Chapter 2

Introduction to Programming

Some readers may disagree, but to me computers and mathematics are like beer and potato chips: two fine tastes that are best enjoyed together. Mathematics provides the foundations of our models and of the algorithms we use to solve them. Computers are the engines that run these algorithms. They are also invaluable for simulation and visualization. Simulation and visualization build intuition, and intuition completes the loop by feeding into better mathematics.

This chapter provides a brief introduction to scientific computing, with special emphasis on the programming language Python. Python is one of several outstanding languages that have come of age in recent years. It features excellent design, elegant syntax, and powerful numerical libraries. It is also free and open source, with a friendly and active community of developers and users.

2.1 Basic Techniques

This section provides a short introduction to the fundamentals of programming: algorithms, control flow, conditionals, and loops.

2.1.1 Algorithms

Many of the problems we study in this text reduce to a search for algorithms. The language we will use to describe algorithms is called *pseudocode*. Pseudocode is an informal way of presenting algorithms for the benefit of human readers, without getting tied down in the syntax of any particular programming language. It's a good habit to begin every program by sketching it first in pseudocode.

Our pseudocode rests on the following four constructs:

if–then–else, **while,** **repeat–until,** and **for**

The general syntax for the **if–then–else** construct is

```
if condition then
  | first sequence of actions
else
  | second sequence of actions
end
```

The condition is evaluated as either true or false. If found true, the first sequence of actions is executed. If false, the second is executed. Note that the **else** statement and alternative actions can be omitted: If the condition fails, then no actions are performed. A simple example of the **if–then–else** construct is

```
if there are cookies in the jar then
  | eat them
else
  | go to the shops and buy more
  | eat them
end
```

The **while** construct is used to create a loop with a test condition at the beginning:

```
while condition do
  | sequence of actions
end
```

The sequence of actions is performed only if the condition is true. Once they are completed, the condition is evaluated again. If it is still true, the actions in the loop are performed again. When the condition becomes false the loop terminates. Here's an example:

```
while there are cookies in the jar do
  | eat one
end
```

The algorithm terminates when there are no cookies left. If the jar is empty to begin with, the action is never attempted.

The **repeat–until** construct is similar:

```
**repeat**
  | sequence of actions
**until** *condition*
```

Here the sequence of actions is always performed once. Next the condition is checked. If it is true, the algorithm terminates. If not, the sequence is performed again, the condition is checked, and so on.

The **for** construct is sometimes called a definite loop because the number of repetitions is predetermined:

```
for element in sequence do
  | do something
end
```

For example, the following algorithm computes the maximum of a function f over a finite set S using a **for** loop and prints it to the screen.[1]

```
set $c = -\infty$
for $x$ in $S$ do
  | set $c = \max\{c, f(x)\}$
end
print $c$
```

In the **for** loop, x is set equal to the first element of S and the statement "set $c = \max\{c, f(x)\}$" is executed. Next x is set equal to the second element of S, the statement is executed again, and so on. The statement "set $c = \max\{c, f(x)\}$" should be understood to mean that $\max\{c, f(x)\}$ is first evaluated, and the resulting value is assigned to the variable c.

Exercise 2.1.1 Modify this algorithm so that it prints the maximizer rather than the maximum. Explain why it is more useful to know the maximizer.

Let's consider another example. Suppose that we have two arrays A and B stored in memory, and we wish to know whether the elements of A are a subset of the elements of B. Here's a prototype algorithm that will tell us whether this is the case:

```
set subset = True
for $a$ in $A$ do
  | if $a \notin B$ then set subset = False
end
print subset
```

[1]That is, it displays the value to the user. The term "print" dates from the days when sending output to the programmer required generating hardcopy on a printing device.

Exercise 2.1.2 The statement "$a \notin B$" may require coding at a lower level.[2] Rewrite the algorithm with an inner loop that steps through each b in B, testing whether $a = b$, and setting `subset` = False if no matches occur.

Finally, suppose we wish to model flipping a biased coin with probability p of heads, and have access to a random number generator that yields uniformly distributed variates on $[0, 1]$. The next algorithm uses these random variables to generate and print the outcome (either "heads" or "tails") of ten flips of the coin, as well as the total number of heads.[3]

```
set $H = 0$
for $i$ in 1 to 10 do
    draw $U$ from the uniform distribution on $[0, 1]$
    if $U < p$ then                                    // With probability $p$
        print "heads"
        $H = H + 1$
    else                                               // With probability $1 - p$
        print "tails"
    end
end
print $H$
```

Note the use of indentation, which helps maintaining readability of our code.

Exercise 2.1.3 Consider a game that pays \$1 if three consecutive heads occur in ten flips and zero otherwise. Modify the previous algorithm to generate a round of the game and print the payoff.

Exercise 2.1.4 Let b be a vector of zeros and ones. The vector corresponds to the employment history of one individual, where 1 means employed at the associated point in time, and 0 means unemployed. Write an algorithm to compute the longest (consecutive) period of employment.

2.1.2 Coding: First Steps

When it comes to programming, which languages are suitable for scientific work? Since the time it takes to complete a programming project is the sum of the time spent writing the code and the time that a machine spends running it, an ideal language would minimize both these terms. Unfortunately, designing such a language is not

[2] Actually, in many high-level languages you will have an operator that tests whether a variable is a member of a list. For the sake of the exercise, suppose this is not the case.

[3] What is the probability distribution of this total?

easy. There is an inherent trade-off between human time and computer time, due to the fact that humans and computers “think” differently: Languages that cater more to the human brain are usually less optimal for the computer and vice versa.

Using this trade-off, we can divide languages into (1) robust, lower level languages such as Fortran, C/C++, and Java, which execute quickly but can be a chore when it comes to coding up your program, and (2) the more nimble, higher level “scripting” languages, such as Python, Perl, Ruby, Octave, R, and MATLAB. By design, these languages are easy to write with and debug, but their execution can be orders of magnitude slower.

To give an example of these different paradigms, consider writing a program that prints out “Hello world.” In C, which is representative of the first class of languages, such a program might look like this:

```
#include <stdio.h>

int main(int argc, char *argv[]) {
      printf("Hello world\n");
      return 0;
}
```

Let’s save this text file as `hello.c`, and compile it at the shell prompt[4] using the gcc (GNU project C) compiler:

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

This creates an executable file called `hello` that can be run at the prompt by typing its name.

For comparison, let’s look at a “Hello world” program in Python—which is representative of the second class of languages. This is simply

```
print("Hello world")
```

and can be run from my shell prompt as follows:

```
python hello.py
```

What differences do we observe with these two programs? One obvious difference is that the C code contains more boilerplate. In general, C will be more verbose, requiring us to provide instructions to the computer that don’t seem directly relevant to our task. Even for experienced programmers, writing boilerplate code can be tedious and error prone. The Python program is much shorter, more direct, and more intuitive.

Second, executing the C program involves a two-step procedure: First compile, then run. The compiler turns our program into machine code specific to our operating system. By viewing the program as a whole prior to execution, the compiler can

[4]Don’t worry if you aren’t familiar with the notion of a shell or other the details of the program. We are painting with broad strokes at the moment.

optimize this machine code for speed. In contrast, the Python interpreter sends individual instructions to the CPU for processing as they are encountered. While slower, the second approach is more interactive, which is helpful for testing and debugging. We can run parts of the program separately, and then interact with the results via the interpreter in order to evaluate their performance.

In summary, the first class of languages (C, Fortran, etc.) put more burden on the programmer to specify exactly what they want to happen, working with the operating system on a more fundamental level. The second class of languages (Python, MATLAB, etc.) shield the programmer from such details and are more interactive, at the cost of slower execution. As computers have become cheaper and more powerful, these "scripting" languages have naturally become more popular. Why not ask the computer to do most of the heavy lifting?

In this text we will work exclusively with an interpreted language, leaving the first class of languages for those who wish to dig deeper. However, we note in passing that one can often obtain the best of both worlds in the speed versus ease-of-use trade-off. This is achieved by *mixed language programming*. The idea here is that in a typical program there are only a few lines of code that are executed many times, and it is these bottlenecks that should be optimized for speed. Thus the modern method of numerical programming is to write the entire program in an interpreted language such as Python, *profile* the code to find bottlenecks, and outsource those (and only those) parts to fast, compiled languages such as C or Fortran.[5]

The interpreted language we work with in the text is Python. However, MATLAB code is provided on the text homepage for those who prefer it. MATLAB has a gentler learning curve, and its numerical libraries are better documented. *Readers who are comfortable with MATLAB and have no interest in Python can skip the rest of this chapter.*

Python is a modern and highly regarded object-oriented programming language used in academia and the private sector for a wide variety of tasks. In addition to being powerful enough to write large-scale applications, Python is known for its minimalist style and clean syntax—designed from the start with the human reader in mind. Among the common general-purpose interpreted languages (Perl, Visual Basic, etc.), Python is perhaps the most suitable for scientific and numerical applications, with a large collection of MATLAB-style libraries organized primarily by the SciPy project (`scipy.org`).[6] The power of Python combined with these scientific libraries makes it an excellent choice for numerical programming.

Python is open source, which means that it can be downloaded free of charge,[7] and that we can peruse the source code of the various libraries to see how they work

[5]Another promising option is Cython, which is similar to Python but generates highly optimized C code.

[6]See also Sage, which is a mathematical tool kit built on top of Python.

[7]The main Python repositories are at `python.org`. Recently several distributions of Python that come bundled with various scientific tools have appeared. The book home page contains suggestions and links.

(and to make changes if necessary).

Rather than working directly with the Python interpreter, perhaps the best way to begin interacting with Python is by using IDLE.[8] IDLE is a free, cross-platform development environment for Python that comes bundled with most Python distributions. After you start IDLE, you will meet an interface to the interpreter that is friendlier than the standard one, providing color syntax highlighting, tab completion of commands, and more.[9] At the IDLE prompt you can start typing in commands and viewing the results:

```
>>> 10 * 10
100
>>> 10**2   # exponentiation
100
```

The result of the calculations is printed when you hit the return key. Notice that with the second calculation we added a *comment*, which is the hash symbol followed by text. Anything to the right of a hash is ignored by the interpreter. Comments are only for the benefit of human readers.

Continuing on with our tour of Python, next let's try assigning values to *variables*. Variables are names for values stored in memory. Here is an example:

```
>>> x = 3           # Bind x to integer 3
>>> y = 2.5         # Bind y to floating point number 2.5
>>> x               # Query the value of x
3
>>> z = x * y       # Bind z to x * y = 7.5
>>> x = x + 1       # Rebind x to integer 4
>>> a, b = 1, 2     # Bind a to 1 and b to 2
```

Names (`x`, `y`, etc.) are also called *identifiers*, and the values assigned to them are called *objects*. You can think of the identifiers as "pointers" to the location in memory where their values are stored. Identifiers are case sensitive (`X` is different from `x`), must start with a letter or an underscore, and cannot be one of Python's keywords.[10]

Observe that assignment of names to values (identifiers to objects) is accomplished via the "=" symbol. Assignment is also called *binding*: setting `x = 3` *binds* the identifier `x` to the integer object 3. Passed the statement `z = x * y`, the interpreter evaluates

[8]See `http://johnstachurski.net/book.html` for more information on how to get started with Python.

[9]IDLE is not the best development environment for Python, but it is the easiest to get started with. A more powerful alternative is the IPython shell combined with a good text editor (such as Emacs or Vim). The following discussion assumes that you are using IDLE—if you are using something else, then you probably know what you are doing and hence need less instruction.

[10]These are: **and, del, from, not, while, as, elif, global, or, with, assert, else, if, pass, yield, break, except, import, class, exec, in, raise, continue, finally, is, return, def, for, lambda, try, True, False,** and **None**.

the expression `x * y` on the right-hand side of the = sign to obtain 7.5, stores this result in memory, and binds the identifier specified on the left (i.e., z) to that value. Passed `x = x + 1`, the statement executes in a similar way (right to left), creating a new integer object 4 and *rebinding* x to that value.

Objects stored in memory have different *types*. The type of our objects can be queried using the built-in function `type()`:

```
>>> type(x)          # x = 4
<type 'int'>
>>> type(y)          # y = 2.5
<type 'float'>
```

The identifier x is bound to integer 4, while y is bound to floating point number 2.5. A floating point number is a number with a decimal point. Like most programming languages, Python distinguishes between floats and integers because integers are more efficient in terms of operations such as addition and multiplication.

Another common type of object is a *string*:

```
>>> s = "godzilla"           # Single or double quotes
>>> s.count("g")             # How many g's?  Returns 1
>>> s.startswith("god")      # Returns True
>>> s.upper()                # Returns "GODZILLA"
>>> s.replace("l", "m")      # Returns "godzimma"
>>> s2 = """
... Triple quotes can be used to create multi-line
... strings, which is useful when you want to
... record a lot of text."""
```

We are using some of Python's *string methods* (e.g., `count()`, `upper()`) to manipulate the string `"godzilla"`. Before discussing methods, let's introduce a fourth data type, called *lists*. Lists are *containers* used to store a collection of objects.

```
>>> X = [20, 30, 40]      # Bind X to a list of integers
>>> sum(X)                # Returns 90
>>> max(X)                # Returns 40
```

We can extract elements of the list using square brackets notation. Like most programming languages, the first index is 0 rather than 1, so `X[0]` references the first element of the list (which is 20), `X[1]` references the second element, and so on. In this context the integers 0, 1, 2 are called the *indexes* of the list. The list can be modified using indexes as follows:

```
>>> X[0] = "godzilla"   # Now X = ["godzilla", 30, 40]
>>> del X[1]            # Now X = ["godzilla", 40]
```

Lists can be *unpacked* into variables containing their elements:

```
>>> x, y = ["a", "b"]  # Now x = "a" and y = "b"
```

We saw that Python has methods for operations on strings, as in `s.count("g")` above. Here `s` is the variable name (the identifier), bound to string `"godzilla"`, and `count()` is the name of a string method, which can be called on any string. Lists also have methods. Method calls on strings, lists and other objects follow the general syntax

```
identifier.methodName(arguments)  # e.g., s.count("g")
```

For example, `X.append(3)` appends 3 to the end of `X`, while `X.count(3)` counts the number of times that 3 occurs in `X`. In IDLE you can enter `X.` at the prompt and then press the TAB key to get a list of methods that can be applied to lists (more generally, to objects of type `type(X)`).

2.1.3 Modules and Scripts

There are several ways to interact with the Python interpreter. One is to type commands directly into the prompt as above. A more common way is to write the commands in a text file and then run that file through the interpreter. There are many ways to do this, and in time you will find one that best suits your needs. The easiest is to use the editor found in IDLE: Open up a new window under the 'File' menu. Type in a command such as `print("Hello world")` and save the file in the current directory as hello.py. You can now run the file by pressing F5 or selecting 'Run Module'. The output `Hello world` should appear at the command prompt.

A text file such as hello.py that is run through an interpreter is known as a *script*. In Python such files are also known as *modules*, a module being any file with Python functions and other definitions. Modules can be run through the interpreter using the keyword **import**. Thus, as well as executing hello.py using IDLE's 'Run Module' command as above, we can type the following at the Python prompt:

```
>>> import hello  # Load the file hello.py and run
'Hello world'
```

When you first import the module `hello`, Python creates a file called hello.pyc, which is a byte-compiled file containing the instructions in hello.py. Note that if you now change hello.py, resave, and import again, the changes will not be noticed because hello.pyc is not altered. To affect the changes in hello.py, use `reload(hello)`, which rewrites hello.pyc.

There are vast libraries of Python modules available,[11] some of which are bundled with every Python distribution. A useful example is `math`:

```
>>> import math              # Module math
```

[11]See, for example, `http://pypi.python.org/pypi`.

```
>>> math.pi                     # Returns 3.141592653589793 1
>>> math.sqrt(4)                # Returns 2.0
```

Here `pi` is a float object supplied by `math` and `sqrt()` is a function object. Collectively these objects are called *attributes* of the module `math`.

Another handy module in the standard library is `random`.

```
>>> import random
>>> X = ["a", "b", "c"]
>>> random.choice(X)                # Returned "b"
>>> random.shuffle(X)               # X is now shuffled
>>> random.gammavariate(2, 2)       # Returned 3.433472
```

Notice how module attributes are accessed using `moduleName.identifier` notation. Each module has it's own *namespace*, which is a mapping from identifiers to objects in memory. For example, `pi` is defined in the namespace of `math`, and bound to the float $3.14 \cdots 1$. Identifiers in different namespaces are independent, so modules `mod1` and `mod2` can both have distinct attribute `a`. No confusion arises because one is accessed as `mod1.a` and the other is accessed as `mod2.a`.[12]

When `x = 1` is entered at the command line (i.e., Python prompt), the identifier `x` is registered in the interactive namespace.[13] If we import a module such as `math`, only the module name is registered in the interactive namespace. The attributes of `math` need to be accessed as described above (i.e, `math.pi` references the float object `pi` registered in the `math` namespace).[14] Should we wish to, we can however import attributes directly into the interactive namespace as follows:

```
>>> from math import sqrt, pi
>>> pi * sqrt(4)               # Returns 6.28...
```

Note that when a module `mod` is run from within IDLE using 'Run Module', commands are executed within the interactive namespace. As a result, attributes of `mod` can be accessed directly, without the prefix `mod`. The same effect can be obtained at the prompt by entering

```
>>> from mod import *    # Import everything
```

In general, it is better to be selective, importing only necessary attributes into the interactive namespace. The reason is that our namespace may become flooded with variable names, possibly "shadowing" names that are already in use.

[12]Think of the idea of a namespace as like a street address, with street name being the namespace and street number being the attribute. There is no confusion if two houses have street number 10, as long as we supply the street names of the two houses.

[13]Internally, the interactive namespace belongs to a top-level module called `__main__`.

[14]To view the contents of the interactive namespace type `vars()` at the prompt. You will see some stuff we have not discussed yet (`__doc__`, etc.), plus any variables you have defined or modules you have imported. If you import `math` and then type `vars(math)`, you will see the attributes of this module.

Modules such as `math`, `sys`, and `os` come bundled with any Python distribution. Others will need to be installed. Installation is usually straightforward, and documented for each module. Once installed, these modules can be imported just like standard library modules. For us the most important third-party module[15] is the scientific computation package SciPy, which in turn depends on the fast array processing module NumPy. The latter is indispensable for serious number crunching, and SciPy provides many functions that take advantage of the facilities for array processing in NumPy, based on efficient C and Fortran libraries.[16]

Documentation for SciPy and NumPy can be found at the SciPy web site and the text home page. These examples should give the basic idea:

```
>>> from scipy import *
>>> integral, error = integrate.quad(sin, -1, 1)
>>> minimizer = optimize.fminbound(cos, 0, 2 * pi)
>>> A = array([[1, 2], [3, 4]])
>>> determinant = linalg.det(A)
>>> eigenvalues = linalg.eigvals(A)
```

SciPy functions such as `sin()` and `cos()` are called *vectorized* (or *universal*) functions, which means that they accept either numbers or sequences (lists, NumPy arrays, etc.) as arguments. When acting on a sequence, the function returns an array obtained by applying the function elementwise on the sequence. For example:

```
>>> cos([0, pi, 2 * pi]) # Returns array([ 1., -1.,  1.])
```

There are also modules for plotting and visualization under active development. At the time of writing, Matplotlib and PyX are popular and interact well with SciPy. A bit of searching will reveal many alternatives.

2.1.4 Flow Control

Conditionals and loops can be used to control which commands are executed and the order in which they are processed. Let's start with the **if/else** construct, the general syntax for which is

```
if <expression>:     # If <expression> is true, then
    <statements>     # this block of code is executed
else:
    <statements>     # Otherwise, this one
```

The **else** block is optional. An *expression* is any code phrase that yields a value when executed, and conditionals like **if** may be followed by any valid expression. Expressions are regarded as false if they evaluate to the boolean value **False** (e.g., `2 < 1`),

[15] Actually a package (i.e., collection of modules) rather than a module.

[16] For symbolic (as opposed to numerical) algebra see SymPy or Sage.

to zero, to an empty list `[]`, and one or two other cases. All other expressions are regarded as true:

```
>>> if 42 and 99: print("foo")   # Both True, prints foo
>>> if [] or 0.0: print("bar")   # Both False
```

As discussed above, a single = is used for *assignment* (i.e., binding an identifier to an object), rather than *comparison* (i.e., testing equality). To test the equality of objects, two equal signs are used:

```
>>> x = y = 1                     # Bind x and y to 1
>>> if x == y: print("foobar")    # Prints foobar
```

To test whether a list contains a given element, we can use the Python keyword **in**:

```
>>> 1 in [1, 2, 3]       # Evaluates as True
>>> 1 not in [1, 2, 3]   # Evaluates as False
```

To repeat execution of a block of code until a condition fails, Python provides the **while** loop, the syntax of which is

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

Here a **while** loop is used to create the list `X = [1,...,10]`:[17]

```
X = []                   # Start with an empty list
i = 1                    # Bind i to integer object 1
while len(X) < 10:       # While length of list X is < 10
    X.append(i)          # Append i to end of list X
    i += 1               # Equivalent to i = i + 1
print("Loop completed")
```

At first `X` is empty and `i = 1`. Since `len(X)` is zero, the expression following the **while** keyword is true. As a result we enter the **while** loop, setting `X = [1]` and `i = 2`. The expression `len(X) < 10` is now evaluated again and, since it remains true, the two lines in the loop are again executed, with `X` becoming `[1, 2]` and `i` taking the value 3. This continues until `len(X)` is equal to 10, at which time the loop terminates and the last line is executed.

Take note of the syntax. The two lines of code following the colon, which make up the body of the **while** loop, are indented the same number of spaces. This is not just to enhance readability. In fact the Python interpreter *determines the start and end of code blocks using indentation.* An increase in indentation signifies the start of a code block,

[17] In the following code, the absence of a Python prompt at the start of each line means that the code is written as a script (module) and then run.

whereas a decrease signifies its end. In Python the convention is to use four spaces to indent each block, and I recommend you follow it.[18]

Here's another example that uses the **break** keyword. We wish to simulate the random variable $T := \min\{t \geq 1 : W_t > 3\}$, where $(W_t)_{t \geq 1}$ is an IID sequence of standard normal random variables:

```
from random import normalvariate
T = 1
while 1:                        # Always true
    X = normalvariate(0, 1)     # Draw X from N(0,1)
    if X > 3:                   # If X > 3,
        print(T)                # print the value of T,
        break                   # and terminate while loop.
    T += 1                      # Else T = T + 1, repeat
```

The program returns the first point in time t such that $W_t > 3$.

Another style of loop is the **for** loop. Often **for** loops are used to carry out operations on lists. Suppose, for example, that we have a list `X`, and we wish to create a second list `Y` containing the squares of all elements in `X` that are strictly less than zero. Here is a first pass:

```
Y = []
for i in range(len(X)):     # For all indexes of X
    if X[i] < 0:
        Y.append(X[i]**2)
```

This is a traditional C-style **for** loop, iterating over the indexes `0` to `len(X)-1` of the list `X`. In Python, **for** loops can iterate over *any* list, rather than just sequences of integers (i.e., indexes), which means that the code above can be simplified to

```
Y = []
for x in X:     # For all x in X, starting with X[0]
    if x < 0:
        Y.append(x**2)
```

In fact, Python provides a very useful construct, called a *list comprehension*, that allows us to achieve the same thing in one line:

```
Y = [x**2 for x in X if x < 0]
```

A **for** loop can be used to code the algorithm discussed in exercise 2.1.2 on page 14:

```
subset = True
for a in A:
```

[18]In text files, tabs are different to spaces. If you are working with a text editor other than IDLE, you should configure the tab key to insert four spaces. Most decent text editors have this functionality.

```
    if a not in B:
        subset = False
print(subset)
```

Here `A` and `B` are expected to be lists.[19]

We can also code the algorithm on page 14 along the following lines:

```
from random import uniform
H, p = 0, 0.5                    # H = 0, p = 0.5
for i in range(10):              # Iterate 10 times
    U = uniform(0, 1)            # U is uniform on (0, 1)
    if U < p:
        print("heads")
        H += 1                   # H = H + 1
    else:
        print("tails")
print(H)
```

Exercise 2.1.5 Turn the pseudocode from exercise 2.1.3 into Python code.

Python **for** loops can step through any object that is *iterable*. For example:

```
from urllib import urlopen
webpage = urlopen("http://johnstachurski.net")
for line in webpage:
    print(line)
```

Here the loop acts on a "file-like" object created by the call to `urlopen()`. Consult the Python documentation on iterators for more information.

Finally, it should be noted that in scripting languages **for** loops are inherently slow. Here's a comparison of summing an array with a **for** loop and summing with NumPy's `sum()` function:

```
import numpy, time
Y = numpy.ones(100000)     # NumPy array of 100,000 ones
t1 = time.time()           # Record time
s = 0
for y in Y:                # Sum elements with for loop
    s += y
t2 = time.time()           # Record time
s = numpy.sum(Y)           # NumPy's sum() function
t3 = time.time()           # Record time
print((t2-t1)/(t3-t2))
```

[19]Actually they are required to be iterable. Also note that Python has a `set` data type, which will perform this test for you. The details are omitted.

On my computer the output is about 200, meaning that, at least for this array of numbers, NumPy's `sum()` function is roughly 200 times faster than using a **for** loop. The reason is that NumPy's `sum()` function passes the operation to efficient C code.

2.2 Program Design

The material we have covered so far is already sufficient to solve useful programming problems. The issue we turn to next is that of design: How to construct programs so as to retain clarity and readability as our projects grow. We begin with the idea of functions, which are labeled blocks of code written to perform a specific operation.

2.2.1 User-Defined Functions

The first step along the road to good program design is learning to break your program up into functions. Functions are a key tool through which programmers implement the time-honored strategy of divide and conquer: problems are broken up into smaller subproblems, which are then coded up as functions. The main program then coordinates these functions, calling on them to do their jobs at the appropriate time.

Now the details. When we pass the instruction `x = 3` to the interpreter, an integer "object" with value 3 is stored in the memory and assigned the identifier `x`. In a similar way we can also create a set of instructions for accomplishing a given task, store the instructions in memory, and bind an identifier (name) that can be used to *call* (i.e., run) the instructions. The set of instructions is called a *function*. Python supplies a number of built-in functions, such as `max()` and `sum()` above, as well as permitting users to define their own functions. Here's a fairly useless example of the latter:

```
def f(x, y):          # Bind f to a function that
    print(x + y)      # prints the value of x + y
```

After typing this into a new window in IDLE, saving it and running it, we can then call `f` at the command prompt:

```
>>> f(2,3)                  # Prints 5
>>> f("code ", "monkey")    # Prints "code monkey"
```

Take note of the syntax used to define `f`. We start with **def**, which is a Python keyword used for creating functions. Next follows the name and a sequence of arguments in parentheses. After the closing bracket a colon is required. Following the colon we have a code block consisting of the function body. As before, indentation is used to delimit the code block.

Notice that the order in which arguments are passed to functions is important. The calls `f("a", "b")` and `f("b", "a")` produce different output. When there are many arguments, it can become difficult to remember which argument should be passed first, which should be passed second, and so on. In this case one possibility is to use *keyword* arguments:

```
def g(word1="Charlie ", word2="don't ", word3="surf."):
    print(word1 + word2 + word3)
```

The values supplied to the parameters are defaults. If no value is passed to a given parameter when the function is called, then the parameter name is bound to its default value:

```
>>> g()              # Prints "Charlie don't surf"
>>> g(word3="swim")  # Prints "Charlie don't swim"
```

If we do not wish to specify any particular default value, then the convention is to use **None** instead, as in x=**None**.

Often one wishes to create functions that *return* an object as the result of their internal computations. To do so, we use the Python keyword **return**. Here is an example that computes the norm distance between two lists of numbers:

```
def normdist(X, Y):
    Z = [(x - y)**2 for x, y in zip(X, Y)]
    return sum(Z)**0.5
```

We are using the built-in function zip(), which allows us to step through the x, y pairs, as well as a list comprehension to construct the list Z. A call such as

```
>>> p = normdist(X, Y)  # X, Y are lists of equal length
```

binds identifier p to the value returned by the function.

It's good practice to include a *doc string* in your functions. A doc string is a string that documents the function, and comes at the start of the function code block. For example,

```
def normdist(X, Y):
    "Computes euclidean distance between two vectors."
    Z = [(x - y)**2 for x, y in zip(X, Y)]
    return sum(Z)**0.5
```

Of course, we could just use the standard comment notation, but doc strings have certain advantages that we won't go into here. In the code in this text, doc strings are used or omitted depending on space constraints.

Python provides a second way to define functions, using the **lambda** keyword. Typically **lambda** is used to create small, in-line functions such as

```
f = lambda x, y: x + y  # E.g. f(1,2) returns 3
```

Note that functions can return any Python object, including functions. For example, suppose that we want to be able to create the Cobb–Douglas production function $f(k) = Ak^{\alpha}$ for any parameters A and α. The following function takes parameters (A, α) as arguments and creates and returns the corresponding function f:

```
def cobbdoug(A, alpha):
    return lambda k: A * k**alpha
```

After saving and running this, we can call `cobbdoug` at the prompt:

```
>>> g = cobbdoug(1, 0.5)   # Now g(k) returns 1 * k**0.5
```

2.2.2 More Data Types

We have already met several native Python data types, such as integers, lists and strings. Another native Python data type is the *tuple*:

```
>>> X = (20, 30, 40)   # Parentheses are optional
```

Tuples behave like lists, in that one can access elements of the tuple using indexes. Thus `X[0]` returns 20, `X[1]` returns 30, and so on. There is, however, one crucial difference: Lists are a *mutable* data type, whereas tuples (like strings) are *immutable*. In essence, mutable data types such as lists can be changed (i.e., their contents can be altered without creating a new object), whereas immutable types such as tuples cannot. For example, `X[0] = 3` raises an exception (error) if `X` is a tuple. If `X` is a list, then the same statement changes the first element of `X` to 3.

Tuples, lists, and strings are collectively referred to as *sequence* types, and they support a number of common operations (on top of the ability to access individual elements via indexes starting at zero). For example, adding two sequences *concatenates* them. Thus `(1, 2) + (3, 4)` creates the tuple `(1, 2, 3, 4)`, while `"ab" + "cd"` creates `"abcd"`. Multiplication of a sequence by an integer n concatenates with n copies of itself, so `[1] * 3` creates `[1, 1, 1]`. Sequences can also be *unpacked*: `x, y = (1, 2)` binds `x` to 1 and `y` to 2, while `x, y = "ab"` binds `x` to `"a"` and `y` to `"b"`.

Another useful data type is a *dictionary*, also known as a mapping, or an associative array. Mathematically, a dictionary is just a function on a finite set, where the programmer supplies the domain of the function plus the function values on elements of the domain. The points in the domain of a dictionary are referred to as its *keys*. A dictionary `d` is created by specifying the key/value pairs. Here's an example:

```
>>> d = {"Band": "AC/DC", "Track": "Jailbreak"}
>>> d["Track"]
"Jailbreak"
```

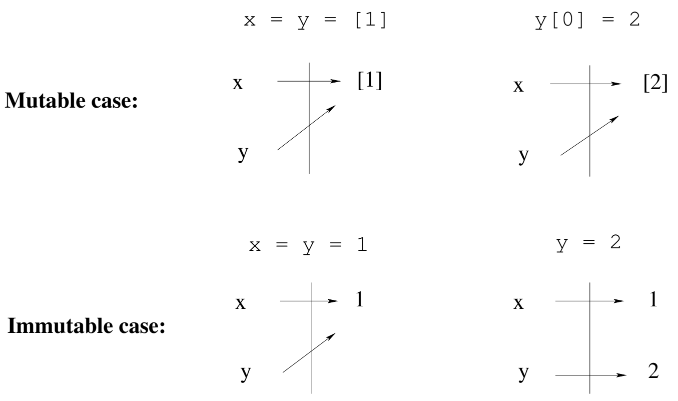

Figure 2.1 Mutable and immutable types

Just as lists and strings have methods that act on them, dictionaries have dictionary methods. For example, `d.keys()` returns the list of keys, and `d.values()` returns the list of values. Try the following piece of code, which assumes `d` is as defined above:

```
>>> for key in d.keys(): print(key, d[key])
```

Values of a dictionary can be any object, and keys can be any immutable object.

This brings us back to the topic of mutable versus immutable. A good understanding of the difference between mutable and immutable data types is helpful when trying to keep track of your variables. At the same time the following discussion is relatively technical, and can probably be skipped if you have little experience with programming.

To begin, consider figure 2.1. The statement `x = y = [1]` binds the identifiers `x` and `y` to the (mutable) list object `[1]`. Next `y[0] = 2` modifies that same list object, so its first (and only) element is 2. Note that *the value of* `x` *has now changed,* since the object it is bound to has changed. On the other hand, `x = y = 1` binds `x` and `y` to an immutable integer object. Since this object cannot be altered, the assignment `y = 2` *rebinds* `y` to a *new* integer object, *and* `x` *remains unchanged*.[20]

A related way that mutable and immutable data types lead to different outcomes is when passing arguments to functions. Consider, for example, the following code segment:

[20]You can check that `x` and `y` point to different objects by typing `id(x)` and `id(y)` at the prompt. Their unique identifier (which happens to be their location in memory) should be different.

```
def f(x):
    x = x + 1
    return x
x = 1
print(f(x), x)
```

This prints 2 as the value of `f(x)` and 1 as the value of `x`, which works as follows: After the function definition, `x = 1` creates a *global variable* `x` and binds it to the integer object 1. When `f` is called with argument `x`, a *local namespace* for its variables is allocated in memory, and the `x` inside the function is created as a *local variable* in that namespace and bound to the same integer object 1. Since the integer object is immutable, the statement `x = x + 1` creates a *new* integer object 2 and rebinds the local `x` to it. This reference is now passed back to the calling code, and hence `f(x)` references 2. Next, the local namespace is destroyed and the local `x` disappears. Throughout, the global `x` remains bound to 1.

The story is different when we use a mutable data type such as a list:

```
def f(x):
    x[0] = x[0] + 1
    return x
x = [1]
print(f(x), x)
```

This prints `[2]` for both `f(x)` and `x`. Here the global `x` is bound to the list object `[1]`. When `f` is called with argument `x`, a local `x` is created and bound to the same list object. Since `[1]` is mutable, `x[0] = x[0] + 1` modifies this object without changing its location in memory, so both the local `x` and the global `x` are now bound to `[2]`. Thus the global variable `x` is modified by the function, in contrast to the immutable case.

2.2.3 Object-Oriented Programming

Python supports both *procedural* and *object-oriented* programming (OOP). While any programming task can be accomplished using the traditional procedural style, OOP has become a central part of modern programming design, and will reward even the small investment undertaken here. It succeeds because its design pattern fits well with the human brain and its natural logic, facilitating clean, efficient code. It fits well with mathematics because it encourages *abstraction*. Just as abstraction in mathematics allows us to develop general ideas that apply in many contexts, abstraction in programming lets us build structures that can be used and reused in different programming problems.

Procedural programming is based around functions (procedures). The program has a *state*, which is the values of its variables, and functions are called to act on these data according to the task. Data are passed to functions via function calls. Functions return output that modifies the state. With OOP, however, data and functions are bundled together into logical entities called *abstract data types* (ADTs). A *class* definition is a blueprint for such an ADT, describing what kind of data it stores, and what functions it possesses for acting on these data. An *object* is an *instance* of the ADT; an individual realization of the blueprint, typically with its own unique data. Functions defined within classes are referred to as *methods*.

We have already met objects and methods. Recall that when the Python interpreter receives the instruction `X = [1, 2]`, it stores the data `[1, 2]` in memory, recording it as an *object of type list*. The identifier `X` is bound to this object, and we can use it to call methods that act on the data. For example, `X.reverse()` changes the data to `[2, 1]`. This method is one of several list methods. There are also string methods, dictionary methods, and so on.

What we haven't done so far is create our own ADTs using class definitions. You will probably find the class definition syntax a little fiddly at first, but it does become more intuitive as you go along. To illustrate the syntax we will build a simple class to represent and manipulate polynomial functions. The data in this case are the coefficients $(a_0, \ldots, a_N)$, which define a unique polynomial

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots a_N x^N = \sum_{n=0}^{N} a_n x^n \qquad (x \in \mathbb{R})$$

To manipulate these data we will create two methods, one to evaluate the polynomial from its coefficients, returning the value $p(x)$ for any x, and another to differentiate the polynomial, replacing the original coefficients $(a_0, \ldots, a_N)$ with the coefficients of p'.

Consider, first, listing 2.1, which sketches a class definition in pseudo-Python. This is *not* real Python code—it is intended to give the feeling of how the class definition might look, while omitting some boilerplate. The name of the class is `Polynomial`, as specified after the keyword **class**. The class definition consists of three methods. Let's discuss them in the order they appear.

The first method is called `initialize()`, and represents a *constructor*, which is a special method most languages provide to build (construct an instance of) an object from a class definition. Constructor methods usually take as arguments the data needed to set up a specific instance, which in this case is the vector of coefficients $(a_0, \ldots, a_N)$. The function should be passed a list or tuple, to which the identifier `coef` is then bound. Here `coef[i]` represents a_i.

The second method `evaluate()` evaluates $p(x)$ from x and the coefficients. We are using the built-in function `enumerate()`, which allows us to step through the `i, X[i]` pairs of any list `X`. The third method is `differentiate()`, which modifies the data

Listing 2.1 (`polyclass0.py`) A polynomial class in pseudo-Python

```python
class Polynomial:

    def initialize(coef):
        """Creates an instance p of the Polynomial class,
        where p(x) = coef[0] x^0 + ... + coef[N] x^N."""

    def evaluate(x):
        y = sum(a*x**i for i, a in enumerate(coef))
        return y

    def differentiate():
        new_coef = [i*a for i, a in enumerate(coef)]
        # Remove the first element, which is zero
        del new_coef[0]
        # And reset coefficients data to new values
        coef = new_coef
```

of a `Polynomial` instance, rebinding `coef` from $(a_0, \ldots, a_N)$ to $(a_1, 2a_2, \ldots, Na_N)$. The modified instance represents p'.

Now that we have written up an outline of a class definition in pseudo-Python, let's rewrite it in proper Python syntax. The modified code is given in listing 2.2. Before working through the additional syntax, let's look at an example of how to use the class, which is saved in a file called polyclass.py in the current working directory:

```python
>>> from polyclass import Polynomial
>>> data = [2, 1, 3]
>>> p = Polynomial(data)  # Creates instance of Polynomial class
>>> p.evaluate(1)         # Returns 6
>>> p.coef                # Returns [2, 1, 3]
>>> p.differentiate()     # Modifies coefficients of p
>>> p.coef                # Returns [1, 6]
>>> p.evaluate(1)         # Returns 7
```

The filename polyclass.py becomes the name of the module (with the ".py" extension omitted), and from it we import our class `Polynomial`. An instance `p` is created by a call of the form `p = Polynomial(data)`. Behind the scenes this generates a call to the constructor method, which realizes the instance as an object stored in memory, and binds the name `p` to this instance. As part of this process a namespace for the object is created, and the name `coef` is registered in that namespace and bound to the

Listing 2.2 (polyclass.py) A polynomial class, correct syntax

```
class Polynomial:

    def __init__(self, coef):
        """Creates an instance p of the Polynomial class,
        where p(x) = coef[0] x^0 + ... + coef[N] x^N."""
        self.coef = coef

    def evaluate(self, x):
        y = sum(a*x**i for i, a in enumerate(self.coef))
        return y

    def differentiate(self):
        new_coef = [i*a for i, a in enumerate(self.coef)]
        # Remove the first element, which is zero
        del new_coef[0]
        # And reset coefficients data to new values
        self.coef = new_coef
```

data `[2, 1, 3]`.[21] The attributes of `p` can be accessed using `p.attribute` notation, where the attributes are the methods (in this case `evaluate()` and `differentiate()`) and instance variables (in this case `coef`).

Let's now walk through the new syntax in listing 2.2. First, the constructor method is given its correct name, which is `__init__`. The double underscore notation reminds us that this is a special Python method—we will meet another example in a moment. Second, every method has `self` as its first argument, and attributes referred to within the class definition are also preceded by `self` (e.g., `self.coef`).

The idea with the `self` references is that *they stand in for the name of any instance that is subsequently created.* As one illustration of this, note that calling `p.evaluate(1)` is equivalent to calling

```
>>> Polynomial.evaluate(p, 1)
```

This alternate syntax is more cumbersome and not generally used, but we can see how `p` does in fact replace `self`, passed in as the first argument to the `evaluate()` method. And if we imagine how the `evaluate()` method would look with `p` instead of `self`, our code starts to appear more natural:

[21]To view the contents of this namespace type `p.__dict__` at the prompt.

```
def evaluate(p, x):
    y = sum(a*x**i for i, a in enumerate(p.coef))
    return y
```

Before finishing, let's briefly discuss another useful special method. One rather ungainly aspect of the `Polynomial` class is that for a given instance `p` corresponding to a polynomial p, the value $p(x)$ is obtained via the call `p.evaluate(x)`. It would be nicer—and closer to the mathematical notation—if we could replace this with the syntax `p(x)`. Actually this is easy: we simply replace the word `evaluate` in listing 2.2 with `__call__`. Objects of this class are now said to be *callable*, and `p(x)` is equivalent to `p.__call__(x)`.

2.3 Commentary

Python was developed by Guido van Rossum, with the first release in 1991. It is now one of the major success stories of the open source model, with a vibrant community of users and developers. van Rossum continues to direct the development of the language under the title of BDFL (Benevolent Dictator For Life). The use of Python has increased rapidly in recent years.

There are many good books on Python programming. A gentle introduction is provided by Zelle (2003). A more advanced book focusing on numerical methods is Langtangen (2008). However, the best place to start is on the Internet. The Python homepage (`python.org`) has links to the official Python documentation and various tutorials. Links, lectures, MATLAB code, and other information relevant to this chapter can be found on the text home page, at `http://johnstachurski.net/book.html`.

Computational economics is a rapidly growing field. For a sample of the literature, consult Amman et al. (1996), Heer and Maussner (2005), Kendrick et al. (2005), or Tesfatsion and Judd (2006).