, give an error, since `func` takes at least three arguments.

Finally, one can write a function with the signature `func(a,b,**c)`.[14] In this case, in addition to the two positional arguments, one can specify an arbitrary number of keyword arguments (which must still come after the positional ones). These are passed to the function as a `dict` container (a container which stores key:value pairs to be described in the next section) where the name of the argument is the key and its value is the value

```
def example(**args):
   for a in args:
      print(a, args[a])

example(first_arg=25, second_arg=42)
# produces
first_arg 25
second_arg 42
```

Similarly, one can call a function with the `**` operator to unpack a `dict` into keyword arguments

```
def func(x,y,z):
   print(x,y,z)

somedict = {'y':2, 'x':1, 'z':3}
func(**somedict)
# produces
1 2 3
```

equivalent to passing all three arguments as keyword arguments.

Python built-in functions, and many from other modules, have what is known as a *doc string*, a string which describes how the function is used, called `__doc__` Try looking at the doc string for the `sum` function

```
print(sum.__doc__)
```

You can add a doc string to your own functions by adding a multi-line comment just after the `def` line:

```
def doEverything(*args):
   """
   doEverything does everything it can to do everything

   usage: doEverything(allArguments)
   """
   <lots of code>
```

[14] The *signature* of a function is the first line in its definition, describing its arguments.

11 Comprehensions

In mathematics, one frequently sees expressions defining sets which look like

$$S = \left\{x^2 : x \in \mathcal{N}, x < 10\right\}$$

(read "the set of x^2 such that x is a positive integer and $x < 10$"), the set of perfect squares less than 100, or

$$Q = \{n : n \in \mathcal{N}, n < 100, f(n) = 0\}$$

the set of all positive integers less than 100 for which some $f(n) = 0$, or

$$L = \{x_i : x_i/x_{i-1} = \delta, x_0 = 1, x_i < 100\}$$

a logarithmically-spaced grid, or

$$P = \left\{a, b, c : a, b, c \in \mathcal{N}, a^2 + b^2 = c^2, c < 100\right\}$$

the set of Pythagorean triples whose hypotenuse is less than 100. These are just finite sets (collections) of numbers. We can use the facilities of the `list` container to write Python expressions to generate such lists in a manner which looks much like the mathematical notation. For example, the perfect squares less than 100 can be written as

```
S = [ n**2 for n in range(1,10) ]
```

This is known as *list comprehension*, an iteration enclosed by a pair of square brackets which produces a list. It even reads like the math – "the set of n^2 for n from one to nine". Of course this produces a list identical to that produced by

```
S = []
for n in range(1,10):
   S.append(n**2)
```

but because the list comprehension is all in one expression, the comprehension executes in much less time. Moreover, the comprehension is arguably easier to read because of its similarity to the mathematical notation we are used to.

The second example can be written

```
Q = [ n for n in range(100) if f(n) == 0 ]
```

where we have included a conditional as a filter on the elements accepted for the list. The logarithmic grid looks like, after some manipulation

```
L = [ pow(delta, i) for i in range(0,100) if pow(delta,i) < 100 ]
```

and the final example, the Pythagorean triples, is (in a not very efficient implementation)

```
P = [ [a,b,c] for a in range(1,101) for b in range(a,101) \
              for c in range(b,101) if a**2 + b**2 == c**2 ]
```

Here we have a triple loop and are producing a list of lists – a fairly complex operation, but still readable, all on one line, and much faster than a triple `for`-loop.

You have to get nested loops the right way round in list comprehensions. If we have the double loop

```
Q = []
for a in something:
   for b in a:
      if condition(a,b):
         Q.append(f(a,b))
```

we translate this to a list comprehension as

```
Q = [ f(a,b) for a in something for b in a if condition(a,b) ]
```

the loops appear in the same order as they do when using multiple `for` statements.

One can, of course, nest comprehensions to produce lists of lists... For example, we can make a piece of the Hilbert matrix $H_{ij} = (i + j - 1)^{-1}$, or

$$H = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \cdots \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \\ \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \\ & & \vdots & & & \end{bmatrix}$$

with

```
H = [ [1/(i+j-1) for j in range(1,6)] for i in range(1,6) ]
```

Here is the Sieve of Eratosthenes as a list comprehension, producing all primes less than 100

```
primes = [x for x in range(2, 101) if all(x%y for y in range(2, min(x, 11)))]
```

The test at the end makes use of the fact that zero is `False` and the function `all`, which evaluates to `True` if and only if all elements of its argument are `True`.

`join` together with a list comprehension is useful to format a list of numbers

```
def fmttup(lst, fmt):
    ffmt = '{:' + fmt + '}'
    return " ".join([ffmt.format(item) for item in lst])

x = [1.0, 2.0, 3.0]
fmttup(x, '.4e')
#produces
'1.0000e+00 2.0000e+00 3.0000e+00'
```

We can also write comprehensions for sets and dictionaries in just the same way, using curly braces instead of square brackets.

The function `zip` often finds use in iterations and especially in list comprehensions. Given two lists, `a` and `b`, the expression `zip(a,b)` evaluates to the list `[(a[0],b[0]), (a[1],b[1]), ...]`. The same is true for any number of lists as arguments; zipping n lists evaluates to a list of n-tuples. The length of the produced list is that of the shortest list given in the arguments to `zip`.

Thus, if we write

```
a = [1,2,3]
b = [-1,-2,-3]
c = list(zip(a,b))
# c is then
[(1,-1),(2,-2),(3,-3)]
```

then `c` becomes a list of length 3 where each element is a 2-tuple. `list(zip(a,b))` is equivalent to the list comprehension

```
[ [ a[i], b[i] ] for i in range(min(len(a), len(b))) ]
```

We can "unpack" the resulting 2-tuples in the usual way, since `zip` is an iterable

```
a = [1,2,3]
b = [-1,-2,-3]
for x,y in zip(a,b):
   print(x,y)
# produces
1 -1
2 -2
3 -3
```

`zip` really becomes useful when we combine it with the unpacking operator `*` (see the section on functions, above). So, for example, we can undo the effect of `zip` using `zip`

```
x = [1,2,3,4]
y = [1,4,9,16]
data = list(zip(x,y)) # the list [(1,1),(2,4),(3,9),(4,16)]
xvals, yvals = zip(*data)
```

This will give us `xvals` equal to the original `x` and `yvals` equal to the original `y`.

Here is an example of a situation which arises when using Matplotlib. `matplotlib.pyplot.plot` takes two arguments, the x and the y values of the curve one wishes to plot. If, instead, we have a list of `(x,y)` pairs in `xy` that we wish to plot, we can write

```
x, y = zip(*xy)
plt.plot(x,y)
```

The call to `zip` returns a list of length two; its first element is a list of x-values, the second a list of y-values. When we unpack the outer list, `x` and `y` get the appropriate inner lists. This is just transposing a matrix

```
m = [ [1,2],
      [3,4],
      [5,6] ]
list(zip(*m))
# produces
[ [1,3,5],
  [2,4,6] ]
```

Why? The `*` operator unpacks `m` into three 2-tuples: `(1,2)`, `(3,4)`, and `(5,6)`. `zip(*m)` is then equivalent to `zip((1,2),(3,4),(5,6))`, and these three 2-tuples are zipped together to produce the result.

12 Iterables

The process of doing something repeatedly is known as *iteration*. One common form of iteration is taking one item at a time from some collection and then acting on it. We have seen this in previous sections, where the `range` object produced sequences of integers, and a `for` loop took each of these integers in turn and executed its block of code.

The general name for an object like `range` which can be used in an iteration is an *iterable*. Iterables have a method `__getitem__()` which can fetch an element from the iterable object. In fact, the operator `[]` is overloaded so that when you write `x[4]` what actually gets executed is `x.__getitem__(4)`.

Most Python containers are iterables; for example a list is iterable

```
components=['cpu','memory','motherboard']
for item in components:
   print(item)
# produces
cpu
memory
motherboard
```

The astute reader might well ask "how does the iterable know what is the next object to give up in an iteration?" The answer is that it doesn't – it simply knows how to return the requested element. Thus an iterable is said not to have *state*. It doesn't remember the state of the iteration, so it doesn't know what to give back next. The object in an iteration which does have state is known as an *iterator*, and it supplies a method `next()` which gives the next element in the container and then advances an internal counter for the element to give the next time it is invoked. The value of that counter is the state retained by the iterator.

A `for` loop

```
for <variable> in <iterable>:
   <statements>
```

works by making an iterator for the specified iterable, and then calling `next` repeatedly to set the variable until `next` indicates that it has run out of elements to serve. This is all done under the hood, so most of the time you don't need to know about the distinction between iterables and iterators just to use a loop.

We can explicitly make an iterator from an iterable using the function `iter()`

```
foo = [1,2,3,4]
f = iter(foo)
print(next(f))
print(next(f))
print(next(f))
print(next(f))
# produces
1
2
3
4
```

If we were to call `next(f)` once again, we would get an error, since there are no more values to provide – in this sense an iterator can only be used once to go through the contents of its associated container. If we want to iterate through a second time, we need to make a second iterator.

13 Basic I/O

As we have seen throughout this guide, the `print` function prints its arguments to the terminal. One can format this output precisely as we formatted strings, above; simply print the formatted string!

To retrieve input from the terminal, the function `input()` keeps reading from the terminal until a newline character is received (from the user typing the enter key). It then returns a string object containing this data (without the newline). If you give `input` a string as an argument, then this is printed before the reading begins

```
line = input('type three integers (separated by commas): ')
words = line.split(',')
vals = [ int(w) for w in words ]
print('you entered', vals)
```

Note the use of a *list comprehension* in the third line. A list comprehension is just another notation for creating a list by processing an iterable element-by-element; it is both much faster to execute and more compact to write than using a `for` loop together with `append`.

One can gain access to the command line which was used to run your python script using the `sys` module. This provides the `argv` object, which is a list of strings; its first element is the command you typed, and subsequent elements are the arguments you typed when running the command

```
import sys
print('program name: ', sys.argv[0])
for i in range(1, len(sys.argv)):
   print('argument %d was %s' % (i, sys.argv[i]) )
```

Try putting this snippet into some file, say `prog.py`, and run it by typing `python prog.py foo bar`.

To write strings to a file, use the `open(filename, mode)` function to return a file object, where `mode` is one of `'w'` for writing or `'a'` for appending (among several other options). One can then give this file object as a keyword argument to `print` or use the `write` method of the file object itself

```
f = open('file', 'w')
print('first line', file=f)
f.write('second line')
f.close()
```

Individual write operations to a hard disk take quite a long time to initiate (at least compared with the speed of main memory), so the operating system *buffers* file output by writing it to a *buffer* (a region in memory reserved to the operating system), and doing an actual write to the hardware only when this buffer fills. Thus, it is good practice to execute the `close` method of the file object to write any data remaining in the buffer to the disk (known as *flushing* the buffer) before the file is closed. When the Python interpreter exits, all open file buffers are flushed anyway, but if you need

to re-open the file in your code, you need to `close` it first to make sure all the output is written correctly.

To read data from a file, begin by creating a file object attached to the desired file with `mode='r'`: `f = open('file.txt', 'r')` To read the entire file as a single string, simply use the `read` method of the file object: `data = f.read()`. This is not usually desirable, however. First, the file may be large and one doesn't necessarily want to fill memory by reading the file all at once. Second, one typically processes files line-by-line, so getting the data this way in the first place is easier. To do so, we can use the fact that the file object is an iterable. This copies a file to the terminal

```
f = open('file.txt','r')
for line in f:
   print(line)
f.close()
```

It is a good idea to close a file after using it, and Python provides a simple way of doing so automatically

```
with open('file.txt','r') as f:
   for line in f:
      print(line)
```

This way, you are less likely to forget to close the file!

Let's say you need to read a file which has a single integer in the first line, two floating point numbers on the second line, an irrelevant third line, and then four columns of data following. The file might look like this:

```
23
1.000 3.1415926
-----------------------------------------
0.0 0.0 1.0000000000000000 0.0000000000000000
1.0 2.5 0.9510593794077145 0.3090081824816504
2.0 5.0 0.8090278863187741 0.5877702605258085
3.0 7.5 0.5878077395640221 0.8090006559383217
4.0 10.0 0.30905224168284645 0.95104506302846
```

You want the columnar data read into a list with four columns and however many rows are in the file. To read the first few, special, lines, we can use the `readline` method, and then resort to the iterable

```
data = []
with open('data.txt', 'r') as f:
   num = int(f.readline())
   words = f.readline().split()
   f1, f2 = [ float(w) for w in words ]
   f.readline()
   for line in f:
      words = line.split()
      data.append([float(w) for w in words])

print("num:", num)
print("f1:", f1, "  f2:", f2)
for i in range(len(data)):
   print(data[i])
```

14 Classes

We are now familiar with a variety of Python objects, and have seen that many of them have both data and methods which operate on that data. Such objects are are called *classes*, and it is time to show how we can make our own classes in Python. As an example, we will make a simple class to represent a one-dimensional vector (we will find that Numpy provides a much better implementation). At a minimum, our `Vector` class will have to store the vector's components. We will add a few functions as well.

All classes begin with a class definition and the definition of the special function `__init__()`, and so will ours

```
class Vector:
   def __init__(self, v):
      self.v = v
```

This is a pretty minimal class, but it demonstrates that a class is defined by a `class` statement and that all classes must define a function called `__init__`. When we instantiate our class, `v = Vector([1,0,1])`, the `__init__` function is called with the arguments we supplied (in this case we are going to use a list to hold the `Vector`'s components). In many languages, this is called the class *constructor* – it does what is necessary to construct an instance of the class. Notice the `self`; self is an object which holds all of the state of the class – all of the class' data. If we wish to store an object within the class instance, we have to add it to `self`. In addition, we must pass `self` as the first argument to all class methods. So our `__init__` function takes the first argument `self` (mandatory), and `v` which is a local variable within the function; to retain the value of `v` after the `__init__` function exits, we must assign it to `self.v`

Note that while we *must* include `self` as the first argument in the definition of all methods in a class, we *do not* include it when we call the method – Python puts it in automatically.

If we instantiate our class, we can then access the member object `v` in the usual way using the *dot operator* (or class object reference operator)

```
q = Vector([1,2,3])
print(q.v)
# produces
[1,2,3]
```

Let us now add another method to our class, the dot product; we will use it with the admittedly rather cumbersome syntax

```
q = Vector([1,0,0])
v = Vector([0,1,0])
answer = q.dot(v)
```

Here is the code

```
from math import sqrt

class Vector:
   def __init__(self, v):
      self.v = v

   # check for compatibility of argument
```

```
def checkarg(self, q, me):
   if not isinstance(q, Vector) or len(q.v) != len(self.v):
      print("Vector::{:s}: argument must be a Vector of length {:d}"\
            .format(me,len(self.v)))
      exit()

#dot product
def dot(self, q):
   self.checkarg(q, 'dot')
   sum = 0;
   for i in range(len(self.v)):
      sum += self.v[i]*q.v[i]
   return sum
```

We have defined the method (function in the class) `dot` with the obligatory first argument `self`, and a second argument which will be the second vector in the dot product. First we check that the object passed to `Vector.dot` is an instance of the `Vector` class. Since every instance of a `Vector` will have a member `v`, the list which stores its components, we also check that the length of `q`'s `v` is the same as our `self.v`. If the argument passes these tests, we compute the dot product using the components `v` in `q` and the components `v` in `self`, and return the (scalar) result. If not, the dot product is not defined, so we quit.

Let's add a 2-norm method to our class. We will use it as `q.norm()`, and the code is quite simple, since the norm is the square root of the dot product of our vector with itself

```
def norm(self):
   return sqrt(self.dot(self))
```

Our `dot` function took as its argument another `Vector` object, but in this case that other `Vector` is our self; we use `self` to refer to the current instance of the class.

Since the length of a `Vector` is just the length of the underlying list `v`, we can implement a `len()` member function as

```
def len(self):
   return len(self.v)
```

Now we try something more difficult – adding two vectors and returning a new instance of the `Vector` class containing their sum

```
def plus(self, q):
   self.checkarg(q, 'plus')
   result = []
   for i in range(len(self.v)):
      result.append(self.v[i] + q.v[i])
   return Vector(result)
```

After checking for compatibility between the other `Vector` and our self as before, we create a new, empty list, and append the sum of each component to it. Finally, we create a new `Vector` object from this list, and return it.

We would like to be able to use the `print` function to print our `Vector`. We use the trick described in the section "More I/O", below, to allow us to say `print(q)` if `q` is a `Vector`

```
def __str__(self):
```

```
      return self.v.__str__()
```

In essence, the method `__str__()` of the `list` object returns a string form of the list; we enclose it between parentheses, put "vector" before it, and our vectors print much as Numpy arrays do.

Finally, we would like to overload the `+` operator to work on our `Vector`s so that we can write `q+v` rather than the more cumbersome `q.plus(v)` as defined above. When Python encounters the `+` operator between two objects, it looks to see if the object has a method `__add__` defined. If so, it evaluates that function; otherwise, an error results. Our full `Vector` class is then

```
from math import sqrt

class Vector:
   def __init__(self, v):
      self.v = v

   # check for compatibility of argument
   def checkarg(self, q, me):
      if not isinstance(q, Vector) or len(q.v) != len(self.v):
         print("Vector::{:s}: error: argument must be a Vector of length {:d}"\
                   .format(me,len(self.v)))
         exit()

   #dot product
   def dot(self, q):
      self.checkarg(q,'dot')
      sum = 0;
      for i in range(len(self.v)):
         sum += self.v[i]*q.v[i]
      return sum

   # 2-norm is sqrt(v.v)
   def norm(self):
      return sqrt(self.dot(self))

   # length of v
   def len(self):
      return len(self.v)

   # addition of two vectors
   def plus(self, q):
      self.checkarg(q,'plus')
      result = []
      for i in range(len(self.v)):
         result.append(self.v[i] + q.v[i])
      return Vector(result)

   # provide the function which overloads '+'
   def __add__(self, q):
      return self.plus(q)

   # define function to make string representation of a vector
```

```
    def __str__(self):
        return "vector(" + self.v.__str__() + ")"
```

We can now use our class

```
x = Vector([1,0,0])
y = Vector([0,1,0])
q = x + y
print('q:', q, ' |q|: ', q.norm())
print('q dot x:', q.dot(x))
q = q + Vector([-1, -1, 0])
print(q)
# produces
q: vector([1, 1, 0])  |q|:  1.4142135623730951
q dot x: 1
vector([0, 0, 0])
```

We could have overloaded the `*` operator to signify a dot product, but since there are several different vector products, this might have been confusing in practice.

15 Lambda and Functional Programming

In a previous example presented while discussing dictionaries, the `sorted` function required us to provide a function which, given a list, would give us the key on which to sort. We could have defined such a function and written that code as

```
def getmykey(x):
    return x[1]
...
for word, times in sorted(count.items(), key=getmykey, reverse=True):
...
```

Note that the named argument `key` specifies a function. Since in Python everything is an object, we can set `key` to be a function (since everything is an object, functions are objects).

Sometimes we wish to specify a function without going to the trouble to define such an object permanently using `def`. In this case, for example, we will never need `getmykey` again. Python provides a facility to define an *anonymous function*, *i.e.* a function without a name, in-place where it is needed. The syntax for this uses the `lambda` operator

```
lambda <arguments>: <expression to return>
```

Thus, our key-getting function `lambda x: x[1]` takes a list `x` and returns its second element, `x[1]`, since we wish to sort on the values in the `dict`. If we had wished to sort the keys in reverse alphabetical order, we would have written `lambda x: x[0]` since, in a dict, the entries are stored as (key,value) pairs.

Because everything in Python is an object, we can of course give a name to an anonymous function by binding to the name: `sqr = lambda x: x**2`. Now, the expression `sqr(5)` evaluates to 25. This is just as if we had defined

```
def sqr(x):
    return x**2
```

Anonymous functions play a role in *functional programming*. There are many programming languages in which there are no operators, only functions and ways to apply them to lists.[15] Python supports this style of programming. There are several ways to apply a function to a list of data. One way is to use `map(function, list)`, which takes a function and applies it to each element of the list, returning an iterator for the values which result. So, for example, we can write

```
for x in map(lambda x: x**2, [1,2,3,4,5]):
   print(x)
```

and get the first five integers, squared. We can turn the result of a `map` into a list simply by `list(map(lambda x: x**2, [1,2,3,4,5]))`, which evaluates to the list `[1,4,9,16,25]`. There is no need to use a `lambda` here; we could just as well have written the name of a function: `list(map(sqr, [1,2,3,4,5]))`.

Any application of `map` can be written as a list comprehension; for the previous example, we have

```
[x**2 for x in [1,2,3,4,5]]
```

The function provided to `map` can take more than one argument, in which case there must be one list provided for each argument. For example

```
a = [1,2,3,4,5]
b = [-1,-2,-3,-4,-5]
list(map(lambda x, y: x+y, a, b))
```

will evaluate to a list of five zeros. `x` and `y` successively pick off the next element from their respective lists, and the function adds them together. Written as a list comprehension, this would be

```
[x+y for x,y in zip(a,b)]
```

One could also have written this as

```
[a[i]+b[i] for i in range(len(a))]
```

The next functional programming function is `filter(func, list)`, which maps a function onto a list and then returns those values of the list for which function evaluates to true. For example, obtain a list of all the primes less than 101

```
list(filter(lambda x: all(x%y for y in range(2, int(sqrt(x)))), range(2,101)))
```

We have already given this example as a list comprehension.

The next tool from functional programming is `reduce`. The author of Python believes that this

[15] Examples of these are Lisp, Scheme, and Haskell.

function leads to unreadable code, so he makes us include it from the `functools` module. Taste aside, reduce is a key tool of functional programming. If we write `reduce(func, [a,b,c,d])`, `func` is recursively applied to the list as `func(func(func(a,b),c),d)`. This can be hard to visualize, so let's take a concrete example: `reduce(lambda x,y: x+y, [1,2,3,4])`. The path of evaluation is then

```
from functools import reduce
plus = lambda x,y: x+y
reduce(plus, [1,2,3,4])
#becomes
plus(plus(plus(1,2),3),4)
plus(plus(3,3),4)
plus(6,4)
10
```

Thus, `reduce(plus, list)` sums the elements in the list. Of course, we could have done this more simply as `sum([1,2,3,4])`, but `reduce` can used for any function. If we wish to find the minimum of a list `numbers`, we might use `reduce(lambda a,b: if a>b a else b, <list>)`.

Unlike `map` and `filter`, one cannot write a `reduce` operation as a list comprehension.

We can implement a Venn diagram sort of procedure to determine who might eat a particular Roman recipe from three sets with

```
from functools import reduce
EatsPepper = {'Nero', 'Flavius', 'Caesar', 'Claudius', 'Trajan', 'Hadrian'}
EatsBoar   = {'Flavius', 'Augustus', 'Caesar', 'Nero', 'Trajan', 'Tiberius'}
EatsGarum  = {'Tiberius', 'Claudius', 'Caesar', 'Augustus', 'Hadrian', 'Fred'}
tmp = reduce(lambda x, y: x.intersection(y), [EatsPepper, EatsBoar, EatsGarum])
print(tmp)
```

The result is, of course, is `{'Caesar'}`. This could also have been computed simply by forming the intersection of the three sets using the set intersection operator `&`

```
EatsPepper \& EatsBoar \& EatsGarum
```

We can write the `factorial()` function previously used as an example as a one-liner

```
def factorial(n): return reduce( lambda x, y: x*y, list(range(2,n+1)), 1)
```

`reduce` takes an optional third parameter which is placed before the beginning of the iterable second argument in the recursion. Thus, if the iterable has no values, `reduce` will not fail, but simply return the third parameter. In this function, if `n` is less than 2, the range would be empty. Adding the initializer thus gives the correct answers for $0! = 1, 1! = 1$.

Finally, we can write a function which composes functions. The composition of functions f, g, and h is f(g(h(x))). The following is an orgy of anonymous functions!

```
def compose(*functions):
   return reduce(lambda f, g: lambda x:f(g(x)), functions, lambda x: x)

xto30 = compose(lambda x: x**2, lambda x: x**3, lambda x: x**5)
xto30(2)
# produces
1073741824
```

How does this work? The first argument to `reduce` is an anonymous function which takes two functions as arguments and returns their composition, as a function of `x`, as an anonymous function. If we give no arguments at all to `compose`, `reduce` returns the third argument, an anonymous function for $f(x) = x$. If there are some functions given as arguments, then the first is composed with the initializer, giving just the first function. This is then composed with the second, and the result is composed with the third, and so on. The result is itself returned as an anonymous function. In the example, we give as arguments three anonymous functions for x^2, x^3, and x^5, and we name the resulting anonymous function `xto30`, since $(((x^2)^3)^5) = x^{30}$. We then compute 2^{30}, the number of bytes in a gigabyte.

16 Copying Objects

As mentioned above, Python doesn't really have variables. When we say `foo = [1,2,3]`, a list object is created by the expression `[1,2,3]` and then is bound to the symbolic name `foo` by the = operator. After this, a reference to `foo` accesses the underlying object to which it is bound.

For scalar objects, when we write

```
foo = 47
goo = foo
foo = 18
```

the statement `goo=foo` creates a new copy of the value contained in `foo`; `foo` now contains 18, while `goo` still contains 47.

For a more complex object like a container, it is the *reference* to the underlying object which is copied; a new object is not created. Consider the following code

```
foo = [1,2,3]
goo = foo
```

What Python in fact does on the second line is to bind the name `goo` to the very same object that is bound to `foo`, so that now that object has *two* names, so to speak. If we change "an element of `foo`", the corresponding element of `goo` is changed as well – it must be so, since there is really only one object here to change

```
foo = [1,2,3]
goo = foo
foo[1] = 17
print("foo:",foo)
print("goo:",goo)
# produces
foo: [1,17,3]
goo: [1,17,3]
```

The "variable" `goo` is known as a *shallow copy* of `foo`. This behavior usually comes as a rather violent surprise to people used to a more traditional language where "everything is an object" is not taken to such extremes and copies are always *deep copies* (new objects).

One can go a long way programming in Python before one encounters problems from shallow copies, but when it happens, it can be very difficult to understand what is going on without under-

standing this behavior. The key is to remember the distinction between objects (which, actually, have no name), and *references to objects*, the "variable names" we actually use in programming.[16]

Often, a shallow copy is just fine, or even just what we want. Consider passing a `list` as an argument to a function

```
def increment0(x):
   x[0] += 1

q = [1,2,3]
increment0(q)
print(q)
# produces
[2, 2, 3]
```

When `q` is passed as an argument to `increment0`, the variable `x` in the function is a shallow copy of `q` – it refers to the same object as `q`. The function then alters one element in that object, and when the function returns, the object `q` still refers to has been altered. If everything was a deep copy, we would have to write something like

```
def increment0(x): # if only deep copies (NOT Python)
   x[0] += 1
   return x

q = [1,2,3]
q = increment0(q)
```

All of the data in the object passed to the function must now be copied into the new object `x`, even though the function alters only one element. The new object would then be returned, replacing the old one. Not only are we incurring the (possibly large) performance penalty of all that unnecessary copying, but while the function was executing there would be *two* full objects present in memory, doubling the memory resources used by this part of the program.

Often, however, we need to keep a copy of a container around while modifying the original. How, then, does one make a deep copy when necessary? The first way is to explicitly tell the interpreter that we want to copy all of the data using array slicing notation

```
foo = [1,2,3]
goo = foo[:]
foo[1] = 17
print("foo:",foo)
print("goo:",goo)
# produces
foo: [1,17,3]
goo: [1,2,3]
```

[16]The astute reader will undoubtedly ask: If objects don't really have a name, what happens to them when their references are reassigned to another object? The answer is known as *garbage collection*. The Python interpreter (a C program) does, in fact, keep track of every object it creates. When you bind an object to a name, the interpreter increments a counter associated with the object. If an object is bound to multiple names, this counter will have a value greater than one. When you change the binding of a name, the counter of the object it *used* to refer to is decremented. When an object's counter reaches zero, it can no longer be used, and becomes "garbage". Periodically, the interpreter goes through its list of objects, finds all the orphaned objects, and deletes them, returning the resources they were using to the operating system for reuse.

This seems to work well, but it is not a panacea; if you don't pay attention to which objects are involved, you can still get in trouble. Consider a list of lists

```
foo = [[1,2,3],[4,5,6],[7,8,9]]
goo = foo[:]
foo[1][1] = 1000
print("foo:",foo)
print("goo:",goo)
# produces
foo: [[1, 2, 3], [4, 1000, 6], [7, 8, 9]]
goo: [[1, 2, 3], [4, 1000, 6], [7, 8, 9]]
```

What is going on? A list of lists is an object which contains a collection of *references to other objects*. When we copied `goo=foo[:]` we made a copy of the three references stored in `foo`, but those references still refer to the original three list objects! Thus, when we changed an element in one of those lists, the change appears in both `foo` and `goo` once again.

To make a deep copy, we need to do something like this, explicitly slice-copying each sub-list in turn

```
foo = [[1,2,3],[4,5,6],[7,8,9]]
goo = [row[:] for row in foo]
foo[1][1] = 1000
print("foo:",foo)
print("goo:",goo)
# produces
foo: [[1, 2, 3], [4, 1000, 6], [7, 8, 9]]
goo: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

but this gets increasingly complicated the more complex the data structure. To solve this problem, one can simply use the `deepcopy` function

```
from copy import deepcopy
foo = [[1,2,3],[4,5,6],[7,8,9]]
goo = deepcopy(foo)
foo[1][1] = 1000
print("foo:",foo)
print("goo:",goo)
# produces
foo: [[1, 2, 3], [4, 1000, 6], [7, 8, 9]]
goo: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

No matter how complex the object, `deepcopy` will truly make an entirely new copy.

The lesson to be learned from this exercise is *not* to make deep copies of everything; the designers of Python decided to use shallow copies by default for good reasons, as outlined above. On the other hand, it is good to know that `deepcopy` exists when needed.

17 Numpy

The Numpy module is perhaps the most import container for numerical computation. Numpy arrays are much like Python lists, except for the fact that all elements must be of the same type;

since their greatest utility is for numerical computation, this is usually a type like `int` or `float`. The `array` method converts a Python list into a Numpy array

```
import numpy as np
oned = np.array([1,2,3,4], int)
twod = np.array([ [1,2,3], [4,5,6] ], float)
```

One can create new arrays with uninitialized space (filled with whatever random bytes happen to be in the memory allocated for the array) or pre-initialized with zeros or ones. Unlike Python lists, a Numpy array must be given a set of dimensions (known as a *shape*) when it is defined; the shape is a tuple of lengths (when an array id 1D, we can omit the tuple)

```
e = np.empty( (3,4), int )    # 3x4 two-d integer array
o = np.ones( (2,2,3), float ) # 2x2x3 floatarray
z = np.zeros( 10 )            # 1x10 array
```

One can also make a new array with the same shape as another one

```
q = np.zeros( (4,12) )
c = np.ones_like(q)
```

The function `np.arange` is analogous with `range`, but it produces a Numpy array

```
x = np.arange(10, dtype=float)
# produces
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
```

One can fill an existing Numpy array with a constant value using `a.fill(42)`.

Functions exist to create various special matrices can be created. The identity matrix

```
I = np.identity(3)
# gives
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

`eye(dim, k)` produces a matrix with ones on the `k`-th diagonal

```
e = np.eye(5,k=-1)
# gives
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  0.]])
```

One can query the shape of an array by looking at the member tuple `shape`

```
a = np.zeros( (20,10) )
print('first dimension is', a.shape[0])
```

The data type of the array is available as

```
print(a.dtype)
```

Like lists, Numpy has the `in` function to see if an element exists in the array

```
3.14 in q
```

Numpy arrays can be reshaped

```
m = np.array(range(9))
M = m.reshape( (3,3) )
print(M)
# produces
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

Reshaping an array produces a shallow copy – the underlying data still refers to the same object – so the number of elements in the reshaped array must be the same as in the original. Numpy has a `copy` method to provide a deep copy

```
m = np.array([1,2,3,4,5,6,7,8,9])
n = m.copy()
```

There is a special method for reshaping an array of any geometry to a 1D array

```
m = np.array(range(9))
M = m.reshape( (3,3) )
q = M.flatten()
```

In this case, this produces not only a 1D array from `M`, it is the *same* array referred to as `m`.

Numpy arrays use a similar syntax for accessing elements as Python lists, except that multi-dimensional arrays are referenced as `nparray[2,3]` as rather than the `list[2][3]` used for lists. They also can be sliced analogously to Python lists. A special form of slicing creates a shallow copy of a subset of an array. This creates a shallow copy of the central 3x3 elements of a 5x5 array

```
M = np.array([ x+y for x in range(1,6) for y in (1.6) ]).reshape( (5,5) )
c = M[1:4,1:4]
```

Such a copy is called a *view* into the larger array. The 5x5 just created is

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

and the view is

```
array([[2, 3, 4],
       [3, 4, 5],
       [4, 5, 6]])
```

If we set an element of the view (indexing it as a 3x3 array)

```
c[1,1] = -1
```

this changes the corresponding value in the full array

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

As we shall see in a moment, one can do element-wise arithmetic on Numpy arrays; unlike on lists, the + operator acting on Numpy arrays is reserved for the mathematical operation. We need to use `concatenate` to paste two or more arrays together

```
a = np.zeros(3)
b = np.ones(4)
c = concatenate( (a,b) )
```

For multidimensional arrays, there are many ways of doing this. One can specify an axis (dimension) along which to do the concatenation (it defaults to `axis=0`). Thus

```
a = np.array([[1,2], [3,4]])
b = np.array([[-1,-2], [-3,-4]])
c = np.concatenate( (a,b) )
# produces
array([[ 1,  2],
       [ 3,  4],
       [-1, -2],
       [-3, -4]])
```

(concatenating along the 0, or row, direction) while

```
c = np.concatenate( (a,b), axis=1 )
# produces
array([[ 1,  2, -1, -2],
       [ 3,  4, -3, -4]])
```

concatenates along the 1, or column, direction.

One can add an axis with the `np.newaxis` parameter

```
a = np.array([1,2,3])
b = a[:,np.newaxis]
# gives
array([[1],
       [2],
       [3]])
# while
b = a[np.newaxis,:]
# gives
array([[1, 2, 3]])
```

`np.newaxis` will become important when we begin to do arithmetic operations on arrays.

The real utility comes when one wants to do arithmetic on arrays. Numpy arrays allow element-wise operations; for two arrays with the same shape, we can write

```
a = np.array([1,2,3])
b = np.array([3,2,1])
c = a + 3*a*b + 4
# gives c
array([14, 18, 16])
```

The expression is applied to all of the array elements; what is more, these implied loops execute as compiled code, and are thus considerably faster than if they were executed as native Python loops.

If the arrays in an expression are not the same dimension, the smaller-dimensioned array will be *broadcast* (repeated)

```
a = np.array([[1,2,3], [4,5,6]])
b = np.array([2,2,2])
b*a
# evaluates to
array([[ 2,  4,  6],
       [ 8, 10, 12]])
```

Obviously, the relevant dimensions must be commensurate for this not to generate an error.

If the direction to broadcast is ambiguous, Numpy will broadcast along `axis=0`, but this can be overridden with `np.newaxis`

```
a = np.zeros( (2,2) )
b = np.array([-1,2])
# a + b gives
array([[-1.,  2.],
       [-1.,  2.]])
# while a + b[:,np.newaxis] gives
array([[-1., -1.],
       [ 2.,  2.]])
```

Element-wise operations work with array slicing as well. For example, if we have a uniform grid of x values $x_i = x_0 + \Delta i,\ i = 0, \ldots, N$ and a vector of values $y = f(x)$ on this grid, we can compute the centered finite-difference approximation to $y'(x)$ at each of the gridpoints but the edges of the grid (where some of the necessary data does not exist)

$$y'(x_i) \sim \frac{y_{i+1} - y_{i-1}}{2\Delta}$$

as

```
import numpy as np

N = 10
delta = 2
x = delta*np.linspace(1,N,N) # evenly-spaced array
y = x**2
dyexact = 2*x

dy = np.zeros_like(y)
dy[1:-1] = (y[2:] - y[:-2])/(2*delta)

print(dy-dyexact)
# produces
[ -4.   0.   0.   0.   0.   0.   0.   0.   0. -40.]
```

`linspace(start, stop, num)` produces a Numpy array of `num` evenly-spaced values from `start` to `stop` (inclusive), which, multiplied by `delta` is our grid. The function `y` is the element-wise

square of `x`. `dy` is defined to be an array of the same shape as `y`, and then all but the outer elements of the derivative are computed by the finite-difference formula; we can't compute the derivative at the edges since we don't have the data for a centered difference. Since the centered difference is second-order accurate and our function is quadratic, the answer is seen to be exact where computed.

Numpy provides its own versions of the standard mathematical functions which also operate element-by-element so one can compute, for example `np.cos(x)**2 + np.sin(x)**2 - 1` which evaluates, of course, to an array of zeros of the same shape as `a`.

Also provided are reduction operations on arrays, *e.g.* `sum` and `prod`

```
a = np.array([1,2,3])
p = np.prod(a)
# gives
6
```

One can compare arrays, leading to boolean arrays with the element-wise result of the comparison

```
a = np.array([3,4,5])
b = np.empty(3)
b.fill(3)
q = a > b
# gives
array([False,  True,  True], dtype=bool)
```

The functions `np.any(q)` does a reduction of logical "or" over a boolean array, giving `True` for the array just computed, and `np.all(q)` does a reduction with "and", giving `False` in this case.

The function `where` applies a conditional to the elements of an array, choosing between one of two arguments

```
a = np.array([1,-2,3,-4,5])
np.where( a<0, -a, a )
# gives
array([1, 2, 3, 4, 5])
```

Of course, we could have used `np.abs(a)` to achieve the same result in this case.

If one places a boolean array within the indexing operator, element-wise operations will only be performed on elements for which the boolean array is `True`

```
def wraptopi(x):
    pi = np.pi
    x = x - np.floor(x/(2*pi)) * 2*pi
    x[x >= pi]  -= 2*pi
    return x
```

(note that `x >= pi` produces such a boolean array)

We can select elements from an array according to an integer array of indices. For example, we can generate a permutation

```
a = np.array([1,2,3,4,5,6,7])
i = np.array([1,3,5,6,4,2,0])
a[i]
```

```
# gives
array([2, 4, 6, 7, 5, 3, 1])
```

Numpy also implements a wide variety of functions for linear algebra, such as the inner product `np.dot(a,b)` (which is the dot product between two vectors, and the matrix product for matching multi-D shapes) and the cross product `np.cross(a,b)`, or the matrix norm `np.linalg.norm(x)`

The inverse of a matrix is given by `np.linalg.inv(M)`, and one may solve a linear system $Mx = b$

```
M = np.array([[3,1], [1,2]])  # matrix of coefficients
b = np.array([9,8])           # RHS
x = np.linalg.solve(M, b)
```

One can find the eigenvalues and eigenvectors of a matrix

```
import numpy as np

<# get a random matrix
np.random.seed(12931239)
M = np.random.random( (5,5) )

# make it symmetric so we get real eigenvalues
M = M + M.T - np.diag(M.diagonal())

# find eigenvalues and eigenvectors
evals,evecs = np.linalg.eig(M)

# sort into decreasing magnitude of eigenvalues
indices = np.abs(evals).argsort()[::-1]
evals = evals[indices]
evecs = evecs[:,indices]

print(evals)
print(evecs)
# produces
[ 4.05560259 -1.24234045 -0.50253687  0.26123755  0.19037583]
[[-0.29157869 -0.52744272 -0.35151154 -0.61304586 -0.37067566]
 [-0.47654677 -0.35996213  0.77131655  0.17483789 -0.13354   ]
 [-0.52485495  0.43935491  0.0845304  -0.49589508  0.52767151]
 [-0.40703916  0.56869043 -0.14654593  0.19464872 -0.67197208]
 [-0.49672069 -0.27529964 -0.50288102  0.55660257  0.33879629]]
```

Here we have used the Numpy random number generator to construct a 5x5 random matrix. The following line adds `M` to its transpose `M.T` and subtracts its diagonal.[17] We then find the eigenvalues and eigenvectors. Since it is often useful to have these sorted in decreasing order of eigenvector magnitude, we call `np.argsort()`, which returns an array of indices permuted so

[17] `np.diagonal` evaluates to a 1-D array with the diagonal elements of `M`; `np.diag` makes the corresponding diagonal matrix from the array of diagonal elements.

that the corresponding eigenvectors are in descending order of magnitude.[18] Finally, we apply this permutation to the eigenvalues and the second index of the eigenvectors.

This just scratches the surface of what Numpy can do, but it should be enough to give a flavour of how Numpy arrays are used. There are many good Numpy tutorials on the web, and `stackexchange.com` has answers to a wide variety of questions.

18 More I/O

The `print()` function prints its arguments to the terminal. Each Python object has a method to format a string for output, `__str__()`. The `print` function itself then simply calls `__str__()` for each object in its argument list and writes the resulting string to the terminal. For example, the doc string for the `int` class tells us

```
>>> print(int.__str__.__doc__)
Return str(self).
```

We can write such a function for our own classes; for example

```
class Test:
   def __init__(self, pos, vel):
      self.r = pos
      self.v = vel
   def __str__(self):
      return "pos = " + str(self.r) + "\nvel = " + str(self.v)

t = Test([1,0],[0,1])
print(t)
# produces
pos = [1, 0]
vel = [0, 1]
```

Note that we didn't have to format the lists we passed into `Test`; they already know how to format themselves! Applying the function `str` to an object returns the result of its own `__str__` function.

[18] We do this by sorting the reversed vector of magnitudes (stride of -1) into increasing order.