An introduction to Python for scientific computing

Table of contents

Overview

Python is an extremely usable, high-level programming language that is now a standard in scientific computing. It is open source, completely standardized across different platforms (Windows / MacOS / Linux), immensely flexible, and easy to use and learn. Programs written in Python are highly readable and often *much* shorter than comparable programs written in other languages like C or Fortran. Moreover, Python comes pre-loaded with standard modules that provide a huge array of functions and algorithms, for tasks like parsing text data, manipulating and finding files on disk, reading/writing compressed files, and downloading data from web servers. Python is also capable of all of the complex techniques that advanced programmers expect, like object orientation.

Python is somewhat different than languages like C, C++, or Fortran. In the latter, source code must first be compiled to an executable format before it can be run. In Python, there is no compilation step; instead, source code is interpreted on the fly in a line-by-line basis. That is, Python executes code as if it were a script. The main advantage of an interpreted language is that it is flexible; variables do not need to be declared ahead of time, and the program can adapt on-the-fly. The main disadvantage, however, is that numerically-intensive programs written in Python typically run slower than those in compiled languages. This would seem to make Python a poor choice for scientific computing; however, time-intensive subroutines can be compiled in C or Fortran and imported into Python in such a manner that they appear to behave just like normal Python functions.

Fortunately, many common mathematical and numerical routines have been pre-compiled to run very fast and grouped into two packages that can be added to Python in an entirely transparent manner. The NumPy (Numeric Python) package provides basic routines for manipulating large arrays and matrices of numeric data. The SciPy (Scientific Python) package extends the functionality of NumPy with a substantial collection of useful algorithms, like minimization, Fourier transformation, regression, and other applied mathematical techniques. Both of these packages are also open source and growing in popularity in the scientific community. With NumPy and SciPy, Python become comparable to, perhaps even more competitive than, expensive commercial packages like MatLab.

This tutorial will cover the Python 2.7 language version. A newer language version, the 3.0 series, also exists, but breaks compatibility with the earlier versions of the language. The 2.7 series is still widely used for scientific computing efforts in Python.

Installation

To use Python, one must install the base interpreter. In addition, there are a number of applications that provide a nice GUI-driven editor for writing Python programs. For Windows

platforms, I prefer to use the freely available Anaconda installation and the included Spyder editor. This single package includes virtually all tools one would need for scientific Python computing, and can be downloaded at:

https://www.anaconda.com/distribution/

Download the installation executable and proceed through the automated setup. Most of the modules that you will need are pre-installed.

Other resources

Python comes standard with extensive documentation. The entire manual, and many other helpful documents and links, can also be found at:

http://docs.python.org

The Python development community also maintains an extensive wiki. In particular, for programming beginners, there are several pages of tutorials and help at:

http://wiki.python.org/moin/BeginnersGuide

For those who have had some programming experience and don't need to start learning Python from scratch, the Dive Into Python website is an excellent tutorial that can teach you most of the basics in a few hours:

http://www.diveintopython.org/

Interactive interpreter

Start Python by typing "python" at a command prompt or terminal. You should see something similar to the following:

```
Python 2.6.5 (r265:79096, Mar 19 2010, 21:48:26) [MSC v.1500 32 bit
(Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The ">>>" at the bottom indicates that Python is awaiting your input. This is the interactive interpreter; Python programs do not need to be compiled and commands can be entered directly, step-by-step. In the interactive interpreter, Python reads your commands and gives responses:

```
>>> 1
1
```

As we will show later, Python can also read scripts, or files that are pre-written lists of commands to execute in sequence. With the exception that output after each line is suppressed when reading from a file, there is no difference in the way Python treats commands entered interactively and in scripts; the latter are simply read in as if they were typed at the interactive prompt. This gives us a powerful way to test out commands in your programs by entering them interactively while writing code.

Comments in Python are indicated using the "#" symbol. Python ignores everything after them until reaching the end of the line.

```
>>> 1    #I just entered the number 1
1
```

Long commands in Python can be split across several lines using the line continuation character "\". When using this character, subsequent lines must be indented by exactly the same amount of space. This is because spacing in Python is syntactic, as we will discuss in greater depth later.

```
>>> 1.243 + (3.42839 - 4.394834) * 2.1 \
...   + 4.587 - 9.293 + 34.234 \
...   - 6.2 + 3.4
```

Here, Python automatically draws the ellipses mark to indicate that the command you are entering spans more than one line. Alternatively, lines are continued implicitly without using the "\" character if enclosing characters (parenthesis, brackets) are present:

```
>>> (1.243 + (3.42839 - 4.394834) * 2.1
...   + 4.587 - 9.293 + 34.234
...   - 6.2 + 3.4)
```

Typically the use of parenthesis is preferred over the "\" character for line continuation.

It is uncommon in practice, but more than one command can be entered on the same line in a Python script using the ";" symbol:

```
>>> 1 + 4 ; 6 - 2
5
4
```

Avoid using this notation in programs that you write, as it will densify your code at the expense of legibility.

There is a generic `help` function in Python that will tell you about almost everything. For example, it will tell you what the proper arguments for a function are:

```
>>> help(sum)
Help on built-in function sum in module __builtin__:
```

```
sum(...)
    sum(sequence, start=0) -> value

    Returns the sum of a sequence of numbers (NOT strings) plus the value
    of parameter 'start'.  When the sequence is empty, returns start.
```

The `help` function will even work with functions and variables that you create yourself, and Python provides a very easy way to add extra descriptive text that the `help` function can use, as we will discuss later on.

Python is a *case sensitive* language. That means that variables and functions must be given the correct case in order to be recognized. Similarly, the following two variables are different:

```
>>> Var = 1
>>> var = 2
>>> Var
1
>>> var
2
```

To exit the Python interactive prompt, we need to use an end-of-file character. Under Windows, this corresponds to the Ctrl-Z key combination; in Linux, it corresponds to Ctrl-D. Alternatively, one can use the `exit()` function:

```
>>> exit()

c:\>
```

Everything is an object

Python enforces a great democracy: everything in it—values, lists, classes, and functions—are objects. An object comes with multiple properties and functions that can accessed using *dot notation*. For example,

```
>>> s = "hello"
>>> s.capitalize()
'Hello'
>>> s.replace("lo", "p")
'help'
```

We could have used dot notation directly on the string itself:

```
>>> "hello".capitalize()
'Hello'
```

The fact that everything is an object has great advantages for programming flexibility. Any object can be passed to a function; one can send values or arrays, for example, but it is equally easy to send other functions as arguments to functions. Moreover, almost everything in Python

can be packaged up and saved to a file, since there are generic routines that pack and unpack objects into strings.

Basic types

Numbers without decimal points are interpreted as integers.

```
>>> type(1)
<type 'int'>
```

The type function tells you the Python type of the argument given it. Here, the return value in this statement tells you that "1" is interpreted as a Python "int" type, the name for an integer. Normal integers require 4 bytes of memory each, and can vary between -2147483648 and 2147483647.

On the other hand, large integers exceeding this range are automatically created as "long" integer types:

```
>>> type(10000000000)
<type 'long'>
```

Long integers can take on any value; however, they require more memory than normal integers and operations with them are generally slower.

To specify a real number, use a decimal point:

```
>>> type(1.)
<type 'float'>
```

Floating-point numbers in Python are *double-precision reals*. Their limitations are technically machine-dependent, but generally they range in magnitude between 10^{-308} to 10^{308} and have up to 14 significant figures. In other words, when expressed in scientific notation, the exponent can vary between -308 and 308 and the coefficient can have 14 decimal places.

Python can also handle complex numbers. The notation "j" indicates the imaginary unit:

```
>>> type(1+2j)
<type 'complex'>
```

Complex math is handled appropriately. Consider multiplication, for example:

```
>>> (1+2j)*(1-2j)
(5+0j)
```

Note that Python represents complex numbers using parenthesis.

For every type name in Python, there is an equivalent function that will convert arbitrary values to that type:

```
>>> int(3.2)
3
>>> float(2)
2.0
>>> complex(1)
(1+0j)
```

Notice that integers are truncated. The `round` function can be used to round to the nearest integer value; it returns a float:

```
>>> int(0.8)
0
>>> round(0.8)
1.0
>>> int(round(0.8))
1
```

Python as a calculator

Add two numbers together:

```
>>> 1+1
2
```

Integer division truncates the fractional part:

```
>>> 8/3
2
```

Floating point division returns a float, even if one of the arguments is an integer. When performing a mathematical operation, Python converts all values to the same type as the highest precision one:

```
>>> 8./3
2.6666666666666665
```

Exponentiation is designated with the "**" operator:

```
>>> 8**2
64
>>> 8**0.5
2.8284271247461903
```

Note that the following result returns a value of 1 due to integer division in the exponent:

```
>>> 8**(1/2)
1
```

The modulo operator "%" returns the remainder after division:

```
>>> 8 % 3
2
>>> 4 % 3.
1.0
```

Boolean values and comparison operators

Standard operators can be used to compare two values. These all return the Boolean constants True or False.

```
>>> 1 > 6
False
>>> 2 <= 2
True
```

The equals comparison involves two consecutive equal signs, "==". A single equal sign is not a comparison operator and is reserved for assignment (i.e., setting a variable equal to a value).

```
>>> 1 == 2
False
```

The not equals comparison is given by "!=":

```
>>> 2 != 5
True
```

Alternatively,

```
>>> not 2 == 5
True
```

The Boolean True and False constants have numerical values of 1 and 0 respectively:

```
>>> int(True)
1
>>> 0==False
True
```

Logical operators can be used to combine these expressions. Parenthesis help here:

```
>>> (2 > 1) and (5 < 8)
True
>>> (2 > 1) or (10 < 8)
True
>>> (not 5==5) or (1 > 2)
False
>>> ((not 3 > 2) and (8 < 9)) or (9 > 2)
True
```

Variable assignment

Variables can be assigned values. Unlike many other programming languages, their type does not need to be declared in advance. Python is *dynamically typed*, meaning that the type of a variable can change throughout a program:

```
>>> a = 1
>>> a
1
>>> b = 2
>>> b == a
False
```

Variables can be incremented or decremented using the "+=" and "-=" operators:

```
>>> a = 1
>>> a = a + 1
>>> a
2
>>> a += 1
>>> a
3
>>> a -= 3
>>> a
0
```

Similar operators exist for multiplication and division:

```
>>> a = 2
>>> a *= 4
>>> a
8
>>> a /= 3.
>>> a
2.6666666666666665
```

Notice in the last line that the variable a changed type from `int` to `float`, due to the floating-point division.

Strings

One of Python's greatest strengths is its ability to deal with strings. Strings are variable length and do not need to be defined in advance, just like all other Python variables.

Strings can be defined using double quotation marks:

```
>>> s = "molecular simulation"
>>> print s
molecular simulation
```

Single quotation marks also work:

```
>>> s = 'molecular simulation'
>>> print s
molecular simulation
```

The former is sometimes useful for including apostrophes in strings:

```
>>> s = "Scott's class"
```

Strings can be concatenated using the addition operator:

```
>>> "molecular " + 'simulation'
'molecular simulation'
```

The multiplication operator will repeat a string:

```
>>> s = "hello"*3
>>> s
'hellohellohello'
```

The `len` function returns the total length of a string in terms of the number of characters. This includes any hidden or special characters (e.g., carriage return or line ending symbols).

```
>>> len("Scott's class")
13
```

Multi-line strings can be formed using triple quotation marks, which will capture any line breaks and quotes literally within them until reaching another triple quote:

```
>>> s = """This is a triple-quoted string.
It will pick up the line break in this multi-line sentence."""
>>> print s
This is a triple-quoted string.
It will pick up the line break in this multi-line sentence.
```

One can test if substrings are present in strings:

```
>>> "ram" in "Programming is fun."
True
>>> "y" in "facetious"
False
```

Special characters in strings

Line breaks, tabs, and other formatting marks are given by special codes called escape sequences that start with the backslash "\" character. To insert a line break, for example, use the escape sequence `\n`:

```
>>> print "This sting has a\nline break"
This string has a
line break.
```

A tab is given by `\t`:

```
>>> print "Here is a\ttab."
Here is a       tab.
```

To include single and double quotes, use `\'` and `\"`:

```
>>> print "Scott\'s student said, \"I like this course.\""
Scott's student said, "I like this course."
```

Since the backslash is a special character for escape sequences, one has to use a double backslash to include this character in a string:

```
>>> print "Use the backslash character \\."
Use the backslash character \.
```

One can suppress the recognition of escape sequences using *literal strings* by preceding the opening quotes with the character "r":

```
>>> print r"This string will not recognize \t and \n."
This string will not recognize \t and \n.
```

String formatting

Number values can be converted to strings at a default precision using Python's `str` function:

```
>>> str(1)
'1'
>>> str(1.0)
'1.0'
>>> str(1+2j)
'(1+2j)'
```

Notice that each of the return values are now strings, indicated by the single quote marks.

To exert more control over the formatting of values in strings, Python includes a powerful formatting syntax signaled by the "%" character. Here is an example:

```
>>> s = "The value of pi is %8.3f." % 3.141591
>>> print s
The value of pi is    3.142
```

In the first line we included in our string a format specification. We signal the insertion of a formatted value using the "%" character. The numbers that follow it tell the size and precision of the string output. The first number always indicates the total number of characters that the value will occupy after conversion to string; here it is 8. This is not a hard limit for Python, but it uses this number to correctly align up decimal points for multiple string calls.

The decimal point followed by a 3 tells Python to round to the nearest thousandth. The "f" character is the final component of the format specification and it tells Python to display the number as a float. Finally, we have to supply after the string the value to be formatted. Another "%" character sits in between the string with the format specification and the value to be formatted.

One can omit the total length specification altogether:

```
>>> print "The value of pi is %.3f." % 3.141591
The value of pi is 3.142.
```

If a width specification is provided, Python tries to line up strings at the decimal point:

```
>>> print "%8.3f" % 10. + "\n" + "%8.3f" % 100.
  10.000
 100.000
```

One can suppress this behavior and force Python to left-justify the number within the width specification by placing a minus sign "-" immediately after the percent operator :

```
>>> print "%-8.3f" % 10. + "\n" + "%-8.3f" % 100.
10.000
100.000
```

To explicitly show all zeros within the specification width, place a zero after the percent in the format specification:

```
>>> print "%08.3f" % 100.
0100.000
```

Python offers many other ways to format floating-point numbers. These are signaled using different format specifications than "f". For example, exponential notation can be signaled by "e":

```
>>> print "%10.3e" % 1024.
1.0240e+003
```

Integer formatting can be performed using either the "i" or "d" flag. By default, Python truncates numbers rather than rounding when performing this operation:

```
>>> print "%i" % 3.6
3
```

All of these formatting codes work with either floats or ints; Python is smart enough to convert between them automatically:

```
>>> print "%.4E" % 238482
2.3848E+005
```

Multiple values can be converted in the same string. To achieve this, place multiple format specifications followed by a list of multiple values contained within parenthesis and separated by commas. The first format specification will be assigned to the first value, the second to the second value, and so on and so forth.

```
>>> print "I am %i years old and %.3f meters tall." % (30, 1.83)
I am 30 years old and 1.830 meters tall.
```

The group of values after the percent sign is actually a *tuple* in Python, and tuples can be substituted in place of the explicit grouping. We will talk more about tuples shortly.

Strings can also be values in format specifications, included using the "s" flag:

```
>>> print "The frame is %.1f by %.1f inches and %s." % (12, 8, "blue")
The frame is 12.0 by 8.0 inches and blue.
```

If one wants to specify the width of a format specification using the value of a variable, a "*" is used in place of the width value and an additional integer precedes the value to be formatted in the subsequent tuple:

```
>>> a = 10
>>> print "%0*i" % (a, 12345)
0000012345
```

In format specifications, the "%" character is special and needs to be escaped using "%%" if one wants to include it in the string:

```
>>> print "I bought %i gallons of 2%% milk." % 2
I bought 2 gallons of 2% milk.
```

Lists

Python's ability to manipulate lists of variables and objects is core to its programming style. There are essentially two kinds of list objects in Python, *tuples* and *lists*. The difference between the two is that the former is fixed and can't be modified once created, while the latter allows additions and deletions of objects, sorting, and other kinds of modifications. Tuples tend to be slightly faster than lists, but the speed benefit is rarely substantial and a good rule of thumb is to always use lists.

Lists can be created with brackets:

```
>>> l = [1,2,3,4,5]
>>> print l
[1, 2, 3, 4, 5]
```

Long lists can be spread across multiple lines. Here, the use of the line continuation character "\" is optional, since Python automatically assumes a continuation until it finds the same number of closing as opening brackets. It is important, however, that indentation is consistent:

```
>>> l = [1, 2, 3,
...      4, 5, 6,
...      7, 8, 9]
... <hit return>
>>> l
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Two lists can be concatenated (combined) using the addition operator:

```
>>> [1,2] + [3,4]
[1, 2, 3, 4]
```

Notice that addition does not correspond to vector addition, in which corresponding terms are added elementwise. For vectors, we will use arrays, described in the tutorial on NumPy.

To repeat items in a list, use the multiplication operator:

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

The increment operators work similarly for lists

```
>>> l1 = [1,2,3]
>>> l2 = [4,5,6]
>>> l1 += l2
>>> l1
[1, 2, 3, 4, 5, 6]
```

The `range` function automatically produces lists made of a sequence of numbers. It has the form `range(start, stop, step)`, where the first and last arguments are optional. Note that `range` always starts at zero and is exclusive of the upper bound (e.g., the list does not include `stop`).

```
>>> range(4)
[0, 1, 2, 3]
>>> range(1, 4)
[1, 2, 3]
>>> range(0, 8, 2)
[0, 2, 4, 6]
```

The length of a list can be checked:

```
>>> len([1, 3, 5, 7])
4
```

Accessing list elements

List elements can be accessed using bracket notation:

```
>>> l = [1,4,7]
>>> l[0]
1
>>> l[2]
7
```

Notice that the first element in a list has index 0, and the last index is one less than the length of the list. This is different than Fortran, but is similar to C and C++. *All sequence objects (lists, tuples, and arrays) in Python have indices that start at 0.*

An out-of-bounds index will return an error:

```
>>> l[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Individual elements can be set using bracket notation:

```
>>> l = [1,4,7]
>>> l[0] = 5
>>> l
[5, 4, 7]
```

Negative indices can be used to identify elements with respect to the end of a list:

```
>>> l = [1,4,7]
>>> l[-1]
7
>>> l[-3]
1
```

Slices or subsections of lists can be extracted using the notation `l[lower:upper:step]` where `lower` gives the *inclusive* lower element index, `upper` gives the *exclusive* upper index, and the optional `step` gives the increment between the two.

```
>>> l = [1,2,3,4,5]
>>> l[0:4]
[1, 2, 3, 4]
>>> l[0:4:2]
[1, 3]
```

If `lower` is omitted, it defaults to 0 (the first element in the list). If `upper` is omitted, it defaults to the list length.

```
>>> l = [1,2,3,4,5]
>>> l[:4]
```

```
[1, 2, 3, 4]
>>> l[2:]
[3, 4, 5]
>>> l[::2]
[1, 3, 5]
```

Negative indices can be used for list slicing as well. To take only the last 3 elements, for example:

```
>>> l[-3:]
[8, 5, 2]
```

To take all but the last two elements:

```
>>> l[:-2]
[4, 2, 8]
```

In slices, list indices that exceed the range of the array do not throw an error but are truncated to fit:

```
>>> l = [4,2,8,5,2]
>>> l[2:10]
[8, 5, 2]
>>> l[-10:3]
[4, 2, 8]
```

List comprehensions

Python provides a convenient syntax for creating new lists from existing lists, tuples, or other iterable objects. These *list comprehensions* have the general form

```
>>> [expression for object in iterable]
```

For example, we can create a list of squared integers:

```
>>> [i*i for i in range(5)]
[0, 1, 4, 9, 16]
```

In the expression above, elements from the list created by the `range` function are accessed in sequence and assigned to the variable *i.* The new list then takes each element and squares it. Keep in mind that Python creates a *new* list whenever a list construction is called. Any list over which it iterates is *not* modified.

The iterable does not have to be returned by the `range` function. Some other examples:

```
 >>> [k*5 for k in [4,8,9]]
[20, 40, 45]
>>> [q**(0.5) for q in (4,9,16)]
[2.0, 3.0, 4.0]
```

```
>>> [k % 2 == 0 for k in range(5)]
[True, False, True, False, True]
>>> [character for character in "Python"]
['P', 'y', 't', 'h', 'o', 'n']
```

More than one iterable can be included in the same list. Python evaluates the rightmost iterables the fastest. For example, we can create all sublists [j,k] for $0 < j < k \leq 3$:

```
>>> [[j,k] for j in range(4) for k in range(j+1,4)]
[[0, 1], [0, 2], [0, 3], [1, 2], [1, 3], [2, 3]]
```

It is also possible to filter items in list comprehensions using if statements. The general form is:

```
>>> [expression for object in iterable if condition]
```

For example, we could have also written the above list of sublists as:

```
>>> [[j,k] for j in range(4) for k in range(4) if j < k]
[[0, 1], [0, 2], [0, 3], [1, 2], [1, 3], [2, 3]]
```

Here is another example that filters a list for elements containing the letter "l":

```
>>> [s for s in ["blue", "red", "green", "yellow"] if "l" in s]
['blue', 'yellow']
```

Here is a similar example, taking the first character of each string:

```
>>> [s[0] for s in ["blue", "red", "green", "yellow"] if "l" in s]
['b', 'y']
```

List operations and functions

Lists can contain any type of object in Python. They can contain mixed types and even other lists:

```
>>> l = [1., 2, "three", [4, 5, 6]]
>>> l[2]
'three'
>>> l[3]
[4, 5, 6]
```

Multiple indices can be used to access lists within lists:

```
>>> l = [1., 2, "three", [4, 5, 6]]
>>> l[3][1]
5
```

You can test if a value or object is in a list:

```
>>> 3 in [1, 2, 3]
True
```

```
>>> 4 in range(4)
False
```

Elements can be deleted from lists:

```
>>> l = [9,2,9,3]
>>> del l[1]
>>> l
[9, 9, 3]
```

Deletion can use slice notation:

```
>>> l = range(5)
>>> del l[1:3]
>>> l
[0, 3, 4]
```

The first instance of a particular element can be removed:

```
>>> l = [1, 2, 3, 2, 1]
>>> l.remove(2)
>>> l
[1, 3, 2, 1]
```

Items can be added to lists at particular locations using `insert(index, val)` or slice notation:

```
>>> l = [1, 2, 3, 4]
>>> l.insert(2, 0)
>>> l
[1, 2, 0, 3, 4]
>>> l = l[:4] + [100] + l[4:]
>>> l
[1, 2, 0, 3, 100, 4]
```

To create an empty list and add elements to it:

```
>>> l = []
>>> l.append(4)
>>> l
[4]
>>> l.extend([5,6])
>>> l
[4, 5, 6]
>>> l.append([5,6])
>>> l
[4, 5, 6, [5, 6]]
```

The difference between the `append` and `extend` list methods is that `append` will add the argument as a new member of the list, whereas `extend` will add all of the *contents* of a list argument to the end of the list.

List items can be counted:

```
>>> [1, 2, 6, 2, 3, 1, 1].count(1)
3
```

You can find the index of the first instance of a list element. If the element is not in the list, an error is produced.

```
>>> l = [1, 5, 2, 7, 2]
>>> l.index(2)
2
>>> l.index(8)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.index(x): x not in list
```

If a list contains all numeric values, it can be summed:

```
>>> sum([0, 1, 2, 3])
6
```

Lists can be sorted:

```
>>> l = [4, 2, 7, 4, 9, 1]
>>> sorted(l)
[1, 2, 4, 4, 7, 9]
>>> l
[4, 2, 7, 4, 9, 1]
>>> l.sort()
>>> l
[1, 2, 4, 4, 7, 9]
```

Notice that the function `sorted` returns a new list and does not affect the original one, whereas the list function `sort` modifies the original list itself.

The `sort` function can take an optional argument, `sort(cmp)`, where `cmp(x,y)` is a user-defined function that returns -1 if x should preceed y, 0 if x and y are equivalent in order, and 1 if x should follow y. By default, Python uses the built-in `cmp` function if no argument is given:

```
>>> cmp(1,2)
-1
```

The sorting functions work with any type for which the `cmp` function is defined, which includes strings:

```
>>> sorted(['pear', 'apple', 'orange', 'cranberry'])
 ['apple', 'cranberry', 'orange', 'pear']
```

For user-defined types called *classes*, it is possible to *overload* the `cmp` function to tell it how to sort. We will discuss classes in greater detail later.

If list members are lists themselves, sorting operates using the first element of each sublist, and subsequent elements as needed:

```
>>> l = [[5, 'apple'], [3, 'orange'], [7, 'pear'], [3, 'cranberry']]
>>> sorted(l)
[[3, 'cranberry'], [3, 'orange'], [5, 'apple'], [7, 'pear']]
```

Lists can also be reversed:

```
>>> l = [1, 2, 3]
>>> l.reverse()
>>> l
[3, 2, 1]
```

Tuples and immutable versus mutable objects

Tuples are similar to lists but are immutable. That is, once they are created, they cannot be changed. Tuples are created using parenthesis instead of brackets:

```
>>> t = (1, 2, 3)
>>> t[1] = 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Like lists, tuples can contain any object, including other tuples and lists:

```
>>> t = (0., 1, 'two', [3, 4], (5,6) )
```

The advantage of tuples is that they are faster than lists, and Python often uses them behind the scenes to achieve efficient passing of data and function arguments. In fact, one can write a comma separated list without any enclosing characters and Python will, by default, interpret it as a tuple:

```
>>> 1, 2, 3
(1, 2, 3)
>>> "hello", 5., [1, 2, 3]
('hello', 5.0, [1, 2, 3])
```

Tuples aren't the only immutable objects in Python. Strings are also immutable:

```
>>> s = "There are 5 cars."
>>> s[10] = "6"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

To modify strings in this way, we instead need to use slicing:

```
>>> s = s[:10] + "6" + s[11:]
```

```
>>> s
'There are 6 cars'
```

Floats, integers, and complex numbers are also immutable; however, this is not obvious to the programmer. For these types, what immutable means is that new numeric values always involve the creation of a new spot in memory for a new variable, rather than the modification of the memory used for an existing variable.

Assignment and name binding

Python treats variable assignment slightly differently than what you might expect from other programming languages where variables must be declared beforehand so that a corresponding spot in memory is available to manipulate. Consider the assignment:

```
>>> a = 1
```

In other programming languages, this statement might be read as "put the value 1 in the spot in memory corresponding to the variable a." In Python, however, this statement says something quite different: "create a spot in memory for an integer variable, give it a value 1, and then point the variable `a` to it." This behavior is called *name binding* in Python. It means that most variables act like little roadmaps to spots in memory, rather than designate specific spots themselves.

Consider the following:

```
>>> a = [1, 2, 3]
>>> b = a
>>> a[1] = 0
>>> a
[1, 0, 3]
>>> b
[1, 0, 3]
```

In the second line, Python bound the variable b to the same spot in memory as the variable a. Notice that it did not *copy* the contents of a, and thus any modifications to a subsequently affect b also. This can sometimes be a convenience and speed execution of a program.

If an explicit copy of an object is needed, one can use the `copy` module:

```
>>> import copy
>>> a = [1, 2, 3]
>>> b = copy.copy(a)
>>> a[1] = 0
>>> a
[1, 0, 3]
>>> b
[1, 2, 3]
```

Here, the `copy.copy` function makes a new location in memory and copies the contents of a to it, and then b is pointed to it. Since a and b now point to separate locations in memory, modifications to one do not affect the other.

Actually the `copy.copy` function only copies the outermost structure of a list. If a list contains another list, or objects with deeper levels of variables, the `copy.deepcopy` function must be used to make a full copy.

```
>>> import copy
>>> a = [1, 2, [3, 4]]
>>> b = copy.copy(a)
>>> c = copy.deepcopy(a)
>>> a[2][1] = 5
>>> a
[1, 2, [3, 5]]
>>> b
[1, 2, [3, 5]]
>>> c
[1, 2, [3, 4]]
```

The `copy` module should be used with great caution, which is why it is a module and not part of the standard command set. The vast majority of Python programs do not need this function if one programs in a *Pythonic* style—that is, if one uses Python idioms and ways of doing things. If you find yourself using the `copy` module frequently, chances are that your code could be rewritten to read and operate much cleaner.

The following example may now puzzle you:

```
>>> a = 1
>>> b = a
>>> a = 2
>>> a
2
>>> b
1
```

Why did b not also change? The reason has to do with immutable objects. Recall that values are immutable, meaning they cannot be changed once in memory. In the second line, b points to the location in memory where the value "1" was created in the first line. In the third line, a new value "2" is created in memory and a is pointed to it—the old value "1" is not modified at all because it is immutable. As a result, a and b then point to different parts of memory. In the previous example using a list, the list was actually modified in memory because it is mutable.

Similarly, consider the following example:

```
>>> a = 1
>>> b = a
>>> a = []
```

```
>>> a.append(1)
>>> a
[1]
>>> b
1
```

Here in the third line, a is assigned to point at a new empty list that is created in memory.

The general rules of thumb for assignments in Python are the following:

- Assignment using the equals sign ("=") means point the variable name on the left hand side to the location in memory on the right hand side.
- If the right hand side is a variable, point the left hand side to the same location in memory that the right hand side points to. If the right hand side is a new object or value, create a new spot in memory for it and point the left hand side to it.
- Modifications to a mutable object will affect the corresponding location in memory and hence any variable pointing to it. Immutable objects cannot be modified and usually involve the creation of new spots in memory.

It is possible to determine if two variable names in Python are pointing to the same value or object in memory using the `is` statement:

```
>>> a = [1, 2, 3]
>>> b = a
>>> a is b
True
>>> b = [1, 2, 3]
>>> a is b
False
```

In the next to the last line, a new spot in memory is created for a new list and b is assigned to it. This spot is distinct from the area in memory to which a points and thus the `is` statement returns `False` when a and b are compared, even though their data is identical.

One might wonder if Python is memory-intensive given the frequency with which it must create new spots in memory for new objects and values. Fortunately, Python handles memory management quite transparently and intelligently. In particular, it uses a technique called *garbage collection*. This means that for every spot in memory that Python creates for a value or object, it keeps track of how many variable names are pointing at it. When no variable name any longer points to a given spot, Python automatically deletes the value or object in memory, freeing its memory for later use. Consider this example:

```
>>> a = [1, 2, 3, 4]  #a points to list 1
>>> b = [2, 3, 4, 5]  #b points to list 2
```

```
>>> c = a              #c points to list 1
>>> a = b              #a points to list 2
>>> c = b[1]           #c points '3'; list 1 deleted in memory
```

In the last line, there are no longer any variables that point to the first list and so Python automatically deletes it from memory. One can explicitly delete a variable using the `del` statement:

```
>>> a = [1, 2, 3, 4]
>>> del a
```

This will delete the variable name `a`. In general, however, it does not delete the object to which a points *unless* `a` is the only variable pointing to it and Python's garbage-collecting routines kick in. Consider:

```
>>> a = [1, 2, 3, 4]
>>> b = a
>>> del a
>>> b
[1, 2, 3, 4]
```

Multiple assignment

Lists and tuples enable multiple items to be assigned at the same time. Consider the following example using lists:

```
>>> [a, b, c] = [1, 5, 9]
>>> a
1
>>> b
5
>>> c
9
```

In this example, Python assigned variables by lining up elements in the lists on each side. The lists must be the same length, or an error will be returned.

Tuples are more efficient for this purpose and are usually used instead of lists for multiple assignments:

```
>>> (a, b, c) = (5, "hello", [1, 2])
>>> a
5
>>> b
'hello'
>>> c
[1, 2]
```

However, since Python will interpret any non-enclosed list of values separated by commas as a tuple it is more common to see the following, equivalent statement:

```
>>> a, b, c = 5, "hello", [1, 2]
```

Here, each side of the equals sign is interpreted as a tuple and the assignment proceeds as before. This notation is particularly helpful for functions that return multiple values. We will discuss this in greater detail later, but here is preview example of a function returning two values:

```
>>> a, b = f(c)
```

Technically, the function returns one thing – a tuple containing two values. However, the multiple assignment notation allows us to treat it as two sequential values. Alternatively, one could write this statement as:

```
>>> returned = f(c)
>>> a, b = returned
```

In this case, `returned` would be a tuple containing two values.

Because of multiple assignment, list comprehensions can also iterate over multiple values:

```
>>> l = [(1,2), (3,4), (5,6)]
>>> [a+b for (a,b) in l]
[3, 7, 11]
```

In this example, the tuple (a,b) is assigned to each item in `l`, in sequence. Since `l` contains tuples, this amounts to assigning a and b to individual tuple members. We could have done this equivalently in the following, less elegant way:

```
>>> [t[0] + t[1] for t in l]
```

Here, t is assigned to the tuple and we access its elements using bracket indexing. A final alternative would have been:

```
>>> [sum(t) for t in l]
```

A common use of multiple assignment is to swap variable values:

```
>>> a = 1
>>> b = 5
>>> a, b = b, a
>>> a
5
>>> b
1
```

String functions and manipulation

Python's string processing functions make it enormously powerful and easy to use for processing string and text data, particularly when combined with the utility of lists. Every string in Python (like every other variable) is an *object.* String functions are member functions of these objects, accessed using dot notation.

Keep in mind two very important points with these functions: (1) strings are immutable, so functions that modify strings actually return new strings that are modified versions of the originals; and (2) all string functions are *case sensitive* so that 'this' is recognized as a different string than 'This'.

Strings can be sliced just like lists. This makes it easy to extract substrings:

```
>>> s = "This is a string"
>>> s[:4]
'This'
>>> "This is a string"[-5:]
'string'
```

Strings can also be split apart into lists. The `split` function will automatically split strings wherever it finds whitespace (e.g., a space or a line break):

```
>>> "This is a string.\nHello.".split()
['This', 'is', 'a', 'string.', 'Hello.']
```

Alternatively, one can split a string wherever a particular substring is encountered:

```
>>> "This is a string.".split('is')
['Th', ' ', ' a string.']
```

The opposite of the split function is the `join` function, which takes a list of strings and joins them together with a common separation string. This function is actually called as a member function of the separation string, not of the list to be joined:

```
>>> l = ['This', 'is', 'a', 'string.', 'Hello.']
>>> " ".join(l)
'This is a string. Hello.'
>>> ", ".join(["blue", "red", "orange"])
'blue, red, orange'
```

The join function can be used with a zero-length string:

```
>>> "".join(["house", "boat"])
'houseboat'
```

To remove extra beginning and ending whitespace, use the strip function:

```
>>> "    string   ".strip()
```

```
'string'
>>> "string\n\n  ".strip()
'string'
```

The `replace` function will make a new string in which all specified substrings have been replaced:

```
>>> 'We code in Python.  We like it.".replace("We", "You")
'You code in Python.  You like it.'
```

It is possible to test if a substring is present in a string and to get the index of the first character in the string where the substring starts:

```
>>> s = "This is a string."
>>> "is" in s
True
>>> s.index("is")
2
>> s.index("not")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
```

Sometimes you need to left- or right-justify strings within a certain field width, padding them with extra spaces as necessary. There are two functions for doing that:

```
>>> s = "apple".ljust(10) + "orange".rjust(10) + "\n"  \
...     + "grape".ljust(10) + "pear".rjust(10)
>>> print s
apple         orange
grape           pear
```

There are a number of functions for manipulating capitalization:

```
>>> s = "this is a String."
>>> s.lower()
'this is a string.'
>>> s.upper()
'THIS IS A STRING.'
>>> s.capitalize()
'This is a string.'
>>> s.title()
'This Is A String.'
```

Finally, there are a number of very helpful utilities for testing strings. One can determine if a string starts or ends with specified substrings:

```
>>> s = "this is a string."
>>> s.startswith("th")
True
>>> s.startswith("T")
False
```

```
>>> s.endswith(".")
True
```

You can also test the kind of contents in a string. To see if it contains all alphabetical characters,

```
>>> "string".isalpha()
True
>>> "string.".isalpha()
False
```

Similarly, you can test for all numerical characters:

```
>>> "12834".isdigit()
True
>>> "50 cars".isdigit()
False
```

Dictionaries

Dictionaries are another type in Python that, like lists, are collections of objects. Unlike lists, dictionaries have no ordering. Instead, they associate *keys* with *values* similar to that of a database. To create a dictionary, we use braces. The following example creates a dictionary with three items:

```
>>> d = {"city":"Santa Barbara", "state":"CA", "zip":"93106"}
```

Here, each element of a dictionary consists of two parts that are entered in *key:value* syntax. The keys are like labels that will return the associated value. Values can be obtained by using bracket notation:

```
>>> d["city"]
'Santa Barbara'
>>> d["zip"]
'93106'
>>> d["street"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'street'
```

Notice that a nonexistent key will return an error.

Dictionary keys do not have to be strings. They can be any *immutable* object in Python: integers, tuples, or strings. Dictionaries can contain a mixture of these. Values are not restricted at all; they can be any object in Python: numbers, lists, modules, functions, anything.

```
>>> d = {"one" : 80.0,  2 : [0, 1, 1],  3 : (-20,-30),  (4, 5) : 60}
>>> d[(4,5)]
60
```

```
>>> d[2]
[0, 1, 1]
```

The following example creates an empty dictionary:

```
>>> d = {}
```

Items can be added to dictionaries using assignment and a new key. If the key already exists, its value is replaced:

```
>>> d = {"city":"Santa Barbara", "state":"CA"}
>>> d["city"] = "Goleta"
>>> d["street"] = "Calle Real"
>>> d
{'city': 'Goleta', 'state': 'CA', 'street': 'Calle Real'}
```

To delete an element from a dictionary, use the `del` statement:

```
>>> del d["street"]
```

There are two ways to test if a key is in a dictionary:

```
>>> d = {"city":"Santa Barbara", "state":"CA"}
>>> "city" in d
True
>>> d.has_key("zip")
False
```

The size of a dictionary is given by the `len` function:

```
 >>> len(d)
2
```

To remove all elements from a dictionary, use the `clear` object function:

```
>>> d = {"city":"Santa Barbara", "state":"CA"}
>>> d.clear()
>>> d
{}
```

One can obtain lists of all keys and values (in no particular order):

```
>>> d = {"city":"Santa Barbara", "state":"CA"}
>>> d.keys()
['city', 'state']
>>> d.values()
['Santa Barbara', 'CA']
```

Alternatively, one can get a list of (key,value) tuples for the entire dictionary:

```
>>> d.items()
[('city', 'Santa Barbara'), ('state', 'CA')]
```

Similarly, it is possible to create a dictionary from a list of two-tuples:

```
>>> l = [("street", "Calle Real"), ("school", "UCSB")]
>>> dict(l)
{'school': 'UCSB', 'street': 'Calle Real'}
```

Finally, dictionaries provide a method to return a default value if a given key is not present:

```
>>> d = {"city":"Santa Barbara", "state":"CA"}
>>> d.get("city", "Goleta")
'Santa Barbara'
>>> d.get("zip", 93106)
93106
```

If statements

`if` statements allow conditional execution. Here is an example:

```
>>> x = 2
>>> if x > 3:
...   print "greater than three"
... elif x > 0:
...   print "greater than zero"
... else:
...   print "less than or equal to zero"
... <hit return>
greater than zero
```

Notice that the first testing line begins with `if`, the second `elif` meaning 'else if', and the third with `else`. Each of these is followed by a colon with the corresponding commands to execute. Items after the colon are indented. For `if` statements, both `elif` and `else` are optional.

A very important concept in Python is that *spacing and indentations carry syntactical meaning*. That is, they dictate how to execute statements. Colons occur whenever there is a set of sub-commands after an `if` statement, loop, or function definition. All of the commands that are meant to be grouped together after the colon *must be indented by the same amount*. Python does not specify how much to indent, but only requires that the commands be indented in the same way. Consider:

```
>>> if 1 < 3:
...     print "line one"
...       print "line two"
  File "<stdin>", line 3
      print "line two"
      ^
IndentationError: unexpected indent
```

An error is returned from unexpected indentation. In contrast, the following works:

```
>>> if 1 < 3:
...       print "line one"
...       print "line two"
... <hit return>
line one
line two
```

It is typical to indent *four* spaces after each colon. Ultimately Python's use of syntactical whitespace helps make its programs look cleaner and more standardized.

Any statement or function returning a Boolean `True` or `False` value can be used in an `if` statement. The number 0 is also interpreted as `False`, while any other number is considered `True`. Empty lists and objects return `False`, whereas non-empty ones are `True`.

```
>>> d = {}
>>> if d:
...     print "Dictionary is not empty."
... else:
...    print "Dictionary is empty."
... <hit return>
Dictionary is empty.
```

Single `if` statements (without `elif` or `else` constructs) that execute a single command can be written in one line without indentation:

```
>>> if 5 < 10: print "Five is less than ten."
Five is less than ten.
```

Finally, `if` statements can be nested using indentation:

```
>>> s = "chocolate chip"
>>> if "mint" in s:
...     print "We do not sell mint."
... elif "chocolate" in s:
...     if "ripple" in s:
...         print "We are all out of chocolate ripple."
...     elif "chip" in s:
...         print "Chocolate chip is our most popular."
... <hit return>
Chocolate chip is our most popular.
```

For loops

Like other programming languages, Python provides a mechanism for looping over consecutive values. *Unlike* many languages, however, Python's loops do not intrinsically iterate over integers, but rather *elements* in *sequences,* like lists and tuples. The general construct is:

```
>>> for element in sequence:
...   <commands>
```

Notice that anything falling within the loop is indented beneath the first line, similar to `if` statements. Here are some examples that iterate over tuples and lists:

```
>>> for i in [3, "hello", 9.5]:
...   print i
... <hit return>
3
hello
9.5
>>> for i in (2.3, [8, 9, 10], {"city":"Santa Barbara"}):
...   print i
... <hit return>
2.3
[8, 9, 10]
{'city':'Santa Barbara'}
```

Notice that the items in the iterable do not need to be the same type. In each case, the variable i is given the value of the current list or tuple element, and the loop proceeds over these in sequence. One does not have to use the variable i; any variable name will do, but if an existing variable is used, its value will be overwritten by the loop.

It is very easy to loop over a part of a list using slicing:

```
>>> l = [4, 6, 7, 8, 10]
>>> for i in l[2:]:
...   print i
... <hit return>
7
8
10
```

Iteration over a dictionary proceeds over its keys, not its values. Keep in mind, though, that dictionaries will not return these in any particular order. In general, it is better to iterate explicitly over keys or values using the dictionary functions that return lists of these:

```
>>> d = {"city":"Santa Barbara", "state":"CA"}
>>> for val in d:
...   print val
... <hit return>
city
state
>>> for val in d.keys():
...   print val
... <hit return>
city
state
>>> for val in d.values():
...   print val
... <hit return>
Santa Barbara
CA
```

Using Python's multiple assignment capabilities, it is possible to iterate over more than one value at a time:

```
>>> l = [(1, 2), (3, 4), (5, 6)]
>>> for (a, b) in l:
...   print a + b
... <hit return>
3
7
11
```

In this example, Python cycles through the list and makes the assignment `(a,b) = element` for each element in the list. Since the list contains two-tuples, it effectively assigns `a` to the first member of the tuple and `b` to the second.

Multiple assignment makes it easy to cycle over both keys and values in dictionaries at the same time:

```
>>> d = {"city":"Santa Barbara", "state":"CA"}
>>> d.items()
[('city', 'Santa Barbara'), ('state', 'CA')]
>>> for (key, val) in d.items():
...   print "The key is %s and the value is %s" % (key, val)
... <hit return>
The key is city and the value is Santa Barbara
The key is state and the value is CA
```

It is possible to iterate over sequences of numbers using the range function:

```
>>> for i in range(4):
...   print i
... <hit return>
0
1
2
3
```

In other programming languages, one might use the following idiom to iterate through items in a list:

```
>>> l = [8, 10, 12]
>>> for i in range(len(l)):
...   print l[i]
... <hit return>
8
10
12
```

In Python, however, the following is more natural and efficient, and thus always preferred:

```
>>> l = [8, 10, 12]
```

```
>>> for i in l:
...   print i
... <hit return>
8
10
12
```

Notice that the second line could have been written in a single line since there is a single command within the loop, although this is not usually preferred because the loop is less clear upon inspection:

```
>>> for i in l: print l
```

If one desires to have the index of the loop in addition to the iterated element, the `enumerate` command is helpful:

```
>>> l = [8, 10, 12]
>>> for (ind, val) in enumerate(l):
...   print "The %ith element in the list is %d" % (ind, val)
... <hit return>
The 0th element in the list is 8.
The 1th element in the list is 10.
The 2th element in the list is 12.
```

Notice that `enumerate` returns indices that always begin at 0, whether or not the loop actually iterates over a slice of a list:

```
>>> l = [4, 6, 7, 8, 10]
>>> for (ind, val) in enumerate(l[2:]):
...   print "The %ith element in the list is %d" % (ind, val)
... <hit return>
The 0th element in the list is 7.
The 1th element in the list is 8.
The 2th element in the list is 10.
```

It is also possible to iterate over two lists simultaneously using the `zip` function:

```
>>> l1 = [1, 2, 3]
>>> l2 = [0, 6, 8]
>>> for (a, b) in zip(l1, l2):
...    print a, b, a+b
... <hit return>
1 0 1
2 6 8
3 8 11
```

The `zip` function can be used outside of `for` loops. It simply takes two or more lists and groups them together, making tuples of corresponding list elements:

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

```
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

This behavior, combined with multiple assignment, is how `zip` allows simultaneous iteration over multiple lists at once.

Like `if` statements, loops can be nested:

```
>>> for i in range(3):
...   for j in range(0,i):
...     print (i, j)
... <hit return>
(1,0)
(2,0)
(2,1)
```

It is possible to skip forward to the next loop iteration immediately, without executing subsequent commands in the same indentation block, using the `continue` statement. The following produces the same output as the previous example using `continue`, but is ultimately less efficient because more loop cycles need to be traversed:

```
>>> for i in range(3):
...   for j in range(3):
...     if i <= j: continue
...     print (i, j)
... <hit return>
(1,0)
(2,0)
(2,1)
```

One can also terminate the innermost loop using the `break` statement. Again, the following produces the same result but is almost as efficient as the first example because the inner loop terminates as soon as the `break` statement is encountered:

```
>>> for i in range(3):
...   for j in range(3):
...     if i <= j: break
...     print (i, j)
... <hit return>
(1,0)
(2,0)
(2,1)
```

While loops

Unlike for loops, while loops do not iterate over a sequence of elements but rather continue so long as some test condition is met. Their syntax follows indentation rules similar to the cases we have seen before. The initial statement takes the form

```
>>> while condition:
```

The following example computes the first couple of values in the Fibonacci sequence:

```
>>> k1, k2 = 1, 1
>>> while k1 < 20:
...   k1, k2 = k2, k1 + k2
...   print k1
1
2
3
5
8
13
21
```

Sometimes it is desired to stop the while loop somewhere in the middle of the commands that follow it. For this purpose, the `break` statement can be used with an *infinite loop*. In the previous example, we might want to print all Fibonacci numbers less than or equal to 20:

```
>>> k1, k2 = 1, 1
>>> while True:
...   k1, k2 = k2, k1 + k2
...   if k1 > 20: break
...   print k1
1
2
3
5
8
13
```

Here the infinite while loop is created with the `while True` statement. Keep in mind that, if multiple loops are nested, the `break` statement will stop only the innermost loop

Functions

Functions are an important part of any program. Some programming languages make a distinction between "functions" that return values and "subroutines" that do not return anything but rather *do* something. In Python, there is only one kind, functions, but these can return single, multiple, or no values at all. In addition, like everything else, functions in Python are objects. That means that they can be included in lists, tuples, or dictionaries, or even sent to other functions. This makes Python extraordinarily flexible.

To make a function, use the def statement:

```
>>> def add(arg1, arg2):
...   x = arg1 + arg2
...   return x
```

Here, `def` signals the creation of a new function named `add`, which takes two arguments. All of the commands associated with this function are then indented underneath the `def` statement, similar to the syntactic indentation used in loops. The `return` statement tells Python to do two things: exit the function and, if a value is provided, use that as the return value.

Unlike other programming languages, functions do not need to specify the types of the arguments sent to them. Python evaluates these at runtime every time the function is called. Using the above example, we could apply our function to many different types:

```
>>> add(1, 2)
3
>>> add("house", "boat")
'houseboat'
>>> add([1, 2, 3], [4, 5, 6])
[1, 2, 3, 4, 5, 6]
>>> add(1, "house")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in add
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

In the last example, an error occurs because the addition operator is not defined for an integer with a string. This error is only thrown when we call the function with inappropriate arguments.

The `return` statement can be called from anywhere within a function:

```
>>> def power(x, y):
...   if x <= 0:
...     return 0.
...   else:
...     return x**y
```

If no `return` statement is present within a function, or if the `return` statement is used without a return value, Python automatically returns the special value `None`:

```
>>> def test(x):
...   print "%11.4e" % x
...   return
...  <hit return>
>>> ret = test(1)
1.0000e+000
>>> ret == None
True
>>> ret is None
True
```

`None` is a reserved, special object in Python, similar to `True` and `False`. It essentially means *nothing*, and will not appear using the `print` statement. However, as seen in the above example, one can test for the `None` value using conditional equality or the `is` statement.

If one wants a function that modifies its behavior depending on the type of the argument, it is possible to test for different types using the type function:

```
>>> def add(arg1, arg2):
...   #test to see if one is a string and the other is not
...   if type(arg1) is str and not type(arg2) is str:
...     arg1convert = type(arg2)(arg1)
...     return arg1convert + arg2
...   elif not type(arg1) is str and type(arg2) is str:
...     arg2convert = type(arg1)(arg2)
...     return arg1 + arg2convert
...   else:
...     return arg1 + arg2
... <hit return>
>>> add(1, "40.")
41
>>> add(40., "1")
41.0
```

Notice that in this example, the `type(arg2)` statement is also used to return the *function* that converts generic objects to the type of arg2, e.g., `int`, `float`, or `complex`. Thus the statement `type(arg2)(arg1)` actually runs this type-conversion function on the string `arg1` to convert it to the type of `arg2`.

Functions can return more than one value using Python's tuple capabilities. To do so, specify a comma-separated list after the return statement:

```
>>> def test(x, y):
...   a = x / y
...   b = x % y
...   return a, b
... <hit return>
>>> test(5, 2)
(2, 1)
>>> c, d = test(5,2)
>>> c
2
>>> d
1
```

Optional arguments in functions

Arguments of functions can be optional. Such optional arguments must have a default value, specified in the `def` statement. If optional arguments are given when a function is called, the arguments will take on the supplied values. If not, they will assume the default values:

```
>>> def fmtWithUnits(x, format = "%.3f", unit = "inches"):
...   return format % x + " " + unit
... <hit return>
>>> fmtWithUnits(7)
'7.000 inches'
>>> fmtWithUnits(7, "%.1f")
'7.0 inches'
>>> fmtWithUnits(7, "%.1f", "feet")
'7.0 feet'
>>> fmtWithUnits(7, unit = "feet")
'7.000 feet'
```

Notice in the penultimate line that we needed to specify the `unit` optional argument explicitly, since we skipped the optional `format` one. In general, it is good practice to explicitly specify optional arguments in this way whether or not one needs to, since this makes it clearer that the arguments in the call are optional:

```
>>> fmtWithUnits(7, format = "%.1f", unit = "feet")
'7.0 feet'
```

Function namespaces

Argument variables within functions exist in their own *namespace*. This means that assignment of an argument to a new value does not affect the original value outside of the function. Consider the following:

```
>>> def increment(a):
...   a = a + 1
...   return a
... <hit return>
>>> a = 5
>>> increment(a)
6
>>> a
5
```

What happened here? Because `a` is an argument variable defined in the `def` statement, it is treated as a new variable that exists only within the function. Once the function has finished and the program exits it, this new `a` is destroyed in memory by Python's garbage-collecting routines. The `a` that we defined outside of the function remains the same.

How, then, does one modify variables using functions? In other programming languages, you may have been used to sending variables to functions to change their values directly. This is not a Python way of doing things. Instead, the Pythonic approach is to use assignment to a function return value. This is actually a clearer approach than the way of many other programming languages because it shows explicitly that the variable is being changed upon calling the function:

```
>>> def increment(a):
...   return a + 1
... <hit return>
>>> a = 5
>>> a = increment(a)
>>> a
6
```

There is one subtlety to this issue. Mutable objects *can* actually be changed by functions if one uses object functions and/or element access. Consider the following example that uses both to modify a list:

```
>>> def modifylist(l):
...   l.append(5)
...   l[0] = 20
... <hit return>
>>> l = [1, 2, 3]
>>> modifylist(l)
>>> l
[20, 2, 3, 5]
```

The reason for the distinction with mutable objects has to do with Python's name-binding approach. Consider the following generic construct:

```
>>> def fn(arg):
...   arg = newvalue
>>> x = value
>>> fn(x)
```

When one calls `fn(x)`, Python creates the new variable `arg` within the function namespace and points it to the data residing in the spot of memory to which `x` points. Setting `arg` equal to another value within the function simply has the effect of pointing `arg` to a new location in memory corresponding to *`newvalue`*, rather than changing the existing spot in memory associated with `x`. Therefore, `x` remains unaffected.

On the other hand, consider the following:

```
>>> def fn(arg):
...   arg[index] = newvalue
>>> x = [values]
>>> fn(x)
```

Here, in the second line, the bracket notation tells Python to do the following: find the area in memory where the *`index`*th element of `arg` resides and put *`newvalue`* in it. This occurs because the brackets after `arg` are actually treated as an object function of `arg`, and thus are inherently a function of the memory and data to which `arg` points. A similar case would exist if we had called some object function that modified its contents, like `arg.sort()`. In these cases, `x` would be modified outside of the function.

Functions as objects

As alluded to previously, functions are objects and thus can be sent to other functions as arguments. Consider the following:

```
>>> def squareme(x):
...   return x*x
... <hit return>
>>> def applytolist(l, fn):
...   return [fn(ele) for ele in l]
... <hit return>
>>> l = [1, 7, 9]
>>> applytolist(l, squareme)
>>> [1, 49, 81]
```

Here, we sent the `squareme` function to the `applytolist` function. Notice that when we send a *function* to another function, we do not supply arguments. If we had supplied arguments, we would have instead sent the return value of the function, rather than the function itself.

Python shows us that a function is an object. Consider, from the above example:

```
>>> squareme
<function squareme at 0x019F60F0>
```

The hexadecimal number in the return value simply tells us where in memory this function lies. We can also test the type:

```
>>> type(squareme)
<type 'function'>
```

Like other objects, we can perform assignment using functions:

```
>>> def a(x, y):
...   return x+y
... <hit return>
>>> b = a
>>> b(1, 4)
5
```

Function documentation

Functions can be self-documenting in Python. A *docstring* can be written after the `def` statement that provides a description of what a function does. This extremely useful for documenting your code and providing explanations that both you and subsequent users can use. The built-in `help` function uses docstrings to provide help about functions.

```
>>> def a(x, y):
...   """Adds two variables x and y, of any type.  Returns single value."""
```

```
...   return x + y
... <hit return>
>>> help(a)
Help on function a in module __main__:

a(x, y)
    Adds two variables x and y, of any type.  Returns single value.
```

It is typical to enclose docstrings using triple-quotes, since complex functions might require longer, multi-line documentation. It is a good habit to write docstrings for your code. Each should contain three pieces of information: (1) a basic description of what the function does, (2) what the function expects as arguments, and (3) what the function returns (including the variable types).

Writing scripts

So far, the examples we have covered have involved commands interpreted directly from the Python interactive prompt. Python also supports scripts, or lists of commands and function definitions (and any other Python constructs) that are defined in files, similar to source code in other programming languages. These *scripts* are no different from the commands and instructions that you would enter at the command prompt. Python scripts end in the extension `.py` in all platforms.

Consider the following contents of a script file called `primes.py` that finds all primes less than or equal to 50:

primes.py

```
def nextprime(primelist):
    #find the maximum term in primelist and start one more than it
    k = max(primelist) + 1
    #starting at this number find the next integer that is
    #not divisible by any number in primelist
    while True:
        FoundDivisor = False
        #search current primes for a divisor; if found, k not a prime
        for prime in primelist:
            if k % prime == 0:
                FoundDivisor = True
                #break the loop since we don't need to test further
                break
        #check if we found any divisors
        if FoundDivisor:
            #try the next number
            k += 1
        else:
            #found a prime; return it
            return k

#set the max prime
```

```
upperlimit = 50

#find all primes less than or equal to this value
l = [2]
while l[-1] < upperlimit:
    l.append(nextprime(l))

#trim out the last element, which is greater than upperlimit
l = l[:-1]

#print out the list
print l
```

We can run this program from the command line by calling Python with an argument that is the name of our script. Python will run the contents of the file as if we typed them at the interactive prompt and then exit. Under Windows, for example, this might look something like:

```
c:\> python primes.py
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
c:\>
```

Modules

It is also possible to import scripts from within the Python interpreter. When files of Python commands are imported in this way they are termed *modules*. Modules are a major basis of programming efforts in Python as they allow you to organize reusable code that can be imported as necessary in specific programming applications. Considering the previous example:

```
>>> import primes
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
>>> type(primes)
<type 'module'>
>>> primes.nextprime
<function nextprime at 0x019BEFB0>
>>> primes.l
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

Notice several features of this example:

- Scripts are imported using the `import` command. Upon processing the `import` statement, Python immediately executes the contents of the file `primes.py` file.
- One does not use the `.py` extension in the `import` command; Python assumes the file ends in this and is accessible in the current directory (if unchanged, the same directory from which Python was started). If Python does not find the script to be imported in the current directory, it will search a specific path called PYTHONPATH, discussed later.

- When Python executes the imported script, it creates an object from it of type module.
- Any objects created when running the imported file are not deleted but are placed as members of the module object. In this way, we can access the functions and variables that were part of the module program by using dot notation, like `primes.l` and `primes.nextprime`.

By making script objects members of the module, Python gives us a powerful way to write reusable code, i.e., code with generic functions and variables that we can import into programs. Modules can also import other modules, so that we can have hierarchies of code with variable degrees of generality.

Module objects can be created and modified just like any other object in Python:

```
>>> primes.l = []
>>> primes.l
[]
>>> primes.k = 5   #create new object in primes module
>>> primes.k
5
```

Importing a module twice does *not* execute it twice:

```
>>> import primes
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
>>> import primes
>>>
```

Python will import a module only once, for reasons of efficiency (in the case, for instance, that many modules import the same sub-module). This can be overridden using the `reload` function:

```
>>> import primes
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
>>> import primes
>>> reload(primes)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
<module 'primes' from 'primes.py'>
```

Sometimes we want scripts to behave differently when we execute them at the command line versus import them into other programs. Commonly we want the script to execute certain commands when run from the command line, but need to suppress this behavior when imported. To achieve this, we need to test whether or not the program is being run from the command line. Consider the following program:

```
test.py
#defined regardless of how we run / import this script
```

```
def multiply(x, y):
    return x*y

if __name__ == "__main__":
    #only executed if run directly from the command line
    print multiply(4, 5)
```

In the penultimate line, we test to see if the script `test.py` has been run from the command line. The variable `__name__` is a special variable that Python creates which tells us the name of the current module. (There are many such special variables, and they are always identified by preceding and trailing double-underscores.) Python gives the value of "`__main__`" to the variable `__name__` if and only if that program is the main program and has been called from the command line (i.e., not imported). Here is the behavior of our program at the command line:

```
c:\> python test.py
20
c:\>
```

And here is its behavior if we import it:

```
>>> import test
>>> test.multiply(2, 3)
6
```

Notice that Python does not execute the `multiply(4, 5)` command when we import, but we still have access to any functions or objects defined in `test.py`.

It is not possible to use path names in the `import` statement. Instead by default, Python will look for modules in three places: (1) the current working directory, (2) a special directory called PYTHONPATH, and (3) the standard Python installation. The second location makes it convenient to store user-written reuseable code in a common folder. PYTHONPATH is actually a system environment variable that Python looks for and can point to such a folder. To set it on Windows machines, one needs to right-click on My Computer > Properties > Advanced > Environment variables. Then, PYTHONPATH can be added to the User Variables category with a value that is the name of a path where your common scripts are.

Standard modules

Python has a "batteries included" philosophy and therefore comes with a huge library of pre-written modules that accomplish a tremendous range of possible tasks. It is beyond the scope of this tutorial to cover all but a small few of these. However, here is a brief list of some of these modules that can come in handy for scientific programming:

- `os` – functions for various operating system operations

- `os.path` – functions for manipulating directory/folder path names
- `sys` – functions for system-specific programs and services
- `time` – functions for program timing and returning the current time/date in various formats
- `filecmp` – functions for comparing files and directories
- `tempfile` – functions for automatic creation and deletion of temporary files
- `glob` – functions for matching wildcard-type file expressions (e.g., "*.txt")
- `shutil` – functions for high-level file operations (e.g., copying, moving files)
- `struct` – functions for storing numeric data as compact, binary strings
- `gzip, bz2, zipfile, tarfile` – functions for writing to and reading from various compressed file formats
- `pickle` – functions for converting any Python object to a string that can be written to or subsequently read from a file
- `hashlib` – functions for cryptography / encrypting strings
- `socket` – functions for low-level networking
- `popen2` – functions for running other programs and capturing their output
- `urllib` – functions for grabbing data from internet servers
- `ftplib, telnetlib` – functions for interfacing with other computers through ftp and telnet protocols
- `audioop, imageop` – functions for manipulating raw audio and image data (e.g., cropping, resizing, etc.)

A complete listing of all of the modules that come with Python are given in the Python Library Reference in the Python Documentation. In addition to these modules, scientific computing can make extensive use of two add-on modules, `numpy` and `scipy`, that are discussed in a separate tutorial. There are also many other add-on modules that can be downloaded from open-source efforts and installed into the Python base.

Reading from files

An important component of any scientific program is the ability to read and write data to files. Python's built-in file utilities make this process very easy. Files are accessed by creating file *objects*. The member functions of these objects provide methods for reading from and/or writing to a file. To create a file object, use the `file` function. Consider a generic file `data.txt` whose contents are:

data.txt

```
#pressure temperature average_energy
```

```
1.0 1.0 -50.0
1.0 1.5 -27.8
2.0 1.0 -14.5
2.0 1.5 -11.2
```

The following example creates an open file object to `data.txt` and reads all of its contents into a string:

```
>>> f = file("data.txt", "r")
>>> s = f.read()
>>> f.close()
>>> s
'#pressure  temperature  average_energy\n1.0  1.0  -50.0\n1.0  1.5  -27.8\n2.0
1.0 -14.5\n2.0 1.5 -11.2'
>>> print s
#pressure temperature average_energy
1.0 1.0 -50.0
1.0 1.5 -27.8
2.0 1.0 -14.5
2.0 1.5 -11.2
```

Here, the `file` function took two arguments, the name of the file followed by a string "r" indicating that we are opening the file for reading. It is possible to omit the second argument, in which case Python defaults to "r"; however, it is usually a good idea to include it explicitly for programming clarity. One can also use the command `open`, which is a synonym for `file`.

Once the file object is created, we can read its entire contents into a string using the `read()` member function. Python reads all characters from the file, including whitespace and line breaks (e.g., "\n" entries).

When we have read the contents, we invoke the `close` function, which terminates the operating system link to our file. It is good to close files after using them, as open file objects consume system resources. In any case, Python automatically closes any open files if there are no more objects pointing to them using its garbage collecting routines. Consider the following:

```
>>> s = file("data.txt", "r").read()
```

This command accomplishes the same result as the last example, but does not create a file object that persists after execution. That is, `file` first creates the object, then `read()` extracts its contents and places them into the variable `s`. After this operation, there are no variables pointing to the file object anymore, and so it is automatically closed. File objects created within functions (that are not returned) are also always closed upon exiting the function, since variables created within functions are deleted upon exit.

One does not have to read all of the contents into a string. We can also read all of the contents into a list of lines in the file:

```
>>> l = file("data.txt", "r").readlines()
>>> l
['#pressure temperature average_energy', '1.0 1.0 -50.0', '1.0 1.5 -27.8',
2.0 1.0 -14.5', '2.0 1.5 -11.2']
```

Operationally, the `readlines()` function is identical to `read().split('\n')`.

If the file is large, it might not be efficient to read all of its contents at one time. Instead, we can read one line at a time using the `readline` function:

```
>>> f = file("data.txt", "r")
>>> s = "dummy"
>>> while len(s):
...   s = f.readline()
...   if not s.startswith("#"): print s.strip()
... <hit return>
1.0 1.0 -50.0
1.0 1.5 -27.8
2.0 1.0 -14.5
2.0 1.5 -11.2
>>> f.close()
```

This example prints out all of the lines that do not start with "#". The while loop continues as long as the last `readline()` command returns a string of length greater than zero. When Python reaches the end of a file, `readline` will return an empty string. It is important to know that `readline()` returns an entire line *including* the line break character '\n' at the end; in this way, a blank line will return a string of nonzero length. It is also for that reason that we used the `strip()` function when printing out the lines in the example above.

The `read` and `readline` functions can also take an optional argument `size` that sets the maximum number of characters (bytes) that Python will read in at a time. Subsequent calls move through the file until the end of the file is reached, at which point Python will return an empty string:

```
>>> f = file("data.txt", "r")
>>> f.read(5)
'#pres'
>>> f.read(5)
'sure '
>>> f.close()
```

The `seek` function can be used to move to a specific byte location in a file. Similarly, the `tell` function will indicate the current byte position within the file:

```
>>> f = file("data.txt", "r")
>>> f.seek(5)
>>> f.read(5)
'sure '
>>> f.tell()
```

```
10L
>>> f.close()
```

The 'L' after the number 10 simply indicates that the returned type is a long integer, since normal integers do not contain enough precision to address all of the bytes in large files.

We end with an example that illustrates some of the elegant ways in which Python can handle files. Imagine we would like to parse the data in the file above into the list called `Data` such that:

```
Data = [[1.0, 1.0, -50.0,], [1.0, 1.5, -27.8], [2.0, 1.0, -14.5], [2.0,
1.5, -11.2]]
```

Here, we need to read the data (ignoring the comment), convert it to floats, and structure it into a list. New Python programmers might take an approach similar to the manner in which this would be accomplished in other languages:

```
>>> f = file("data.txt", "r")
>>> Data = []
>>> line = f.readline()
>>> while len(line):
...   if not line.startswith("#"):
...     l = line.split()
...     Pres = float(l[0])
...     Temp = float(l[1])
...     Ene = float(l[2])
...     Data.append([Pres, Temp, Ene])
...   line = f.readline()
... <hit return>
>>> f.close()
```

We could shorten the program by using the `readlines` function and by moving the file object creation into the loop itself:

```
>>> Data = []
>>> for line in file("data.txt", "r").readlines():
...   if not line.startswith("#"):
...     l = line.split()
...     Pres = float(line[0])
...     Temp = float(line[1])
...     Ene = float(line[2])
...     Data.append([Pres, Temp, Ene])
... <hit return>
```

Ultimately, however, we can make these operations much more compact using Python's list comprehensions:

```
>>> Data = [[float(x) for x in line.split()]
...         for line in file("data.txt", "r").readlines()
...         if not line.startswith("#")]
... <hit return>
```

Here, we use two nested list comprehensions: the inner one loops over columns in each line, and the outer one over lines in the file with a filter established by the `if` statement.

Writing to files

Writing data to a file is very simple. To begin writing to a new file, open a file object with the "w" flag:

```
>>> f = file("new.txt", "w")
>>> f.write("This is the first line.")
>>> f.write("  Still on the first line.")
>>> f.write("\nThis is the second line.")
>>> f.close()
```

Here, the contents of our file `new.txt` would look like:

```
This is the first line.  Still on the first line.
This is the second line.
```

The `write` flag "w" tells Python to create a new file ready for writing, and the function `write` will write a string verbatim to the current position within the file. Subsequent `write` statements therefore append data to the file. Notice that `write` writes the string text *explicitly* and so line breaks must be specified in the strings if desired in the file.

If the "w" flag is used on a file that already exists, Python will overwrite it completely. Alternatively, one can *append* data to an existing file using the "a" flag:

```
>>> f = file("new.txt", "a")
>>> f.write("\nThis is the third line.")
>>> f.close()
```

Our file would now look like:

```
This is the first line.  Still on the first line.
This is the second line.
This is the third line.
```

The `write` function only accepts strings. That means that numeric values must be converted to strings prior to writing to the file. This can be accomplished using the `str` function, which formats values into a default precision, or using string formatting:

```
>>> f = file("new.txt", "w")
>>> pi = 3.14159
>>> f.write(str(pi))
>>> f.write('\n')
>>> f.write("%.2f" % pi)
>>> f.close()
```

Binary data and compressed files

When storing numeric data, it is inefficient to write them to files in textual format because it requires many more characters to express a textual version of a float at the same precision it would require to hold it in memory. There are two approaches to more efficient writing of numeric data that results in smaller file sizes.

The first approach is not to store values in a legible format but to write them to the file in a way similar to their representation in memory. To do so, we must convert a value to a binary representation in string format. The `struct` module can be used for this purpose. However, there are some subtleties to the different data types (`struct` uses C, rather than Python, types) that can make this approach a bit confusing.

The second approach is to write to, and subsequently also read from, compressed files. In this way, numeric data written in human-readable form can be compressed to take up much less space on disk. This approach is sometimes more convenient because numeric values can still be read by human eyes when data files are decompressed by various utilities outside of Python.

Conveniently, Python comes with modules that enable one to read and write a number of popular compressed formats in an almost completely transparent manner. Two formats are recommended: the Gzip format, which achieves reasonable compression and is fast, and the Bzip2 format, which achieves higher compression but at the expense of speed. Both formats are standardized, open, can be read by most common decompression programs, and are single-file based, meaning they compress a single file, not *cabinets* or *archives* of multiple files, which complicates things.

To write to a new Gzip file, we import the `gzip` module and create a `GzipFile` object in a manner identical to the way we created a file object:

```
>>> import gzip
>>> f = gzip.GzipFile("data.txt.gz", "w")
>>> f.write("This is some test data for compression.")
>>> f.close()
>>> print gzip.GzipFile("data.txt.gz", "r").read()
This is some test data for compression.
```

Here, Python takes care of compression (and decompression) entirely behind the scenes. The only difference from our earlier efforts is that we have replaced the file function with the `gzip.GzipFile` call and we have given the extension ".gz" to the file we create, in order to indicate that it is a compressed file. In fact, `gzip` objects behave exactly like file objects, and implement all of the same functions (`read`, `readline`, `readlines`, `write`). This makes it very easy and transparent for storing data in a compressed format. One minor

exception, however, is that the `seek` and `tell` functions do not work exactly the same and should be avoided with compressed files.

The `bz2` module works in exactly the same manner:

```
>>> import bz2
>>> f = bz2.BZ2File("data.txt.bz2", "w")
>>> f.write("This is some test data for compression.")
>>> f.close()
>>> print bz2.BZ2File("data.txt.bz2", "r").read()
This is some test data for compression.
```

In general, compression is only recommended for datasets on disk that are large (e.g., > 1MB) and that are read or written only a few times during a program. For disk-intensive programs that are speed-limited by the rate at which they can read and write to disk, compression will incur a considerable computational overhead and it is probably best to work with an uncompressed file. In these latter cases, the large datasets can ultimately be compressed by outside utilities after all programs and analyses have been performed.

File system functions

Python offers a host of other modules and functions for accessing and manipulating files and directories on disk. The latter are indicated by strings. Python recognizes directory hierarchies using the forward slash character, regardless of the particular operating system (Windows, Linux, or MacOS). On Windows machines, it is also possible to use the backwards slash character; however, in strings this must be escaped since '\' normally tells Python that a special code is being used. For example, both of the following point to the same file on a Windows machine:

```
>>> print "c:/temp/file.txt"
c:/temp/file.txt
>>> print "c:\\temp\\file.txt"
c:\temp\file.txt
```

The `os` module contains a large number of useful file functions. In particular, the sub-module `os.path` provides a number of functions for manipulating path and file names. For example, a filename with a path can be split into various parts:

```
>>> import os
>>> p = "c:/temp/file.txt"
>>> os.path.basename(p)
'file.txt'
>>> os.path.dirname(p)
'c:/temp'
>>> os.path.split(p)
('c:/temp', 'file.txt')
```

The opposite of the `split` function is the `join` function. It is a good idea to always use `join` when combining pathnames with other pathnames or files, since `join` takes care of any operating-system specific actions. `join` can take any number of arguments:

```
>>> os.path.join("c:\\temp", "file.txt")
'c:\\temp\\file.txt'
>>> os.path.join("c:\", "temp", "file.txt")
'c:\\temp\\file.txt'
```

If the path name is not absolute but relative to the current directory, there is a function for returning the absolute version:

```
>>> os.path.abspath("/temp/file.txt")
'C:\\temp\\file.txt'
```

Several functions enable testing the existence and type of files and directories:

```
>>> p = 'c:/temp/file.txt'
>>> os.path.exists(p)
True
>>> os.path.isfile(p)
True
>>> os.path.isdir(p)
False
```

Here, the `isfile` and `isdir` functions test both for the existence of the object as well as their type.

One can get the size on disk (in bytes) of a file:

```
>>> os.path.getsize('c:/temp/file.txt')
39482
```

Several functions in the main `os` module allow interrogating and changing the current working directory:

```
>>> os.getcwd()
'C:\\temp'
>>> os.chdir("..")
>>> os.getcwd()
'C:\\'
```

Note that the notation `".."` signifies the containing directory one level up.

A directory can be created:

```
>>> os.mkdir("c:/temp/newdir")
```

To delete a file:

```
>>> os.remove("c:/temp/deleteme.txt")
```

To delete a directory:

```
>>> os.rmdir("c:/temp/newdir")
```

To rename a file:

```
>>> os.rename("c:/temp/file.txt", "c:/temp/newname.txt")
```

The `shutil` module provides methods for copying and moving files:

```
>>> import shutil
>>> shutil.copy("c:/temp/file.txt", "c:/temp/copied.txt")
>>> shutil.move("c:/temp/file.txt", "c:/moved.txt")
```

Finally, the `glob` module provides wildcard matching routines for finding files and directories that match a specification. Matches are placed in lists:

```
>>> import glob
>>> glob.glob("c:\\temp\\*.dat")
['c:\\temp\\1.dat', 'c:\\temp\\2.dat', 'c:\\temp\\3.dat']
```

Here the "*" wildcard matches anything of any length. The "?" wildcard will match anything of length one character. Multiple wildcards can appear in a `glob` specification:

```
>>> glob.glob("c:\\*\\?.dat")
['c:\\temp\\1.dat', 'c:\\temp\\2.dat', 'c:\\temp\\3.dat', 'c:\\dat\\0.dat']
```

`glob` returns both files and directories. List comprehensions provide an easy way to filter for one or the other.

```
>>> [p for p in glob.glob("p*") if os.path.isdir(p)]
['papers', 'presentations', 'proposals']
```

Command line arguments

It is very common to write programs that run with options from the command line, i.e., the DOS command prompt in Windows or a terminal in Linux or MacOS. Usually, one provides a number of arguments to the program that are detected. Let's say we wanted a program to take an input file `in.txt` and produce an output file `out.txt` in the following way at the prompt:

```
C:\> python program.py in.txt out.txt
```

In Windows, if Python is associated with files ending in '.py', we can just write instead:

```
C:\> program.py in.txt out.txt
```

In Linux, we can accomplish the same behavior by including in the very first line of our program a comment directive that tells the system to use Python to execute the file:

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

Either way, we would like to capture the arguments `in.txt` and `out.txt`. To do this, we use the `sys` module and its member variable `argv`:

program.py

```
#!/usr/bin/env python
import sys
print sys.argv
InputFile = sys.argv[1]
OutputFile = sys.argv[2]
```

Running program.py from the command line:

```
C:\> program.py in.txt out.txt
['program.py', 'in.txt', 'out.txt']
```

Notice that `argv` is a list that contains the (string) arguments in order. The first argument, with index 0, is the name of the program that we are executing. Subsequent arguments correspond to space-separated items that we input on the command line when running the program. The form of `argv` is exactly the same whether or not we call Python directly, since the Python executable is ignored:

```
C:\> program.py in.txt out.txt
['program.py', 'in.txt', 'out.txt']
C:\> python program.py in.txt out.txt
['program.py', 'in.txt', 'out.txt']
```

Classes

So far, we have only dealt with built-in object types like floats and ints. Python, however, allows us to create new object types called *classes*. We can then use these classes to create new objects of our own design. In the following example, we create a new class that describes an atom type.

atom.py

```
class AtomClass:
    def __init__(self, Velocity, Element = 'C', Mass = 12.0):
        self.Velocity = Velocity
        self.Element = Element
        self.Mass = Mass
    def Momentum(self):
        return self.Velocity * self.Mass
```

We can import the `atom.py` module and create a new instance of the `AtomClass` type:

```
>>> import atom
>>> a = atom.AtomClass(2.0, Element = 'O', Mass = 16.0)
>>> b = atom.AtomClass(1.0)
>>> a.Element
'O'
>>> a.Mass
12.0
>>> a.Momentum()
32.0
>>> b.Element
'C'
>>> b.Velocity
1.0
```

In this example, the `class` statement indicates the creation of a new class called `AtomClass`; all definitions for this class must be indented underneath it. The first definition is for a special function called `__init__` that is a *constructor* for the class, meaning this function is automatically executed by Python every time a new object of type `AtomClass` is created. There are actually many special functions that can be defined for a class; each of these begins and ends with two underscore marks.

Notice that the first argument to the `__init__` function is the object `self`. This is a generic feature of any class function. This syntax indicates that the object itself is automatically sent to the function upon calls to it. This allows modifications to the object by manipulating the variable `self`; for example, new object members are added using expressions of the form `self.X = Y`. This approach may seem unusual, but it actually simplifies the ways in which Python defines class functions behind the scenes.

The `__init__` function gives the form of the arguments that are used when we create a new object with `atom.AtomClass(2.0, Element = 'O', Mass = 16.0)`. Like any other function in Python, this function can include optional arguments.

Object members can be accessed using dot notation, as shown in the above example. Each new instance object of a class acquires its own object members, separate from other instances. Functions can also be defined as object members, as shown with the `Momentum` function above. The first argument to any function in this definition must always be `self`; calls to functions through object instances, however, do not supply this variable since Python sends the object itself automatically as the first argument.

Many special functions can be defined for objects that tell Python how to use your new type with existing operations. Below is a selected list of some of these:

special class method	behavior / purpose

`__del__(self)`	A destructor; called when an instance is deleted using `del` or via Python's garbage collecting routines.
`__repr__(self)`	Returns a string representation of the object; used by `print` statements, for example
`__cmp__(self, other)`	Defines a comparison method with other objects. Returns a negative number if self < other, zero if self == other, and a positive number if self > other. Used to evaluate comparison statements for objects, like `a > b`, or for sorting.
`__len__(self)`	Returns the length of the object; used by the `len` function.

`__getitem__(self, key)`
`__setitem__(self, key, value)`
`__delitem__(self, key)`
	Define methods for accessing and modifying elements of an object via bracket notation, e.g., `a[key] = value`.

`__contains__(self, item)`	Called for an object when the `in` statement is used, e.g., `item in a`.

`__add__(self, other)`
`__sub__(self, other)`
`__mul__(self, other)`
`__div__(self, other)`
`__mod__(self, other)`
`__pow__(self, other)`
	Methods that are called when various arithmetic operations are executed on objects, e.g., a + b, a – b, a * b, a / b, a % b, and a**b. In other programming languages, these functions might be termed *operator overloading*.

Classes can be an extremely convenient way for organizing data in scientific programs. However, this benefit does not come without a cost: oftentimes stratifying data across a class will slow your program considerably. Consider the atom class defined above. We could put a separate position or velocity vector inside each atom instance. However, when we perform calculations that make intense use of these quantities—such as a pairwise loop that computes all interatomic distances—it is inefficient for Python to jump around in memory accessing individual position variables in each class. Rather, it would be much more efficient to store all positions for all atoms in a single large array that occupies one location in memory. In this case, we would consider those quantities that appear in the slowest step of our calculations (typically the pairwise loop) and keep them *outside* of the classes as large, easily manipulated arrays and then put everything else that is not accessed frequently (such as the element name) *inside* the class definitions. Such a separation may seem messy, but ultimately it is essential if we are to achieve reasonable performance in numeric computations.

Exceptions

Python offers a simple way to test for errors as a part of a program using the `try` and `except` statements:

test.py

```
def multiply(x, y):
    try:
        ret = x * y
    except StandardError:
        ret = 0
    return ret
```

Here we have defined a function that performs multiplication that we can call for any type. If multiplication is not defined for a particular type, an error is thrown that is caught by the `except` statement. Rather than stop our program, this error causes our own error-handling code to be executed. The `try` statement defines the range of code in which we are testing for this error. Consider this example:

```
>>> import test
>>> test.multiply(3, 6)
18
>>> test.multiply("3", "6")
0
>>> "3" * "6"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'str'
```

In the example above, we caught the kind of error called `StandardError`, which is a broad category that includes the specific kind of error `TypeError`. Python has a large hierarchy of errors that can be caught. Taken from the Python manual:

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- Exception
      +-- GeneratorExit
      +-- StopIteration
      +-- StandardError
      |    +-- ArithmeticError
      |    |    +-- FloatingPointError
      |    |    +-- OverflowError
      |    |    +-- ZeroDivisionError
      |    +-- AssertionError
      |    +-- AttributeError
      |    +-- EnvironmentError
      |    |    +-- IOError
      |    |    +-- OSError
      |    |         +-- WindowsError (Windows)
      |    |         +-- VMSError (VMS)
      |    +-- EOFError
      |    +-- ImportError
      |    +-- LookupError
```

```
|     |     +-- IndexError
|     |     +-- KeyError
|     +-- MemoryError
|     +-- NameError
|     |     +-- UnboundLocalError
|     +-- ReferenceError
|     +-- RuntimeError
|     |     +-- NotImplementedError
|     +-- SyntaxError
|     |     +-- IndentationError
|     |           +-- TabError
|     +-- SystemError
|     +-- TypeError
|     +-- ValueError
|     |     +-- UnicodeError
|     |           +-- UnicodeDecodeError
|     |           +-- UnicodeEncodeError
|     |           +-- UnicodeTranslateError
+-- Warning
      +-- DeprecationWarning
      +-- PendingDeprecationWarning
      +-- RuntimeWarning
      +-- SyntaxWarning
      +-- UserWarning
      +-- FutureWarning
   +-- ImportWarning
   +-- UnicodeWarning
```

In addition to catching errors, we can also *throw* errors using the `raise` statement:

```
>>> raise FloatingPointError, "A floating point error has occurred."
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FloatingPointError: A floating point error has occurred.
```

The ability to raise errors is convenient for adding user-defined information when improper calls to our functions or objects are made. Ultimately this helps us locate bugs in our code.

Timing functions and programs

It is often useful to be able to time routines in our program, to get a sense of the relative computation demands of different parts of it. A very simple approach is to use the `time` module:

```
>>> import time
>>> time.time()
1236970442.9519999
```

The `time()` function of the `time` module gives the time in seconds as measured from a reference date called the *epoch*. Ultimately, we are interested in time differences between two points in our program and so this exact date is unimportant. Consider the following code snippet from a script that computes the time required for a particular function `ComputeEnergies()` to finish:

```
t1 = time.time()
ComputeEnergies()
t2 = time.time()
print "The time required was %.2f sec" % (t2 - t1)
```

For long programs, adding such statements for each function execution would be very tedious. Python includes a *profiling* module that enables you to examine timings throughout your code. There are two modules: `profile` and `cProfile`. These modules are entirely identical except that `cProfile` has been written mostly in C and is much faster. `cProfile` is always recommended unless you have an older version of Python that doesn't include it.

To use `cProfile` to profile a single function,

```
import cProfile
cProfile.run("ComputeEnergies()")
```

Notice that we send to the run function in `cProfile` a string that we want to execute. After `ComputeEnergies()` finishes, `cProfile` will print out a long list of statistics about timings in for that function and the functions it calls.

To profile a complete script, we can run `cProfile` on it from the command line:

```
c:\> python –m cProfile myscript.py
```

After running, we get a report that looks something like this (abbreviated):

```
         15686 function calls (15618 primitive calls) in 10.570 CPU seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 __config__.py:3(<module>)
        1    0.000    0.000    0.000    0.000 __future__.py:48(<module>)
        1    0.000    0.000    0.000    0.000 __future__.py:70(_Feature)
        5    0.000    0.000    0.000    0.000 __future__.py:71(__init__)
        1    0.000    0.000    0.000    0.000 __init__.py:161(c_ushort)
        1    0.000    0.000    0.000    0.000 __init__.py:165(c_long)
        1    0.000    0.000    0.000    0.000 __init__.py:169(c_ulong)
        1    0.000    0.000    0.000    0.000 __init__.py:186(c_float)
        1    0.000    0.000    0.000    0.000 __init__.py:190(c_double)
        1    0.000    0.000    0.000    0.000 __init__.py:199(c_longlong)
        1    0.012    0.012    0.036    0.036 __init__.py:2(<module>)
```

```
     1    0.000    0.000    0.000    0.000 __init__.py:203(c_ulonglong)
     1    0.000    0.000    0.001    0.001 md.py:25(InitPositions)
     1    0.007    0.007   10.569   10.569 md.py:3(<module>)
    11    0.001    0.000    0.001    0.000 md.py:54(RescaleVelocities)
     1    0.000    0.000    0.000    0.000 md.py:71(InitVelocities)
     1    0.001    0.001    0.001    0.001 md.py:84(InitAccel)
     1   10.350   10.350   10.358   10.358 md.py:99(RunTest)
     1    0.000    0.000    0.000    0.000 memmap.py:1(<module>)
     1    0.000    0.000    0.000    0.000 memmap.py:17(memmap)
     1    0.003    0.003    0.005    0.005 numeric.py:1(<module>)
     3    0.000    0.000    0.000    0.000 numeric.py:142(extend_all)
     1    0.000    0.000    0.000    0.000 numeric.py:1685(seterr)
     1    0.000    0.000    0.000    0.000 numeric.py:1774(geterr)
    52    0.000    0.000    0.000    0.000 {min}
   100    0.001    0.000    0.001    0.000 {numpy.core.multiarray.array}
    18    0.000    0.000    0.000    0.000 {numpy.core.multiarray.empty}
     1    0.000    0.000    0.000    0.000 {numpy.core.multiarray.zeros}
     2    0.000    0.000    0.000    0.000 {numpy.core.umath.geterrobj}
     2    0.000    0.000    0.000    0.000 {numpy.core.umath.seterrobj}
     3    0.001    0.000    0.001    0.000 {open}
    42    0.000    0.000    0.000    0.000 {ord}
     8    0.000    0.000    0.000    0.000 {range}
     6    0.000    0.000    0.000    0.000 {setattr}
    15    0.000    0.000    0.000    0.000 {sys._getframe}
 10095    0.005    0.000    0.005    0.000 {time.time}
```

The names to the right are names of the modules and functions called by our program. Some of them might not look familiar; this is usually the case when modules that have functions that call other functions in the same and other modules. The numbers in columns give statistics about the program timing:

- `ncalls` – number of times a function was called
- `tottime` – total time spent in a function, summed over all calls
- `percall` – average time per call spent in a function
- `cumtime` – total time spent in a function and all the functions called by it
- `percall` – average time per call spent in a function and all the functions called by it