PYTHON

AS A SECOND LANGUAGE

PSL

Steven L. Tanimoto

Dept. of Computer Science and Engineering

University of Washington

Seattle, WA 98195

tanimoto@cs.washington.edu

Based on a tutorial in the *IEEE Computer Society Ready Notes* Collection

Preface

Who this is for

This tutorial is intended for students who already know how to program a computer, but who are new to Python. It covers language basics and emphasizes aspects of the language that tend to support rapid prototyping or that are of interest from a programming languages perspective.

What this is about

Python is a general purpose programming language gaining in popularity. It has a variety of features that suit it well for applications in rapid prototyping, scripting, and artificial intelligence. Many professionals in scientific computation use it as a scripting language. It is being taught as a first language at a number of institutes of higher learning, including the Massachusetts Institute of Technology. It is also the featured language in the author's forthcoming text on image processing. Python supports automatic memory management, dynamic typing, imperative programming, object-oriented programming, and functional programming. This tutorial provides an introduction to the language with a focus on those aspects of the language most relevant to intelligent software applications.

What is special about this tutorial

Many people look to the web for tutorials on the Python language. There are several tutorials available, including both free ones and hardcopy books for sale at normal book prices. Most of these tutorials are generic and not optimized for people looking for a particular slant, such as for intelligent applications, for networking, or for numerical applications. This tutorial is aimed at the student who already knows how to program in a language such as C, C++ or Java. However, it does not assume the reader knows any Python yet. This tutorial focuses on the relatively recent versions of the language (3.1 and 3.2, as opposed to 2.5, 2.6 or 2.7).

About the author

Steven Tanimoto is a professor of computer science and engineering at the University of Washington, Seattle. He joined the faculty in 1977. He has taught courses on programming for 30 years, often using Lisp, but most recently using the Python language. The first version of his artificial intelligence text appeared in 1987 and sold approximately 30,000 copies, including book club sales. He was the Editor-in-Chief of the *IEEE Transactions on Pattern Analysis and Machine Intelligence* from 1986 to 1990. He is Fellow of the IEEE and of the International Association for Pattern Recognition.

January 2, 2012

Contents

Chapter 1

Overview

Python is a programming language growing in popularity that has characteristics that make it amenable to applications in rapid prototyping, scripting, and experimental programming such as that done in artificial intelligence research. Here, we introduce the language to readers who we assume already know how to program, but in another language, like Java, C, or C++. This document is a brief introduction to Python. Python has many capabilities and options that cannot be covered in a short tutorial. The focus here is on (a) language basics, and (b) features that illustrate programming language concepts.

This is not intended as an introduction to programming in general. We assume the reader is already familiar not only with the programming process, but with all the common programming constructs, such as sequences of instructions, the idea of syntax, conditional execution, loops, fundamental data types such as integers, floating-point numbers, boolean values, and strings, and function calling, function definition, and the notion of object-oriented classes, instances, and methods. We assume the reader is already familiar with data structures such as arrays, linked lists, stacks, and queues.

1.1 Why Python?

Python is a relatively young language, developed by Guido van Rossum around 1990. Here are some of the reasons why people use it:

- Interactivity. Python is good for experimental, conversational prototyping.
- Attractive syntax. Python's uncluttered syntax makes minimal use of parentheses and other punctuation. Instead, it primarily uses indentation to delimit blocks of code.
- Free availability. Python programming tools can be downloaded from the web without charge.

- Popularity. The fact that Python is popular means that lots of people are using it, and consequently, lots of people have contributed online documentation and examples.
- Good support for artificial intelligence programming. Python provides good support for the kinds of computing people do in artificial intelligence:
    1. strong support for symbolic computation and reasonable support for numerical computation.
    2. support for rapid prototyping. Ease of expression, rather than computational speed, is key here. With programming languages, there is generally a tradeoff between ease of expression on the one hand and correctness and reliability on the other. Python leans towards ease of expression.

Since Python is a relatively new language, its design was able to encompass many ideas pioneered by other programming languages: functional programming constructs such as mapping functions, automatic memory management including garbage collection, and object-oriented programming features such as classes and inheritance.

1.2 History

The Python language was originally created by Guido van Rossum during the early 1990s in Amsterdam at the Stichting Mathematisch Centrum. In 1995, when Guido moved to the Corporation for National Research Initiatives in Reston, Virginia, he continued to work on Python. During May of 2000, Guido van Rossum moved to BeOpen.com and formed their PythonLabs team. Later that year, in October, the team left BeOpen.com and jointed Digital Creations, which later changed its name to Zope Corp. The following year (2001) saw the creation of the Python Software Foundation, which owns the intellectual property associated with the Python language and tools. Zope Corporation supports PSF as a sponsoring member. Guido van Rossum still plays a prominent role within the Python development community, which now operates within an open-source framework. Guido is known as the "benevolent dictator for life" for Python.

There are two important versions of Python in use today (in 2012). One is known as Python 2.x, and here x is generally 5, 6 or 7. The other is known as Python 3.y, and here y is generally 0, 1, or 2. While programs developed in Python 2.5 would generally work fine when run with 2.6 and 2.7, they would typically not run in Python 3.0 (or later versions) without some changes being made to the code. The development community decided that fixing certain deficiencies of Python 2.x was worth some degree of backward incompatibility in the long run. This introduction to Python assumes that the reader is working with Python version 3.1.

1.3 Bibliographical Information

There are numerous books that teach the Python language. Two of the most popular are *Learning Python*, by Lutz and Ascher, and *Dive Into Python*, by Mark Pilgrim.

References

1. Lutz, M., and Ascher, D. 2003. *Learning Python*, 2nd ed. Sebastopol, CA: O'Reilly.

2. Pilgrim, M. 2004. *Dive Into Python*.Berkeley, CA: Apress.

3. van Rossum, Guido. 2004. *Python Tutorial*, Release 2.4 (Fred L. Drake, Jr., editor) `http://www.python.org/doc/current/tut/tut.html`.

Chapter 2

Logistics of Using Python

2.1 Downloading Python

Anyone with a good internet connection can obtain Python programming software by going to the official Python web site.

`www.python.org/download`

Choose a version compatible with your computer: either Windows or Unix and Mac OS X. Note that if you have a Macintosh with OS X 10.x, then you already have Python installed, because Python is included with the operating system software. However, it may not be version 3.1. Apple tends to ship older versions of Python with new Macintoshes. You can usually get a reasonably up-to-date version of Python for the Mac at the Python web site.

2.2 Installing Python

For Windows users: If you downloaded the Windows installer for Python, simply run that program by double-clicking on its icon. For Unix or Mac OS X, see the instructions at the python.org web site.

2.3 Starting Python

If you downloaded and installed Python with the IDLE development environment, simply go to your Windows Start menu, select Programs, and then select Python and then IDLE.

If you are using Python with a Mac under OS X or in Unix (e.g., Linux), and working in the command shell, then type `python` followed by the Enter key.

2.4 Exiting from Python

To quit Python, type Control-D at the Python prompt. To exit from IDLE you have an additional option for quitting: Close the IDLE window.

2.5 Using IDEs and Editors

The combination of Python with IDLE makes a very nice development environment. Most of what you need in order to use them is self-explanatory.

The most basic menu commands are these on the File menu.

- New Window. This creates a new window for editing text. If you create a new window, enter some text, and then (using Save As menu item on the File menu) save the text as a file with the `.py` extension, then the IDLE editor will recognize this as a python program and perform syntax highlighting automatically.
- Open... This will create a new window and load an existing file.
- Save. This saves the text in the current window to its associated file.
- Save As... This asks the user for a file name and location, and then associates the text with this file name and saves the text in the file.

Chapter 3

Interaction With Python

One of the most compelling features of Python is its support for interactive sessions. You can type expressions and commands at the Python prompt, and they will be evaluated or executed. Interacting with Python in this way is also a very effective way to develop familiarity with Python.

3.1 The Read-Eval-Print Loop

In the Python Shell window, you can type expressions at the prompt, enter them for evaluation, and see the results. Here is an example.

```
>>> 5 + 12
17
```

Interaction between the user and the Python interpreter takes place in a processing loop known as a *Read-Eval-Print loop*. Some refer to this by its initials (REPL). In each cycle of the loop, the system reads an expression that the user types in. This involves not only getting the characters typed by the user, but also parsing the expression and building an internal representation of it in the computer's memory. After this is complete, the computer *evaluates* the expression, using rules of the Python language to come up with a value that the expression specifies. For example, the value of the expression "5 + 12" is 17. Finally, the new value is printed or displayed on the screen so that the user can see it. These three steps: reading, evaluation, and printing, are repeated until the user terminates the session.

3.2 Calculator-Style Interaction

It is easy to use the Python interpreter as a kind of calculator. By typing in arithmetic expressions and watching the printed values, a user can do at least as much, if not more, than what can be done with a pocket calculator.

```
>>> 5 - 12
-7
>>> 5 * 12
60
```

However, there are some pitfalls to look out for. One of these is that old versions of Python (2.x) assumed that integers divided by integers should result in integers. The quotient of 17 divided by 5, in Python 2.7, is 3, rather than 3.4 as it is in Python 3.y.

```
>>> 17 / 5
3.4
```

In case you need your arithmetic expressions to work in both 2.x and 3.y, you can make either argument a floating-point real number. In Python 2.x, you would get

```
>>> 17 / 5.0
3.3999999999999999
```

but in 3.y this has been fixed:

```
>>> 17 / 5.0
3.4
```

If you do want integer division, with any remainder discarded, use the integer division operator:

```
>>> 17 // 5.0
3
```

Other calculation functions can be performed if we first import Python's `math` module, as the following example suggests.

```
>>> import math
>>> math.sqrt(5)
2.23606797749979
```

In addition to performing single operations, Python can directly evaluate complicated expressions that include nested subexpressions. The following example finds a solution to a quadratic equation: $3x^2 + 13x + 5 = 0$.

```
>>> (-13 + math.sqrt(13*13 - 4*3*5))/(2*3)
-0.42661558184824155
```

3.3 Assignment (binding)

After expression evaluation, perhaps the most useful feature of an interactive programming language is the ability to save expressions and values by binding them to variable symbols. This is accomplished using the equals sign as in the following examples.

```
>>>  x = 5
>>>  x + 12
17
>>>  y = x * x
>>>  y + 12
37
```

3.4 Value of the Last Expression

There is a special symbol that gets bound to the value of the most recently evaluated expression. This symbol is the underscore character. The following excerpt shows a continuation of the session begun in the previous example. First we examine the value of the underscore symbol. It turns out to be 37. Then we multiply the underscore symbol by itself, getting 1369 Repeating this leads to 1874161, since the underscore symbol gets rebound to the value 1369, and 1369 times 1369 is 1874161.

```
>>>  _
37
>>>  _*_
1369
>>>  _*_
1874161
```

Chapter 4

Language Essentials

4.1 Variables

The lexical elements in a Python program fall into the following categories: identifiers that are reserved words, other identifiers, operators, numbers, and special symbols.

An identifier is a string of characters that starts with a letter or the underscore symbol and then contains any number of additional letters, underscore symbols, or digits.

The reserved words are the following identifiers. They have built-in meanings and may not be used as variables.

```
False      class      finally    is         return
None       continue   for        lambda     try
True       def        from       nonlocal   while
and        del        global     not        with
as         elif       if         or         yield
assert     else       import     pass
break      except     in         raise
```

The character case is significant in Python, so the identifiers `x2` and `X2` are distinct. In other words, Python is case-sensitive.

An identifier that is not a reserved word can be used as a variable, and so can be bound to a value.

```
>>>  John = 27
>>>  John
27
>>>  and = 5
SyntaxError: invalid syntax
>>>  int(3.14)
3
>>>  int = 10
```

```
>>>  int(3.14)
TypeError: 'int' object is not callable
```

These examples show various attempts to create variables. First is the use of `John` as a variable storing the integer 27. This is fine. Then, trying to use the reserved word `and` as a variable leads to a syntax error. The identifier `int` is not a reserved word, but it does come with a predefined binding. We can rebind it to another value, but the original value will then be lost. Thus, trying to call `int` as a function after its function definition has been overwritten by a non-function value such as the number 10 causes an error.

Python's variables are *polymorphic*. This means that the type of value a variable may hold is not fixed and can change.

```
>>>  x = 4
>>>  x = x + 0.5
>>>  x
4.5
>>>  x = "five"
>>>  x
'five'
```

This feature is sometimes convenient in rapid prototyping, when a function might need to return a number or a string, depending on the circumstances. It is also often convenient not to have to enter type declarations into your program, as you might in a language such as Java or C. On the other hand, it requires that the programmer be careful to avoid mistakenly assigning one type of value to a variable when another might have been intended.

4.2 Numbers

Python provides several kinds of number representations. Most important are the integers and floats.

Integers in Python are normally represented in the decimal (base 10) system. Starting with Python version 2.2, there is only one kind of integer: an unlimited precision integer. With earlier versions, standard integers were restricted in magnitude and a special "long" integer was not. Now, the system will automatically convert between internal representations as needed so that small integers are efficiently represented, but integers needing many digits can also be accommodated. Python 3.1 thus has only one type of integer, and these integers have unlimited precision.

If needed, integers can be expressed in bases other than 10. For octal representation, precede the numerals with an extra zero and a lower-case o. For example 0o177 represents the decimal number 127. (Note: In Python 2.x, an integer was interpreted as octal if it started with a zero; the lower-case o was not used, and that led to unnecessary bugs.) For hexadecimal, precede the numerals with 0x. For example, 0xFF represents decimal 255. The case of the letters A

through F in this hexadecimal is not significant. The expression 0xff also means 255.

Floats are floating-point numbers in Python. There are two ways to specify a float in Python. One is to use a decimal point in the number: for example, .05 specifies five one-hundredths. The other is to use E notation: for example 5E-2 specifies the same number. Here the E means to multiply by the power of 10 given by the following integer, in this case we multiply by 10^{-2}. The two methods can be combined, as in, for example, `2.7654321E5` which stands for 276543.21.

Integers and floats can be added, multiplied, etc. When an integer is combined with a float, the resulting type is float. This tends to preserve the precision in a computation.

If it is necessary to explicitly convert from an integer to a float or vice-versa, the `int` and `float` functions can be used, as shown here:

```
>>> int(3.14)
3
>>> float(3)
3.0
```

Numbers can be compared with the standard relational operators, except for equality. The following examples illustrate this.

```
>>> 2 < 3
True
>>> 2.5 < 3
True
>>> 3.0 < 3
False
>>> 2 > 3
False
>>> 3 = 3
SyntaxError: can't assign to literal
>>> 3 == 3
True
```

So one has to use a double equals sign to compare for equality. Other numeric comparison operators include `<=` (less than or equal to), `>=` (greater than or equal to), and `!=` (not equal to).

4.3 Boolean Values and Operations

In early versions of Python (up through 2.2) `True` was represented by the integer 1 (or anything nonzero), and `False` was represented by the integer 0. Beginning with version 2.3, `True` and `False` are separate values. However, numbers can still be used in boolean contexts and will generally be converted automatically as needed.

The standard operations on boolean values are `and`, `or`, and `not`. Here are some examples of expressions using these operators.

```
>>>  2 < 3 and 3 < 4
True
>>>  2 < 3 < 4
True
>>>  not (2 == 2) or not (3 == 3)
False
```

4.4 Strings

Python strings can be specified either using single quotes or double quotes. Having a choice can be useful if the string itself needs to contain quotation marks.

```
>>>  name = 'John'
>>>  name
'John'
>>>  name2 = "John"
>>>  name2
'John'
>>>  x = "Programmer's choice"
>>>  x
"Programmer's choice"
>>>  "ab'cd\"ef"
'ab\'cd"ef'
```

Here we see that if no quotation marks are included in the string, it doesn't matter what kind of quotation marks are used to input the string. The default for printout is single quotes. However, when there is a single quotation mark (apostrophe) in the string, then double quotes are used for both input and output. If both kinds of quotation marks appear in the string, then one or the other will have to be escaped with a backlash.

Strings can be concatenated using the + operator.

```
>>>  "abc" + "def" + 'ghi'
'abcdefghi'
```

Strings are one type of sequence. Another type is the list.

4.5 Lists

In Python, a list is expressed by enclosing elements in square brackets, separating them with commas.

```
>>>  x = [0, 1, 2]
>>>  y = ['a', 'b', 'c']
```

```
>>>  x + y
[0, 1, 2, 'a', 'b', 'c']
```

Just as it does with strings, the + operator can be used to concatenate lists.

Lists can contain other lists as elements. For example, continuing the session of the previous example, we can embed the lists `x` and `y` in a new list.

```
>>>  [x, y]
[[0, 1, 2], ['a', 'b', 'c']]
```

We'll cover lists in more detail later.

4.6 Dictionaries

Python provides a form of associative memory called *dictionaries* that make it possible to store and retrieve information on the basis of keys that do not have to be array indices. A dictionary is sometimes called a *hash*. Here is an example.

```
>>>  atomic_mass = { 'H':1.00974, 'He': 4.002602,
                     'Li': 6.941, 'Be': 10.811}
>>>  atomic_mass['Li']
6.941
>>>  atomic_mass['Au'] = 196.96655
>>>  del atomic_mass['He']
>>>  for key, value in atomic_mass.items():
         print('Element ' + key + ' has atomic mass ' + str(value))

Element H has atomic mass 1.00974
Element Li has atomic mass 6.941
Element Au has atomic mass 196.96655
Element Be has atomic mass 10.811
```

Here we have constructed a dictionary by assigning a group of key:value pairs, separated by commas and all surrounded by curly brackets, to a variable. Individual associations can be created by assignments as in the example with key 'Au'. Individual values can similarly be retrieved. The `del` function can be used to delete an association from the dictionary. To retrieve all the keys and values in a loop, the dictionary's `items` method is used, as illustrated. The items don't come out in any standard order, so if you want them ordered according to keys, you would first retrieve the keys, convert them to a list, sort them, and loop through the sorted list accessing the corresponding values; one way of doing that is shown here:

```
>>>  chemicals = list(atomic_mass.keys())
>>>  chemicals.sort()
>>>  for element in chemicals:
         print(element + ' has mass ' + str(atomic_mass[element]))
```

```
Au has mass 196.96655
Be has mass 10.811
H has mass 1.00974
Li has mass 6.941
```

The keys for a dictionary can be numbers, strings, or any other data object that is " immutable." Lists are not immutable (which means that they are "mutable"), but tuples, such as `('John', 21)` are. A list can be converted into a tuple using the function `tuple`, and a tuple can be converted to a list using `list`.

4.7 Indentation and Block Structure

One of Python's most lauded features is its use of indentation to represent program structure. Where other languages might use curly bracket, parentheses, or `begin` and `end` pairs, Python typically begins a subprogram block with a colon, and then uses a reduction of indentation or in some cases a blank line to end the subprogram block. Here is an example involving a conditional expression.

```
if x < y:
    print(x)
else:
    print(y)
print(x+y)
```

When Python processes a program file, it detects the amount of indentation used by the programmer to set the subprogram apart from the surrounding code. Then it uses any change from this amount as a signal that either a new subprogram block is beginning or the current one is ending.

Programmers like this for two reasons. First, it forces all Python code to have at least some degree of readability. Secondly, it helps avoid certain errors that occur in other languages, such as unbalanced parentheses in Lisp or unbalanced curly brackets in Java.

At first, it might seem that since indentation matters to the Python interpreter, the programmer has no choice about formatting. However, Python allows some flexibility. As already mentioned, the amount of indentation is set by the programmer in the beginning of each subprogram block. Also, multiple statements can be put on one line by separating them with semicolons. Finally, a long line can be broken by placing a backslash at the end of each incomplete line and continuing the expression anywhere on the next line. For example, the following format is legal.

```
x =\
 1 + 2\
+ 3
y = 5; z = 6
```

In some situations, the backslash can be omitted without complaint from Python.

4.8 Defining Functions

New functions can be defined in Python using `def`. Here is an example of a simple function being defined.

```
>>>  def triple(x):
         return 3*x

>>>  triple(5)
15
```

A function definition can specify any number of arguments, from zero on up.

```
>>>  def hi():
         print('Hi there!')

>>>  hi()
Hi there!
>>>  def hello(name='friend'):
         print('Hello, ' + name)

>>>  hello()
Hello, friend
>>>  hello('John')
Hello, John
```

The first example here shows the definition and use of a function, `hi`, that takes zero arguments and prints a message. The second example shows how a similar function can have an optional argument. In the first call to `hello`, no argument is given, and the default value, `'friend'`, is used. In the second call, an argument, `'John'`, is given, and that is used as the value of `name`. Multiple arguments can be specified (either as required or as optional), separating them by commas in the parameter list. However, all the required arguments should be specified before any of the optional ones, as in the following example.

```
>>>  def f(a, b, c, d=1, e=2):
         return a+b+c+d+e

>>>  f(1,2,3,4)
12
```

Python functions can call themselves. That is, they can be *recursive*. Here is a Python implementation of the factorial function.

```
>>>  def fact(n):
         if n == 1: return 1
         return n * fact(n-1)

>>>  fact(5)
```

```
120
>>>  fact(25)
15511210043330985984000000
```

Recursive functions are especially important in processing lists and trees. (Trees include lists that contain sublists.)

4.9 The Special Value `None`

In the example functions given, some of them explicitly returned values, using the `return` statement. Two of them did not. The functions `hi` and `hello` printed some strings, but did not explicitly return any value. By default, a function returns a special value called `None`. This value serves as a sort of placeholder. In our interactive Read-Eval-Print loop, if the result of the evaluation phase is `None`, nothing is printed. On the other hand, if an explicit call to `print` is made with the value `None`, then the word "None" will actually be printed out. This can be seen below.

```
>>> a = None
>>> a
>>> print(a)
None
```

It is often useful to give a variable an initial binding of `None`, and then to rebind it in code that might or might not be successfully executed. Before illustrating that, however, we present the `if` construct, and material on looping and input/output.

4.10 Conditionals (if)

Python's conditional construct can be used in the standard ways with one or two cases, or, with the `elif` keyword it can be used to handle any number of cases. Here's an example showing a one-case situation.

```
if a > 0:
    print('positive')
```

Here's an example showing a two-case situation.

```
if a > 0:
    print('positive')
else:
    print('nonpositive')
```

Now here is a 3-case example.

```
if x < y:
    print(x)
elif y > x:
    print(y)
else:
    print('equal!')
```

Of course, `if` statements can be nested, as in the following example.

```
if x > 100:
    print('too big')
else:
    if x < 50:
        if x < 25:
            print('first quartile')
        else:
            print('second quartile')
    else:
        if x < 75:
            print('third quartile')
        else:
            print('fourth quartile')
```

4.11 Loops with `while` and `for`

To create loops, either the `while` or the `for` constructs can be used. Here's an example of a `while` loop used to print out the squares of the numbers 0 through 9.

```
n = 0
while n < 10:
    print(n*n)
    n += 1
```

By the way, the statement `n += 1` causes the value of `n` to be increased by one.

Here's a way to accomplish the same result using a `for` loop:

```
for n in range(10):
    print(n*n)
```

This takes advantage of the `range` function, which returns a list of integers. In this case the list starts with 0 and goes up to 9.

The `break` statement can be used to escape from a loop. Here is another way of expressing the computation above.

```
n = 0
while True:
    print(n*n)
    n += 1
    if n == 10: break
```

4.12 File Input and Output

Python makes it easy to read and write files of text. The following example creates a file `mydata.txt` in the current directory.

```
f = open("mydata.txt", "w")
f.write("This is some text.\n")
f.write("Here is a second line.\n")
f.close()
```

The variable `f` is used here to hold a file object returned by the `open` function. The first argument to `open` is a string: the name of the file. The second argument is either `"r"`, `"w"`, or `"a"`. These represent "read", "write", and "append" modes, respectively.

Once the file has been successfully opened, in this case created, data can be written to it. One simply calls the `write` method of the file object, passing a string argument to it. After all data has been written to the file, the `close` method should be called.

To read in the file created in the example above, the following code can be used. It begins by opening the file, but for reading instead of writing. We could have used the same variable `f` for the file object, but we are using `g` now, to emphasize that it is actually a different file object instance, since `open` has been called again.

```
g = open("mydata.txt", "r")
line1 = g.readline()
print(line1)
rest = g.read()
print(rest)
g.close()
```

This example shows two different ways of reading data from the file. One is to get the next line of text; we use `readline` for this purpose. The other is to get all the remaining data in the file, even if it extends across multiple lines; this is done using the `read` method. The string returned by `read` may thus contain any number of newline characters, where as the string returned by `readline` contains at most one, and it has one, it is at the end of the string. One additional method for reading that is relatively useful is `readlines`; it returns all the remaining text from the file but as a list of lines rather than as one long string.

There is a convenient way to loop through all the lines of a file that is already open for reading. It uses a combination of `for`, `in`, and the file object. The following example reads our file one line at a time and writes the data out with line numbers in front of each line.

```
g = open("mydata.txt", "r")
i = 0
for line in g:
  print("Line "+ str(i) + ": "+ line, end='')
  i += 1
```

Also, note the use of the optional "end" keyword argument to the print function. This overrides the default which is the newline character. If Python cannot complete any of these input or output operations, it raises an exception of type `IOError`. You can easily catch such errors and handle them in your code using the `try` and `except` construction discussed in the following section.

4.13 Exception Handling

When a runtime error occurs, the Python system "raises an exception." The normal flow of control is interrupted, and the call stack is unwound until an exception handler for a matching exception type is found. If no handler is found within the program, execution of the entire program is terminated with an error message.

You can provide an error handler for a possible exception using the `try` and `except` construct, as shown in the following example.

```
try:
  g = open("mydata.txt", "r")
except IOError:
  print('Python could not open the file mydata.txt.')
```

If the file `mydata.txt` does not exist in the current directory, or if its permissions prevent Python from reading it, the error exception will be raised, and the message following the `except IOError:` line will be printed.

We can extend this example a little bit to provide separate error messages for failure to open the file and failure to read its contents. The following does this. Note that no attempt to read the file is made if it was not opened successfully. To accomplish this, `g` is given an initial binding of `None` (which is treated as `False` within the boolean context of the `if` statement) and `g` is later tested to find out if it was successfully rebound.

```
g = None
try:
  g = open("mydata.txt", "r")
except IOError:
  print('Python could not open the file mydata.txt.')
if g:
  try:
    lines = g.readlines()
    print("The list of lines is:")
    print(lines)
  except IOError:
    print('Error while trying to read the data in the file.')
```

Exceptions are raised not only when input or output operations fail, but for such conditions as arithmetic overflow or division by zero. Here is an example of the latter.

```
try:
  5/0
except ZeroDivisionError:
  print('You cannot divide by zero. Sorry.')
```

The types of exceptions form a hierarchy. When the runtime system is looking for an error handler, it uses the first one to match the exception raised. Thus, if a program uses multiple handlers with overlapping coverage, it is typical to put more specific exception handlers closer to the code in which those exceptions might be raised, and to have more general error handlers dealing with broader scopes of the program.

If the identifier following the `except` keyword is empty (i.e., no exception type is given), then the most general exception type is assumed.

```
try:
  do_some_stuff()
except:
  print('There was some kind of exception raised.')
```

Chapter 5

Objects and Classes

Object-oriented programming is supported by Python, but there are several conventions which give it a different flavor from object-oriented programming in C++ or Java.

5.1 Defining Classes

Let's consider an example in which we create a class called `Employee` and some instances.

```
class Employee:
    "A class of company workers"
    count = 0
    def __init__(self, first_name='Jane', last_name='Doe', age=25):
        self.start_date = '3 JAN 2012'
        self.first_name = first_name
        self.last_name = last_name
        self.age = age
        Employee.count += 1
    def up_age(self):
        self.age += 1
    def describe(self):
        return self.first_name+' '+self.last_name+\
                ', age '+str(self.age)+', started '+\
                self.start_date
jane = Employee()
john = Employee('John','Smith',23)
john.up_age()
john.describe()
```

The first line of the definition's body is a string that documents the class. This documentation string is optional. Also, it can be a multiline string, if desired. A

multiline string can be specified by enclosing it within triple double quotes, for example

```
"""This is a
multiline
string"""
```

The second line of the body initializes a class variable, `count`, to zero. This illustrates the use of a variable within the class that is independent of any instance variables.

After that, an initialization function is defined to help construct instances of the class. This function is not required. However, without it, new instances would not have any values associated with them. The name of this function, `__init__` is important. If this exists, it is automatically called whenever a new instance is created. The first parameter of the function is `self`. This is customary for all Python instance methods. It doesn't have to have this name, but some class browsers might not work correctly if any other name is used. When the method is called, the instance is automatically used as the first argument of the function call, and it is referred to as `self` within the body of the method.

The assignment statements, starting with `self.start_date = '3 JAN 2012'`, all create instance variables and give them initial values. Note the use of dot notation here.

The statement `Employee.count += 1` shows how the class variable must be accessed, also using dot notation. Here, executing the statement causes the count to be increased by one.

The method `up_age` is an example of an instance method that changes the value of an instance variable. When called, it increases the value of an employee's age by 1. The instance method `describe` returns a string giving a detailed description of the employee.

After the class definition, the example shows two examples of creating instances. The first uses no arguments, which means that `__init__` gets called with only the `self` argument. The other parameters get the default values specified in the parameter list. The second instance, for `john`, gives particular values that get passed by the constructor to `__init__` thereby initializing it with non-default values.

The calls `john.up_age()` and `john.describe()` show the format for invoking methods on instances of objects. The last call results in the return of the string

```
'John Smith, age 24, started 3 JAN 2012'
```

5.2 Subclasses and Inheritance

One of the most important facilities in object-oriented programming is inheritance from classes to subclasses. Python supports this, as demonstrated in the following example, which subclasses the Employee class of our last example.

```
class Summer_Intern(Employee):
    "A subclass of Employee"
    def __init__(self, first_name='Jane', last_name='Doe',\
                 age=25, school='Univ. of Washington'):
        Employee.__init__(self, first_name, last_name, age)
        self.school = school
    def describe(self):
        return Employee.describe(self)+' from '+self.school

henry = Summer_Intern('Henry','Mann',19)
henry.start_date = '13 JUN 2011'
henry.up_age()
henry.describe()
```

This example defines `Summer_Intern` to be a subclass of `Employee`. The method `up_age` is inherited. The methods `__init__` and `describe` are overridden. The new `__init__` calls the parent `__init__` method by making an explicit function call with its fully qualified name. The new `describe` method does likewise. The value finally returned by the call `henry.describe()` is

```
'Henry Mann, age 20, started 13 JUN 2011 from Univ. of Washington'
```

This example also shows that the instance variables of an object can be accessed directly outside the class definition. The `start_date` attribute of `henry` is rewritten, in this case.

Multiple inheritance is supported by Python, although it can be hazardous to use it, because of the ease with which bugs can arise when working with unnecessarily complex class structures. To create a class that inherits from multiple classes, list all the parent classes with a definition of the form

```
class Child_Class(Parent_Class1, Parent_Class2, Parent_Class3):
```

The system will search at runtime for inherited methods or instance variables matching a given calling form or variable but performing a depth-first search (in Python versions 2.1 and before) or breadth-first search (in newer versions) among `Parent_Class1` and its ancestors, then `Parent_Class2` and its ancestors, etc., and it will stop as soon as a matching method or variable is found.

5.3 Callable Objects

Any instance of some class C can be called as if it were a function, provided that C has defined for it a method named `__call__` and whose first argument is `self`. This is an important feature for some types of experimental programming, because it allows objects to be used in a functional programming style. The following example shows the power of callable objects.

```
class Adder:
```

```
  def __init__(self, delta_x):
    self.increment = delta_x

  def __call__(self, x):
    return x + self.increment

add1 = Adder(1)   # a callable function that adds 1
add10 = Adder(10) # a callable function that adds 10
print(add1(5), add10(5), add1(5)*add10(5)) # prints 6 15 90
```

Here we see that the class `Adder` is used to create two instances, `add1` and `add10`. Because the class has a method named `__call__` defined appropriately, these instances can be used as functions. The expression `add1(5)` evaluates to 6, and the expression `add10(5)` evaluates to 15. These callable objects, which might as well be called functions, have been created at runtime. It should be clear that the construct of callable objects allows a program to create as many functions as it wants at runtime.

Building on the previous example, let's take advantage of the fact that objects (including callable objects) can themselves be arguments to functions. We now define a class called `Compose` that will construct for us the functional composition of two given functions.

```
class Compose:
  def __init__(self, f, g):
    self.f = f
    self.g = g

  def __call__(self, arg):
    return self.g(self.f(arg))

add11 = Compose(add1, add10)
print(add11(100))
```

When the new callable object is created, its two member variables, `f`, and `g` are assigned the values of `add1` and `add10`, which are themselves callable objects. When the new callable object (the value of variable `add11`), its `__call__` method is invoked, and the value to be returned is computed by first applying the value of `f` to 100 and then applying the value of `g` to that, resulting in 111.

Creating, at runtime, new functions from old functions, as illustrated here, is typical of functional programming.

Chapter 6

List Processing

Artificial intelligence programming largely began with the invention of the language Lisp by John McCarthy in about 1960. "Lisp" stands for List Processing. When Lisp was invented, it suddenly made it possible to perform experiments with non-numerical, symbolic computing with relative ease. Experiments with text processing, logical inference, and game playing soon followed. List processing remains an important technology not only for artificial intelligence but for almost all kinds of applications involving complex data structures.

6.1 Literal List Expressions

As we saw earlier, in Python, a list is represented by writing a sequence of data elements in square brackets, with commas separating the elements. Here are some examples of literal expressions that represent lists.

```
[0, 1, 2]
['a', 'few', 'words']
[['nesting', ['nesting']]]
```

Here are some expressions that create lists

```
>>>  list(range(5))
[0, 1, 2, 3, 4]
>>>  list(range(2, 5))
[2, 3, 4]
>>>  list(range(2, 7, 2))
[2, 4, 6]
>>>  "a b c".split()
['a', 'b', 'c']
```

The `range` function returns an iterable object that can be converted into a list of numbers. If given a single argument, the range starts at 0 and goes to one less than the argument. If given two arguments, it starts at the value of the

first argument and goes to one less than the second argument. If given three arguments, the third argument specifies a step value.

The `split` method of the string class returns a list of substrings found by splitting at the space characters in the given string.

Now let us examine methods for accessing the components of lists and composing new lists.

6.2 Accessing List Elements

The kth element of a list can be obtained using a subscript k in square brackets, as shown below. If a negative subscript is given, the indexing is done working backwards from the end of the list.

```
>>>  x = [2, 3, 5, 7, 11]
>>>  x[0]
2
>>>  x[1]
3
>>>  x[-1]
11
```

6.3 Slices of Lists

Parts of lists called *slices* can be accessed using an extension of the subscript notation, as follows.

```
>>>  x = [2, 3, 5, 7, 11]
>>>  x[0:3]
[2, 3, 5]
>>>  x[3:5]
[7, 11]
>>>  x[3:]
[7, 11]
```

Here the number before the colon gives the starting index for the sublist. The number after the colon, if there is one, gives the index of the element just past the last one in the sublist. If no second number is given, then the sublist goes to the end of the original list. Just as one can omit the second number, the first number can also be omitted, and it defaults to 0. It's also allowed to omit both numbers, leaving only the colon. This causes a complete copy of the list to be made; it's a convenient way to force Python to copy a list.

```
>>> L = ['a', 'b', 'c']
>>> M = L[:]
>>> M.remove('b')
>>> M
```

```
['a', 'c']
>>> L
['a', 'b', 'c']
```

In the above example, we bind `M` to a copy of `L` and then delete the second element from the copy. The original list is unaffected.

6.4 Heads and Tails of Lists

Many list processing algorithms work with functions that access the "head" and "tail" of a list. To get the head of list, we simply access its element in position 0, and to access the tail of the list, (suppose it is represented by `L`), we use the expression `L[1:]`.

To compose a list from a new head and an existing tail, we can use an expression of the form `[head]+tail`. For example, we might have

```
>>> head = 'a'
>>> tail = ['b', 'c']
>>> [head]+tail
['a', 'b', 'c']
```

In list processing, this is known as a *cons* operation, the construction of a new list from a given head and tail.

6.5 Lengthening Lists

When building up lists, several methods are available. We have already seen the use of `+` for list concatenation. Here it is again with some alternatives:

```
>>> L = ['a', 'b']
>>> L + ['c', 'd']
['a', 'b', 'c', 'd']
>>> L
['a', 'b']
>>> L += ['s']
>>> L
['a', 'b', 's']
>>> L.extend(['o', 'r', 'b'])
>>> L
['a', 'b', 's', 'o', 'r', 'b']
>>> L.append('s')
>>> L
['a', 'b', 's', 'o', 'r', 'b', 's']
>>> L.pop()
's'
>>> L
['a', 'b', 's', 'o', 'r', 'b']
```

The first example shows that concatenation alone does not change the value of `L`, only creating a temporary result. However, the use of `+=` causes `L` to be rebound to the result of concatenation. Another way to put new elements onto the end of a list is with the `extend` method, which takes a list of elements to be added as additional elements. On the other hand, the `append` method adds just one element to the end of the list. If the list is considered to be a stack whose top is at the end, then `append` is equivalent to a `push` operation. Although the name `push` is not predefined in Python, the corresponding method, `pop`, is defined, as the example shows.

6.6 List Membership and Modification

To test whether some value is an element of a list, use the `in` operator as follows

```
>>> 'b' in ['a', 'b', 'c']
True
>>> 0.0 in [0, 1, 2]
True
>>> 0.01 in [0, 1, 2]
False
>>> [0] in [0, 1]
False
>>> [0] in [[0], [1]]
True
```

The following examples show various other list methods.

```
>>> L = [9, 5, 7]
>>> L.insert(2, 'x')
>>> L
[9, 5, 'x', 7]
>>> L.index('x')
2
>>> L.remove('x')
>>> L
[9, 5, 7]
>>> L.reverse()
>>> L
[7, 5, 9]
>>> L.sort()
>>> L
[5, 7, 9]
```

To insert an element at an arbitrary position in a list, use `insert` with the desired position and the element to add. To find the position where some element occurs in a list, use the `index` method. If the element isn't in the list, an error is raised. To remove the first occurrence of an element, the `remove` method can be used.

The order of elements in the list can be changed using the `reverse` and `sort` methods.

6.7 List Comprehensions

Python offers a very elegant language construct for representing new lists in terms of others. It is called the *list comprehension*, and it often has the form
`[` *expression* {`for` *variable* `in` *list-expression*}`]`
where the variable occurs somewhere in the first expression.

Here is an example.

```
[x*10 for x in range(5)]  # denotes [0, 10, 20, 30, 40]
```

Here we see one list defined as the result of multiplying each element of the other list by 10. As we will see in a later chapter, this can also be expressed in a functional notation using `map`. However, using list comprehension gives us a more concise representation.

List comprehensions can be more complex. Here is the general form of a list comprehension.
`[` *expression* {`for` *variable* `in` *iterable*} {*lc-clause*}`*]`
Thus there can be any number of *lc-clause* parts of a list comprehension. Each *lc-clause* has one of two forms:
`for` *variable* `in` *iterable*
or
`if` *expression*
In the next example, there is an *lc-clause* that references a second set.

```
[(x, y) for x in ['a','b'] for y in [0, 1]]
 # denotes [('a', 0), ('a', 1), ('b', 0), ('b', 1)]
```

This example, shows how a list comprehension can easily be used to express the cartesian product of two given sets. We have a set of all possible ordered pairs where the first element comes from the set `['a', 'b']` and the second comes from the set `[0, 1]`.

In our third example, an *lc-clause* using `if` performs a filtering function within the list.

```
[x*10 for x in range(5) if x > 1]  # denotes [20, 30, 40]
```

The *iterable* used in a list comprehension does not have to be a list. For example, it could be a string.

```
[c for c in "ATOMIZE"] # results in
  # ['A', 'T', 'O', 'M', 'I', 'Z', 'E']
```

It is possible for a programmer to define new iterators that can be used in list comprehensions. Any object `x` that implements a method `x.next()` and that raises a `StopIteration` exception when no more elements are available can be considered to be an iterator.

Chapter 7

Simple String Processing

In this section, we present some additional methods for working with strings, beginning with a review of concatenation.

7.1 String Concatenation

As mentioned earlier, we can concatenate two strings using the `+` operator. As the following example shows, an existing string can be extended using the `+=` operator.

```
>>>  s1 = 'abc'
>>>  s2 = 'def'
>>>  s1 + s2
'abcdef'
>>>  s1 += s2
>>>  s1
'abcdef'
```

Just as strings can be added, a sort of multiplication can be performed between a string and an integer.

```
>>>  'abc' * 3
'abcabcabc'
```

In general when a string is multiplied by a number n, the result of n copies of the string concatenated together. (The same sort of multiplication can be done with lists, by the way.)

7.2 Substring Extraction

Just as we can take individual elements or slices of lists, we can take individual characters or substrings of strings. The following illustrates this.

```
>>> S = 'abcdef'
>>> S[1]
'b'
>>> S[1:4]
'bcd'
>>> S[2:]
'cdef'
>>> S[:5]
'abcde'
```

If a string needs to be copied, an expression such as `S[:]` will do the trick.

7.3 String Comparison

Strings can be compared for equality with the `==` operator, as for other Python objects. They can also be compared with the same operators (`<`, `<=`, etc.) used for numbers. The results depend upon the ASCII codes of the characters involved. For alphabetical comparison of strings with different character cases, they should be converted to a common case before comparison.

```
>>> 'appleseed' < 'Jonny'
False
>>> 'appleseed'.lower() < 'Jonny'.lower()
True
```

A simple form of string matching can be performed using the `find` method, which searches a string for the first occurrence of another string as a substring of the first. If such an occurrence exists, the position where the occurrence begins is returned. Otherwise, -1 is returned.

```
>>> S = 'appleseed'
>>> S.find('plese')
2
>>> S.find('pear')
-1
>>> S.find('e')
4
>>> S.find('e', 5)
6
```

The last example shows that we can require the matching to begin at a given position in the string. One use for this is when we want to find occurrences other than the first occurrence.

More elaborate string matching is possible using the regular expression module described later.

7.4 Getting Input from the User

Sometimes, interactive programs need to be able to input lines of text from the user. Python provides the `input` function for this purpose.

```
>>>  text = input('Type something here: ')
Type something here: Beautiful weather today!
>>>  text
'Beautiful weather today!'
```

The argument to `input` is the string that will be used as a prompt. Exactly one line of text is taken from the user and returned.

7.5 String Formatting

Python has a powerful means of producing strings that involve values. For readers familiar with the C language's `printf` function, this will look familiar.

```
>>>  name = 'John'
>>>  age = 18
>>>  height = 176.5
>>>  s = 'Student %s, aged %d is %.2fm tall.' % (name, age, height)
>>>  s
'Student John, aged 18 is 176.50m tall.'
```

The formatting operator requires a format string preceding it and a "tuple" of expressions after it. There should be one expression in the tuple for each data directive in the format string. In this example, the data directive `%s` indicates that a string value should replace it. The directive `%d` specifies that an integer (in decimal – base 10) should go here. The directive `%.2f` indicates that a float should be put here, with two places after the decimal point, and as many before the decimal point as needed.

Chapter 8

Modules and Scopes of Bindings

8.1 Modules

A Python *module* is a library of code, created in a separate file, intended be used in other programs. We have already seen examples of the use of modules, where some particular function needs to be imported so that it can be used. One example was the use of the `choice` function from the module named `random`.

```
import random
x = random.choice(range(100))
```

Besides helping developers to provide and use additional functionality, the module mechanism serves several other purposes: (1) it allows separate compilation, meaning that all of the imported code can be compiled in advance of the time it is needed; (2) it allows separation of name spaces, so that symbols used within the module need not interfere with those in the hosting program; (3) it allows arbitrary load-time computation to be performed. The first of these allows a module to be compiled once and then loaded, relatively quickly, when needed. The second permits encapsulation of the symbols used in the module, for the same reasons that object-oriented programming and modular development promote encapsulation: name-space protection and reduction of complexity to users of modules. The third purpose obviates the need to explicitly call module initialization code; importing the module is usually enough.

There are two different forms used for importing modules. Let's consider them in turn. The first is considered safer and is usually the preferred form in engineering larger programs. The second may be preferred in interactive sessions and small programs because of its convenience. The first form has already been shown above.

With this form, the module is imported, and the symbols that it uses are made available within the current session via a dictionary that is bound to the

symbol `random`. Using the `choice` function within the module requires using a fully-qualified name, i.e., first referencing the dictionary bound to `random` and then the dictionary entry for `choice` within it. Although using the fully-qualified name may seem more cumbersome, the name space for the module is kept distinct from that of the session or the main program, and that is a good precaution to take when a number of modules are involved. For a list of symbols that are available in the module, the dictionary bound to `random` can be listed as follows:

```
import random
dir(random)
```

The opposite style of importing is illustrated in the next example.

```
from random import *
x = choice(range(100))
```

Here, all the symbols defined at the top (global) level within the random module are entered into the main program's dictionary directly. If any object already had the name `choice` its reference would be overwritten during the importing. Not only `choice`, but 51 other symbols are also imported, since the `*` tells Python to bring in all the available symbols from `random`.

The risk of overwriting something in such an import can be reduced by specifying specific symbols to import. This is shown in the following.

```
from random import choice
x = choice(range(100))
```

A comma-separated list of names can follow the `import` keyword here.

Before we leave the subject of importing modules, there is a variant of the first form of importing that can ease the pain of code refactoring when the name of the module is changed during development. Suppose that a module MyModule is undergoing development and, to keep file versions straight, we are appending version numbers to the file name, e.g., MyModule001.py, MyModule002.py, etc. To avoid changing the module name, in the host program, in possibly many fully-qualified references to symbols in the module, we can declare an alias for the module in the `import` statement. Here is an example.

```
import MyModule002 as MyModule
x = MyModule.do_stuff()
```

When the name of the module gets changed to MyModule003, the main program need only be changed in one place: in the `import` statement, because there is an alias that is used in all of the references to symbols in the module.

8.2 Bindings and Scopes

A *binding* is an association between a symbol (i.e., a variable) and a value. Bindings can be created by assignment, as in, for example, `x = 5`. Bindings

are also created when an argument to a function gets tied to the corresponding parameter in the definition of the function when the function is called. The *scope* of a binding is the region of the program in which a particular binding is accessible.

Python variable bindings can have either of two kinds of scopes. Top-level bindings are said to have "global" scope and can be accessed anywhere in the program. Variables that are bound within a function definition have locally scoped bindings by default. However, such a variable use can be made to refer to a global binding through the use of the keyword `global`.

```
w = 2
x = 7
y = 13
z = 25
def foo(x):
   y = x + w
   global z
   z = x + w
   return y + z

print('foo(0)=',foo(0))
print('w=',w,'; x=',x,'; y=',y,'; z=',z)
# foo(0)= 4
# w= 2 ; x= 7 ; y= 13 ; z= 2
```

This example starts by creating four global bindings: one for each of `w`, `x`, `y`, and `z`. In the parameter list for the function `foo`, a local binding for `x` is established that has as its scope the body of the definition. In the first line of the body, the variable `y` is mentioned. Since it is inside the body, a new, local binding is established for it. On the next line, `z` is mentioned, but the keyword `global` indicates that any mention of `z` in the body of the function should refer to the global binding of `z`. So the global binding gets altered by the line `z = x + w`.

When the function `foo` is called with 0 as the argument, the local binding of `x` gets value 0, and the local binding of `y` gets 0 + 2, which is 2. The binding for `z` receives the same value, since the expression being evaluated is identical to that for `y`. The value returned is the local value of `y`, which is 2, added to the global value of `z`, which is now 2. Therefore the value returned by the function is 4. At the end of the example, all the global bindings except `z` have their original values.

Chapter 9

Mathematical Operations

9.1 Arithmetic

Python provides a variety of features for mathematical computing. As we saw earlier, the standard operations of addition, subtraction, multiplication, and division can be specified using `+`, `-`, `*`, and `/`, with the caveat that `/` represents the quotient in integer division when both of its arguments are integers. To get the remainder when dividing `x` by `y`, use the expression `x % y`. This is also known as the "mod" operator.

9.2 Complex Numbers

Although they are not used very often in artificial intelligence, except when doing signal processing or mathematical applications, complex numbers are supported directly in Python. The complex number with real part a and imaginary part bi is represented in Python as `a + bj`. Thus Python uses the letter `j` to represent the square root of -1. Here are some examples:

```
>>>  2 + 3j
(2+3j)
>>>  (2 + 3j) * (5 + 7j)
(-11+29j)
```

9.3 Standard Mathematical Functions

Various standard mathematical functions are available in the `math` module. For a list of those available, we can execute the following:

```
>>>  import math
>>>  dir(math)
```

This prints out a long list of functions and named constants, including 'acos', 'asin', 'atan', 'atan2', 'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp', 'fabs',
'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log', 'log10',
'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt',
'tan', 'tanh'.

9.4 Random Numbers and Choices

If random numbers are needed, methods for generating them can be found in the `random` module. The `randint` method returns random integers as the following example shows. Also shown are the `choice` method and the `random` method.

```
>>>  from random import *
>>>  randint(1, 10)
7
>>>  choice(['yes', 'no', 'maybe'])
'yes'
>>>  random()
0.651771744811
```

The two arguments to `randint` specify the lowest and highest possible integers that can be returned. The function chooses a random integer in that range. The argument to `choice` is a list of alternatives. One will be randomly selected. The method `random` takes no arguments, and it returns a random floating-point number between 0.0 and 1.0.

Here are some more examples of using these techniques.

```
>>>  from random import *
>>>  def toss_die():
       return choice(['.', '..', '...',
                      '....', '.....', '......'])

>>>  toss_die()
'....'
>>>  toss_die()
'.'
>>>  normalvariate(100, 10)
103.39922768303251
```

The function `toss_die` returns a string with a random number of dots between 1 and 6, inclusive. The function `normalvariate` takes two arguments representing the mean and standard deviation of a normal (Gaussian) distribution. A randomly generated element from that distribution is returned.

Chapter 10

Advanced String Processing

Much of the recent work in artificial intelligence has focused on the Internet. The availability of large bodies of text on the web has increased the importance of good text-handling facilities. Python provides a module that lets the programmer use the full power of regular expressions for pattern matching and string manipulation.

10.1 Regular Expression Module

A simple example of using regular expression matching is given in the following:

```
>>> from re import search
>>> if search('love', 'She loves you.'):
       print("All's fair in love and war.")

All's fair in love and war.
```

This example begins by importing a particular function, `search`, from the regular expression module, which is named `re`. In the next line, `search` is called with two string arguments. The first represents a regular expression. In this case the regular expression is just a string of characters to look for in the other string. Since `'love'` does occur in the other string, the result is considered true, and the `print` statement on the next line is executed.

There is a related function, `match`, that works like `search`, but only succeeds if the pattern occurs at the beginning of the string.

Regular expressions can involve any of many special constructs that represent components of patterns. A few of the pattern constructs are given in the table below. A full discussion of regular expressions is beyond the scope of this Python introduction. Note that regular expressions are a standard means of pattern

matching and the are used in many languages besides Python, such as Perl and Java.

```
Code  What it matches
----  ---------------
\w    alphanumeric character (letter or digit)
\W    non-alphabetic character
\s    whitespace character
\S    non-whitespace character
\d    digit
\.    any character
\\    backslash
\b    a break before or after a word
+     one or more occurrences of the preceding expression
*     zero or more occurrences of the preceding expression
?     zero or one occurrences of the preceding expression
|     either the expression before or the expression after
(     begin subexpression
)     end subexpression
```

10.2 Returning and Replacing Matched Substrings

Suppose we want to find and return all the nonnegative integers occurring in a string. We want all the digits of each integer, too. The regular expression that matches a nonnegative integer is `\d+` and in Python it is `'\\d+'`. The `findall` function in the `re` module returns a list of all matching substrings for a given pattern and string. We can also replace each matched substring with something else. Here's an example.

```
>>>  from re import *
>>>  findall('\\d+', 'There are 5280 feet in 1 mile.')
['5280', '1']
>>>  sub('\\d+', 'some', 'There are 5280 feet in 1 mile')
'There are some feet in some mile'
```

If computational speed is important, it is possible to compile a regular expression and then repeatedly apply the pattern without recompiling it each time. See the Python documentation if you need this feature.

10.3 Raw String Expressions

One of the challenges of using regular expressions in Python is the fact that many patterns need to include backslash characters. In Python, the backslash character, within a string, usually serves as an escape character. To include

a backslash character in a string literal, the backslash must be doubled. For example, the string consisting of a backslash followed by a letter `d` could be specified in Python as `'\\d'`. In order to avoid having to double the backslashes, we can use what is known as a *raw string*. To do this, immediately precede the string by the letter `r`. For example:

```
>>>  r'\d+'
'\\d+'
>>>  print(r'\d+')
\d+
```

Raw strings are particularly helpful when the regular expression itself must be designed to match backslash characters, as one might do if building a preprocessor for LaTeX source files or some other language that uses lots of backslashes. A regular expression that matches a single backslash is `\\` which without raw strings would be `'\\\\'` in Python. With raw strings, it is just `r'\\'`.

Chapter 11

Functional Programming Techniques

11.1 Motivation

In rapid prototyping, we often need to work with lists of data. A typical manipulation of a list is to transform it into another list in which each element has been processed by a function. A style of programming called "functional programming" suits such a pattern very well. However, functional programming is a very powerful kind of programming in its own right, whether operating on lists or any other kind of data.

In this chapter, we'll review mechanisms for defining functions and cover a new one: the "lambda expression." Then we'll consider a function called `map` that operates with other functions to transform lists, and we'll compare the use of `map` with the use of list comprehensions. Then, we'll jump into functional programming more generally. We'll consider the classical functional programming operations known as "currying" and "composition."

11.2 Revisiting the Defining of Functions

The most straightforward way to define a new function is using `def`. An alternative method makes a callable object. The examples below show these alternative approaches plus a third one.

```
def double1(x):
  return 2*x

class foo:
  def __call__(self, x):
    return 2*x
```

```
double2 = foo()

double3 = lambda x: 2*x

print(double1(7), double2(7), double3(7))
# prints   14 14 14
```

The third method for defining a function uses a *lambda expression*. It begins with the keyword `lambda`. Then it has one or more variable names representing parameters; in this example, there is a single parameter, `x`. If there were two or more parameters, they would be separated by commas. After the parameter(s) is a colon. Following the colon is the body of the lambda expression. The body provides a specification of what value the function returns. The lambda expression does not use the `return` keyword.

As we'll see shortly, a lambda expression is sometimes more convenient than the use of `def` or a callable object, because it is short, can be written in the location in which it is used, and it does not require making up a name for the function.

11.3 List Transformations

Since producing one list from another is a common need in artificial intelligence programming, this will be a good way to illustrate how we can use functional programming.

We begin by reviewing a traditional way of transforming a list using a recursive function definition. Suppose we wish to add 1 to each element of a list. We could write a recursive function to do this as follows.

```
def add_1_to_each(L):
   if L==[]: return []
   return [L[0]+1] + add_1_to_each(L[1:])

add_1_to_each([4, 7, 11.0])
# returns  [5, 8, 12.0]
```

However, there is an alternative that is more concise. It uses a built-in function, `map` to apply another function to each element of a list. In order to get an appropriate other function, we could define it in the usual way as follows.

```
def add1(x): return x + 1

list(map(add1, [4, 7, 11.0]))
# [5, 8, 12.0]
```

But there is another way to do this, without having to create a full-blown function definition with a name. Instead, we use the `lambda` keyword and the lambda expression syntax. This allows us to create a small, anonymous function right

in the place where we need it. The following solution is obviously more concise than the previous ones.

```
list(map(lambda n: n+1, [4, 7, 11.0]))
# [5, 8, 12.0]
```

The anonymous function is given by the expression `lambda n: n+1`. This function takes a single argument, but such functions could take multiple arguments. The value it returns is specified in the expression following the colon. The keyword `return` is not needed. Here's an example of mapping a function of two arguments.

```
list(map(lambda a, b: 2*a+b, [1, 2, 3], [4, 5, 6]))
# [6, 9, 12]
```

Here each element of the output list is created by applying the lambda function to a pair of corresponding elements in the lists of arguments. The 6 is the value of 2*1 + 4. Note that if the lambda function takes two arguments, then there must be two lists of values following the function in the call to `map`.

List comprehensions, which we introduced earlier in the chapter on lists, provide an alternative means of specifying list transformations. The two examples we've just given using lambda expressions can be expressed using list comprehensions as follows.

```
[n+1 for n in [4, 7, 11.0]]
# [5, 8, 12.0]
[2*a+b for (a, b) in zip([1, 2, 3], [4, 5, 6])]
# [6, 9, 12]
```

This last example uses the `zip` function to pair off corresponding elements from each of two lists into a list of pairs. By itself, `zip` gives us an iterable object that can be understood by converting it to a list.

```
list(zip([1, 2, 3], [4, 5, 6]))
# [(1, 4), (2, 5), (3, 6)]
```

The variables used as arguments in a lambda expression are new local variables that don't overwrite similarly named local variables in the environment of their use. Also, the variables `n`, `a`, and `b` in our examples above with list comprehensions are new instances of variables; any previous bindings for them are preserved.

A basic limitation of lambda expressions in Python, unlike in Common Lisp and Scheme, where they are also found, is that, in Python, the body of a lambda expression cannot contain control structures such as loops or conditionals. It is possible to work around this restriction through the use of "computed conditionals" and embedded function calls where the called functions can contain arbitrary uses of control structures. However, this can be awkward, and writing named functions for this purpose loses the main attraction of lambda expressions by requiring named functions and functions no longer defined exactly where they are used.

11.4 Reductions of Lists

Another application of functional programming to lists is in computing "reductions." A reduction is a computation that starts with a list and ends up with a scalar. That is, we start with many values and end up with a single value. A simple example is adding up all the elements in a list. A traditional (imperative) style of accomplishing this is to use a loop:

```
lst = [1, 3, 5, 7]
sum = 0
for x in lst: sum+= x
print(sum)
# 16
```

However, using the functional approach, no explicit iteration is needed.

```
defun add(x,y): return x+y

from functools import reduce
reduce(add, [1, 3, 5, 7])
# 16
```

The first argument to `reduce`, just as to `map` is always a function. However, the function must take two arguments. The function can be given as a defined function, as with `add` in the example, but it can also be given as a callable object or as a lambda expression. The following works fine.

```
from functools import reduce
reduce(lambda x,y: x+y, [1, 3, 5, 7])
# 16
```

The elements of lists don't have to be numbers. For example, they could be lists of numbers.

```
from functools import reduce
reduce(lambda x,y: x+y, [[0, 2, 4], [1, 3], [5, 7]])
# [0, 2, 4, 1, 3, 5, 7]
```

In this example, three lists have been concatenated into one list; that is, the original two-level list has been flattened into a long one-level list.

Reduction over a two-dimensional list, represented as a list of lists, can sometimes be accomplished by flattening the list. Assuming the `add` function has been defined as above, we have

```
data = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
from functools import reduce
reduce(add, reduce(add, data))
# 45
```

This example takes advantage of the fact that adding numbers and concatenating lists are both performed with the `+` operator, here wrapped up in the `add` function.

Another method is to combine `map` and `reduce`.

```
data = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
from functools import reduce
reduce(add, map(lambda lst: reduce(add, lst), data))
# 45
```

This approach has the advantage that we could easily use different operations within the columns (sublists) of the array from that used to combine columns. In the following, we multiply within sublists but add these products.

```
data = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
from functools import reduce
reduce(add, map(lambda lst: reduce(lambda x,y: x*y, lst), data))
# 630
```

Reduction works not only with lists but with any kind of sequences. Here is an example using a tuple.

```
from functools import reduce
reduce(lambda s,t: s+" "+t, ("Concatenation", "with", "spaces."))
# 'Concatenation with spaces.'
```

To summarize the use of reduction, `reduce` gives us a way to combine many parts using a function designed to handle two at a time.

11.5 Functional Composition

A common operation in functional programming is combining two functions to produce a new one. While this is done within imperative code so commonly that we don't raise a fuss about it, the creation of the new function usually happens at compile time rather than run time. To create the new function at run time, we need a method that combines functions as its run-time operation. The following two examples will illustrate (a) ordinary compile-time combination of functions and (b) run-time composition of functions.

```
def f(x): return 2*x
def g(x): return x+1
def h(x): return g(f(x))
h(10)
# 21

def compose(f1, f2):
  return lambda x: f2(f1(x))
```

```
h2 = compose(f,g)
h2(10)
# 21
```

Now let's illustrate functional composition with some of the other techniques of functional programming. First, let's create a class of callable objects that represent certain permutation functions. We'll call the class `make_transposer`. Its constructor will take an integer n and return a function that transposes the n-th element of a list with that to its right.

```
class make_transposer:
  def __init__(self, n):
    self.n = n

  def __call__(self, lst):
    new_list = lst[:]
    n = self.n
    new_list[n:n+2] = [lst[n+1],lst[n]]
    return new_list

transpose0 = make_transposer(0)
transpose2 = make_transposer(2)
transpose4 = make_transposer(4)

to_six = range(6)
print(transpose0(to_six), transpose2(to_six), transpose4(to_six))
# [1, 0, 2, 3, 4, 5] [0, 1, 3, 2, 4, 5] [0, 1, 2, 3, 5, 4]

transpose024 = reduce(compose, [transpose0, transpose2, transpose4])
print(transpose024(to_six))
# [1, 0, 3, 2, 5, 4]
```

In this example, the variable `transpose0` is bound to a callable object that interchanges the first two elements of a list. Thus elements 0 and 1 of the range of 6 get printed out in the order 1, 0 after this permutation has been applied. Similarly `transpose2` flips the elements in positions 2 and 3 (third and fourth elements) of the list; and `transpose4` flips the elements in positions 4 and 5. Since these callable objects are equivalent to functions, they can be used as arguments to `compose`. By using `reduce`, we can compose all three of these permutations to obtain a single permutation that transposes the first three pairs of elements at once.

11.6 Currying Functions With Arguments

Sometimes we need a function g of a single argument that acts as a given function f of two arguments, one of whose values is fixed. As a simple example,

consider the function `add` once again. It adds two numbers together. (It can also concatenate two strings together, but let's ignore this for now.) Suppose we want a function that always adds 5 to its argument. We could, of course, create such a function directly using the standard approach:

```
def add5(x): return x+5
```

However, as an alternative, we can derive an equivalent function from the more general `add` function using an operation called *currying*. The following definition for `curry` does the trick.

```
def curry(given_function, *fixed_args):
    def curried_function(*variable_args):
        return given_function(*(fixed_args+variable_args))
    return curried_function

add5 = curry(add, 5)

print(add5(12))
# 17
```

The definition of `curry` specifies that there be a first argument and some number of additional arguments called fixed arguments. The first argument to `curry` must be an existing function. This function should accept multiple arguments (well, at least one, to do any good). When `curry` is called, these fixed arguments will be bound to the formal parameter `fixed_args` as a list. The result returned by `curry` is a new function, temporarily referred to within the body of `curry` as `curried_function`. This function takes zero or more arguments. When the curried function is called, these arguments will be bound as a list to the formal parameter `variable_args`. The result that will be returned by the curried function is the result returned by the given function when passed a list of arguments consisting of all the fixed arguments followed by all the variable arguments.

With the `curry` function, a program can create new functions at run time by specializing an existing function. The specialization is accomplished by reusing the existing function and forcing certain arguments to be held constant and then not requiring the passing of values for those arguments.

11.7 Closures

Artificial intelligence programmers have traditionally used a programming technique known as closures to encapsulate certain persistent variable bindings (and consequently, state information) into functions. This was accomplished in languages such as Common Lisp and Scheme by the use of lambda expressions. In Python, lambda expressions do not capture a corresponding set of bindings, and thus, they will not work as expected by someone used to Common Lisp or Scheme.

However, the effect of using closures can be so closely simulated with the use of callable objects, that Python programmers do not generally feel a need for real closures. The following class of callable objects represent counters that count in multiples of some given value, n.

```
class counter_by_n:
    def __init__(self, n):
        self.n = n
        self.value = 0
    def __call__(self):
        self.value += self.n
        return self.value

c7 = counter_by_n(7)
print(c7(), c7(), c7())
# 7 14 21
c5 = counter_by_n(5)
print(c5(), c5(), c7())
# 5 10 28
```

Here one might feel constrained by the fact that there can be only one `__call__()` method per class, and so it might be impossible to have a cooperating pair of closures that operate on the same stored variable. The limitation is trivial, however, because there can be any number of other named methods, and these methods can easily be assigned as the values of variables. The following example illustrates this.

```
class account:
  def __init__(self):
    self.balance = 0.0
  def credit(self, amount):
    self.balance += amount
    return self.balance
  def debit(self, amount):
    self.balance -= amount
    return self.balance
  def methods(self):
    return [self.credit, self.debit]

smith_account = account()
deposit, withdraw = smith_account.methods()
deposit(30)
# 30.0
withdraw(4)
# 26.0
```

Here `deposit` and `withdraw` look like functions and are used like functions. However, they refer to methods of the class `account` and they are associated

with the particular instance, `smith_account`. Thus, they fill a very similar role to that of closures in functional languages. One difference is that closures in functional languages automatically capture the bindings of local variables that are within scope of the methods. In Python, the bindings to be captured must be explicitly coded using the `self` symbol with the dot notation.

Index