Chapter 21

Input and Output

“Data is not information, information is not knowledge, knowledge is not understanding, understanding is not wisdom.”
— Clifford Stoll

“It is relatively easy to design for the perfect cases, when everything goes right, or when all the information required is available in proper format.”
— Donald Norman

The field of computing grew out of applications characterized by input-output problems. Businesses tabulate sales and inventory figures, producing quarterly reports. Scientific applications process experimental data, generating statistical summaries, graphs and charts. Recent computing needs transcend simple reports: they require interactive, mobile and dynamic data sharing. That said, the basics of how most software works depends on some form of handling input data sets and creating results in desired, standard formats. Though new formats for output include video or 3D descriptions (there are now so-called 3D printers that manufacture objects from software designs), it’s still a good idea to learn simple character formats using ASCII, which is what this chapter offers.

The starting concepts for this chapter are input from keyboard, output to the console, and file i/o (reading and writing files that are on persistent storage, like disk or flash memory). Along the way, there are dependent concepts that turn out to be useful later for other purposes: Python has a formatting “mini-language” for precise control of how numbers and strings are placed; there are different options for conversion of input data from character form to internal data types. The plan of this chapter is to begin with the simple case of input from the keyboard and output to a console. Although this should be simple, this is an area where Python2 and Python3 significantly differ; the discussion has a number of examples showing language features for the two versions of Python. The remainder of the chapter thereafter deals with files.
Input and Raw Input

Python2 has two built-in functions that solicit text input from a user at the keyboard. These two functions, called `input()` and `raw_input()`, have in common that they suspend a running function or script indefinitely, waiting until a user types in some character data and presses Enter on the keyboard. Here is a simple demonstration of `input()`, done in an interactive Python2 session:

```python
>>> x = input("Type a number here --> ")
Type a number here --> 7
>>> x
7
```

What you cannot see above is the pause waiting for user input. The line “Type a number here --> 7” was partly made by Python2 and partly made by the user. Python2 printed the text Type a number here --> and put the cursor just after this text, waiting for the user to enter something. The string argument to `input()` is called the prompt, and it is displayed just before where you expect the user to type in some text. The prompt is optional; if you don’t supply a prompt, Python2 will wait for user input on a new, blank line. The prompt is a nice way to indicate that something is expected from the user (otherwise, Python2 would wait forever, or until the window is closed).

The `input()` function is usually called from a script or some function within a script. The script below looks simple, however the examples that run the script show there’s more to `input()` than one might first think.

```python
a = input("number please: ")
b = input("another one please: ")
print("you entered", a, b)
print("types of a,b are", type(a), type(b))
```

Let’s suppose this three-line script is in a file `demo.py` and then, at a Linux shell prompt:

```bash
> python demo.py
number please: 201.5
another one please: -40
you entered 201.5 -40
types of a,b are <class 'float'> <class 'int'>
>
```

This first example running `demo.py` is straightforward, and easy to understand. Let’s see another example with the same script:
Input and Output

> python demo.py
number please: 12*60
another one please: a/4 + 22
you entered 720 182
types of a,b are <class 'int'> <class 'int'>
>

The surprise here is that Python will accept an expression as the value that the user enters via keyboard, which can be any expression of the kind that might appear in normal Python statements. However, it has to be an expression: functions, methods, operators are allowed, but assignment statements, if, for, and so on are not allowed. Above, the input text “a/4 + 22” references variable a, which was assigned 720 by the previous statement. A somewhat strange example is:

> python demo.py
number please: [x**2 for x in range(5)]
another one please: "fabricate"[:4]
you entered [0, 1, 4, 9, 16] fabric

types of a,b are <class 'list'> <class 'str'>
>

Though this method of getting input from a user is quite flexible, it opens the door to many mistakes:

> python demo.py
number please: one
NameError: name 'one' is not defined
>

The input one caused the script to stop with an error, because Python could not evaluate the string “one” and get a value (if the input had been "one" then Python would at least get some value, a string). Worse even than input that causes a script to stop with an error could be input of an expression containing mutating method calls: when Python evaluates somelist.reverse() it changes what is in the variable somelist. Therefore some people consider the input() function to be a security problem, because it can allow the user to crash a program, trick it into giving bad results, and so on.

The raw_input() function does not suffer from such security problems: scripts do not crash and users cannot do more than supply characters for input to the script. Here is a rewriting of demo.py using raw_input():

```python
a = raw_input("number please: ")
b = raw_input("another one please: ")
print("you entered", a, b)
print("types of a,b are", type(a), type(b))
```

Copyright © 2010 Ted R. Herman, all rights reserved.
Input and Output

The example below purposefully has some input that would be illegal for the `input()` function:

```python
> python demo.py
number please: 1 plus 2
another one please: seven
you entered 1 plus 2 seven
types of a,b are <class 'str'> <class 'str'>
>
```

This example reveals why `raw_input()` doesn’t run into trouble: the result of calling `raw_input()` is always a string. Python won’t interpret this string as an expression, so there is no danger of causing the script to fail with an error.

Python3

In Python3, the `input()` was removed, and `raw_input()` renamed to `input()`; in other words, Python3’s `input()` function behaves the way Python2’s `raw_input()` does.

Validation

On the one hand Python2’s `raw_input()` (or Python3’s `input()`) does not fail, no matter what a user enters from the keyboard. On the other hand, getting a string may not be what is needed. Suppose we would like to make small Python2 script `demo.py` that gets two numbers from the keyboard:

```python
mass = raw_input("Enter mass: ")
velocity = raw_input("Enter velocity: ")
mass = int(mass)
velocity = int(velocity)
print("mv =", mass*velocity)
```

Here’s a sample run of the script:

```bash
> python demo.py
Enter mass: 10
Enter velocity: 30
mv = 300
>
```

So far, so good. But what if the user makes a mistake?
For conversion from a string to an integer, Python requires that the string contain only numeric digits whereas the keyboard input contained a period (Python is more actually more flexible: the result of \texttt{int("-0095")} is 95, for instance). It may seem therefore that \texttt{raw_input()} is no better than Python2’s \texttt{input()} for dealing with user mistakes. However there are other steps one can add to the demo.py to validate the user input before attempting conversion:

```
def valid(numstring):
    if len(numstring)<1:
        return False
    for char in numstring:
        if char not in "0123456789":
            return False
    return True

def getinput(varname):
    while True:
        R = raw_input("Enter " + varname + ": ")
        if valid(R):
            return R
        print("Please retry - input must be digits only")

mass = int( getinput("mass") )
velocity = int( getinput("velocity") )
print("mv =", mass*velocity)
```

The \texttt{valid()} function will only return \texttt{True} if the argument is a string that \texttt{int()} would convert without error. The \texttt{getinput()} function will ask the user to retry, and retry again (with no limit on retries) until what the user keys in is valid – which is why this way of handling input is called \textit{input validation}.

It’s relatively easy to validate a string that should consist only of numeric digits. But many input formats are considerably more complex: dates, times, and scientific numbers with exponents are not easy to validate. In a later chapter, we will revisit the topic of input validation and see a more powerful technique that Python offers for this purpose.
Output Formatting

Python’s `print()` function (Python3) or the `print` statement (Python2) convert various Python types such as strings, lists, integers, and dictionaries to some standard format for display. Whereas strings need no conversion (`print` only has to interpret control sequences like \n for proper output), all the other types first need conversion to strings. For instance,

```python
>>> print( [1,2,3] )
[1, 2, 3]
```

shows that `print` first converted the list `[1,2,3]` into a string, and then displayed that string. We can also see the same conversion interactively:

```python
>>> str( [1,2,3] )
' [1, 2, 3]'
>>> repr( [1,2,3] )
' [1, 2, 3]'
```

The built-in `repr()` function is the preferred way to convert a type into a string, though `str()` works as well. One crucial advantage of `repr()` is seen in the interactive experiments with a string:

```python
>>> str("abc")
'abc'
>>> repr("abc")
"'abc'"
```

For a string, `repr()` creates a string that, when printed, would show how Python encodes a string (don’t worry if this makes little sense right now, since our goal for `repr()` is mainly to convert lists, dictionaries, and other more complex types). What if `repr`’s conversion of a list into a string isn’t exactly what is desired? Here’s a simple way to get a conversion that does not introduce blanks:

```python
>>> repr( [1,2,3] ).replace( ' ',',')
'[1,2,3]'
```

The trick of using string methods, or even functions you may write, lets you take the string that `repr` produces and change it as you like.

Templates

The idea of a text `template` is familiar to anyone who has received form letters.
Dear __________,

Good news! We’ve added _______ value points to your account. And if you purchase an upgrade to your current package, ...

A program fills in the blanks before sending the form letter, putting a customer name in the first blank and an amount in the second blank. Python3 and later versions of Python2 support a built-in string method format() that implements the idea of a template. A simple example of this, shown as an interactive session, is:

```python
>>> "starting time {0}, ending time {1}".format(1200, 1930)
'starting time 1200, ending time 1930'
>>> "W{0}{1}t {0} wh{1}l{2}".format('a', 'i', 'e."
'Wait a while.'
```

The format method’s template contains special areas such as {0} that will be replaced by arguments to the method. We call the special areas {0}, {1}, ..., pattern fields in the template. The first argument (1200 and ‘a’ above) replaces {0} wherever this pattern field occurs, the second argument replaces {1}, and so on. The template, or pattern string, is shown as a string above, but typically would be a variable of type str; such a variable could be thousands of characters, representing many lines of text.

The format() goes far beyond replacing pattern fields. The format() method substitutes for the text in the pattern field, guided by information from the arguments and from instructions in the special areas themselves. A full enumeration of format’s power is beyond this chapter, though the main features are worth mention:

- **format()** converts its arguments to strings during substitution.
- **format()** can either use positional arguments (where {0} refers to the first argument, {1} the second argument, and so on), or format() may use a dictionary as its argument, in which case substitution is driven by the names of keys in the dictionary (which are given within the pattern fields).
- When numbers are converted to strings, the pattern field can say whether the number should be left-justified, right-justified, printed in decimal, hexadecimal, using exponent notation, the number of significant digits, and other formatting details.

Taking full advantage of format() is not simple. Indeed, the official Python documentation has a section entitled “Format Specification Mini-Language”, indicating that format() actually uses a specialized data language describing how arguments substitute for text in a string. The following table shows just a few examples and what they intend, for pattern fields.

---

Copyright © 2010 Ted R. Herman, all rights reserved.
To understand this small table, it’s helpful to show some examples. The first example shows that the argument index can be omitted, and Python will simply substitute pattern fields in the order they occur:

```python
>>> "{:12}aaaaaa{:e}".format('hi',2345.6789)
'hi aaaaaa2.345679e+03'
>>> "{1:-^10.2f} and {0:,} ok".format(1.5,235111982)
'---1.50--- and 235,111,982 ok'
```

Experienced Python programmers do not typically memorize the full formatting mini-language (what can be put into pattern fields); rather, they learn to locate the documentation on the language themselves, then experiment perhaps interactively to be sure that what they have put into the pattern fields gets the right results.

### Python2 Substitution

Before `format()` was introduced into the Python language, there was another template operator. This older operator is still supported in Python2, though it is not available in Python3. We mention it here in case you happen to look at older scripts or have need to work with an older version of Python2, such as version 2.4. Recall that Python has a remainder operator, "%" often used to test whether an integer is even or odd. The remainder operator is only defined for integers. Taking advantage of this known limitation, Python used % for another purpose, with strings. This examples shows how the operator handled substitution in templates:

```python
>>> "M is %d and V is %5.2f in the equation" % (100,12.5)
'M is 100 and V is 12.50 in the equation'
```

For the older-style % operator, pattern fields were identified by the % character, and a mini-language of formatting controlled some details of number conversion to strings.
Reading Files

The functions, scripts and examples of this section suppose that `ex.txt` is a file containing these five lines:

```
A flea and a fly in a flue
Were imprisoned, so what could they do?
Said the fly, "let us flee!"
"Let us fly!" said the flea.
So they flew through a flaw in the flue.
```

The file `ex.txt` would be located in some directory, which determines its “full address” for technical purposes. For instance, on a Linux system, it might be `/home/user/Desktop/ex.txt` whereas on a Windows system it could be `C:\Users\Robert\Desktop\ex.txt`. When you see the file through a windowing system with folders and icons, the file might just appear with the name `ex`, because the window system may hide the “.txt” suffix: window systems are designed for users, not for software developers. Whatever the situation, you will probably need to know the full address and work with directories (folders), possibly changing the “working directory” (using the `cd` command under Windows or Linux) to use files with Python.

We start with some interactive exercises that read from a file. The first concept to know about is a file object, which has the type `file` in Python. For reading files, there are three basic operations, `open`, `close`, and `read` (there are a few others as well, for more advanced work). Here is a script showing the creation of a file object:

```python
>>> F = open("ex.txt", 'r')
>>> G = open("ex.txt")
>>> print(type(F), type(G))
<class 'file'> <class 'file'>
```

The example creates two file objects; each of these file objects does the same as the other, it serves as a basis for reading the file `ex.txt` – this only works without the full address because this interactive exercise was done in the same directory as where `ex.txt` resides. There’s no reason to create two file objects to read a file. The exercise just demonstrates four different ways to do the same thing. The argument `r` in two cases above tells Python that the file should be prepared for reading (not writing). If left out, this argument defaults to `r` anyway.

**Note for Python2**: another built-in called `file()` can be used as well as `open()`, but `file()` is not valid in Python3.
Suppose `F` has the file object for `ex.txt`, as created above. Now, to read the file,

```python
>>> text = F.read()
>>> len(text)
166
>>> text.count('\n')
5
```

The `read()` method makes a string from the contents of the file; you can see that the string has 166 characters and 5 newline characters (one at the end of each line). Now, suppose we need to process the text in the file. The following script puts the ideas above together with a loop to print the three most frequently occurring words in the file.

```python
F = open("ex.txt")
text = F.read()
wordlist = text.split()
countable = { } # will be dictionary words & counts
for word in wordlist:
    if word not in countable:
        countable[word] = 0
    countable[word] += 1
# next, make a list of (count,word) tuples
freqlist = []
for word in countable:
    new = (countable[word],word)
    freqlist.append(new)
# sort the list by increasing order of count values
freqlist.sort()
# print highest three instances
print(freqlist[-1],freqlist[-2],freqlist[-3])
```

When the script runs, it prints

```
('the', 3) ('a', 3) ('us', 2)
```

You may notice some deficiencies in this script. One could argue that the count for 'a' should be 4, not 3; but the script counted A and a as different words. Also, the script included punctuation marks in the words. A more careful version of this script would overcome these deficiencies.

**Note for Python3:** the `read()` function may not work for files that contain non-text characters. Suppose `myfile.dat` contains some data in binary (not text). The way to read from such a file is inform Python, when the file is being opened, to treat the file as a binary file. Here's an example:
F = open("myfile.dat","rb") # "b" means binary
data = F.read()

As a result of these two statements, variable data contains the file, however the type of data is a new (Python3) type called bytes. It’s beyond this chapter to fully explain the bytes type, which is much like a string, but for binary values. The conversion between string and bytes types is by methods,

A.encode() for str A, returns a bytes copy
B.decode() for bytes B, returns a str copy

Reading Lines

Quite often, the data in text files is organized into lines, which are separated by \n, or in the case of Windows, by \r\n strings. For many applications, it is natural to process the input line by line. Here’s a script to make print a list of the line lengths in ex.txt:

F = open("ex.txt")
text = F.read()
linelist = text.split('\n')
lengths = []
for line in linelist:
    lengths.append(len(line))
print(lengths)

The output from this script may surprise you:

[26, 39, 28, 28, 40, 0]

(To understand why there’s a 0 at the end of the list, you may need to review how the split method works in Chapter 11.) This way of processing lines, by reading in the whole file to a string, then using split, is considered poor style for two reasons:

1. The split('\n') technique alone may not be what you want, as the example above indicates; also, it will leave in the \r characters if the text file has Windows file encoding, but not have these characters under Linux systems. This makes life more complicated than it should be.

2. Reading in the entire file to a string may not be practical for very long files. It would be better to read such a file a bit at a time. Fortunately, Python has several ways to do this.
Rather than use `read()`, there is another, somewhat "intuitive" way to read the lines of a file:

```python
F = open("ex.txt")
lengths = []
for line in F:
    lengths.append(len(line))
print(lengths)
```

The output from this is:

```
[27, 40, 29, 29, 41]
```

Notice here that an ordinary for loop can use a file object as though it were a sequence; doing so, the `loop_iter` variable takes on a new string, the contents of a line in the file, in each iteration. Also, in each iteration, the `line` variable contains the full line, including the ending `\n` character (which explains why the lengths of the lines are one larger than in the earlier, similar example).

One cautionary note on using a `for` loop to process the lines in a file: unlike `for` loops over ordinary sequences, a `for` iteration over lines of a file only works once:

```python
F = open("ex.txt")
firstlinecount = 0
for line in F:
    firstlinecount += 1
secondlinecount = 0
for line in F:    # attempt second time
    secondlinecount += 1
print([firstlinecount,secondlinecount])
```

This script prints `[5, 0]`. The file object behave as though a single iteration over its lines “deplete it”, so that it has no more lines for another loop. However, a simple change makes a difference:

```python
F = open("ex.txt")
firstlinecount = 0
for line in F:
    firstlinecount += 1
F.close()
F = open("ex.txt")
secondlinecount = 0
for line in F:    # attempt second time
    secondlinecount += 1
print([firstlinecount,secondlinecount])
```
Now it prints [5, 5]. The close() method tells Python that a file object is finished, and will not be read further. However, the line following creates a new file object and assigns it to F (it happens to be for the same input file), so the loop following works as expected.

Reading Bytes

It makes sense to read lines of a file that is text. The same does not hold for many other kinds of file, including mp3 files, image files (gif, jpg, etc), or raw data from scientific instruments. These are considered binary files, as opposed to text files. Python can read binary files into a string, using the read() method, but again this is impractical for large files. The technique for reading binary files is to read them in chunks, a bit at a time, processing the input as bytes of data. The way to do this is to supply an argument for the read method, which tells Python how many bytes from the file to read. As an illustration, the following script treats ex.txt as though it were a binary file, counting the number of characters equal to s:

```python
F = open("ex.txt")
scount = 0
while True:
    char = F.read(1)
    if len(char)<1:
        break
    if char=="s":
        scount += 1
print(scount)
```

This script sets up what looks like an infinite loop; the rationale for this is that we may not know, at the outset, how many bytes the file contains. Each assignment `char = F.read(1)` is essentially a request, to Python, to read the next byte from the file and put that into variable `char`. However, at the end of the file, this request will fail: as a result, `char` will be an empty string at the end. That’s why the following if statement checks the length of `char`, to see whether the end of the file has been reached — this terminates the while loop. The choice of using `read(1)` was motivated by the goal, looking at one character at a time in a loop. Other applications might read four bytes in each iteration, in which case the method would be `read(4)` to request the next four bytes.

There is much more to reading and interpreting the bytes of a binary file than the example above would suggest. Typically, another module `struct` is used to convert bytes into integers, floating point numbers, etc. You can read about `struct` online if you encounter the need to process binary files in some application.
Standard Input

Another criticism of examples above is that they “hard code” the file name. That is, the name of the file ex.txt is written into the Python code. What you would like to use the same script one time for ex.txt, another time for june.txt, and so forth? It would be nice to do this without having to change the Python code. Linux has a convention to allow just this capability, called redirection.

stdin

Long ago, in the early days of operating systems, most programs were simple: they would read input, typically from punched cards, magnetic tape, or paper tape, and print output results on paper. The input device, whether a card reader or a tape reader, had a symbolic name like STDIN or SYSIN. Later, as computing media progressed from cards and tape to other forms, this notion persisted. Programs were supplied, automatically, with an input file named stdin or something similar. Python can use this feature, shown here in a script:

```python
import sys
linecount = 0
for line in sys.stdin:
    linecount += 1
print(linecount)
```

The sys module has many functions and variables related to the operating system support of Python, including the stdin file. Suppose this script is put into linecount.py, and we would like to count the lines in ex.txt. From a command prompt, it is run like this:

```
> python linecount.py < ex.txt
5
```

The part of the command specifying the input file, that is, the part with tells which file will be used for sys.stdin, is the “< ex.txt”. This is a convention of the operating system (shell language of system commands), to redirect where a program will look for stdin input. Note: if this is omitted, and if a program tries to read from stdin, then the input will be expected from the user at the keyboard. As an example, consider this:

```
> python linecount.py
one
two
```
stop
exit
quit
help, get me out!

Actually, this goes on and on with more lines, because the user has no way to
tell the program that stdin has ended via the keyboard device. The moral of
the story is that it is better to use input() and validation if keyboard input is
appropriate; otherwise, use redirection to associate sys.stdin with some file.
The following interaction at a Linux shell prompt illustrates why redirection is
handy:

> python linecount.py < monday.txt
3128
> python linecount.py < tuesday.txt
405
> python linecount.py < tuesday.txt
11218

The linecount.py script was invoked three times on three different files, with-
out having to make any change to the script.

Multiple Files

There is no limit in Python to the number of files that a program can use. An
example below shows a loop through two files. One thing to keep in mind is
that there is some memory overhead for each file. Large programs that read
many files may benefit from reducing the overhead. The memory overhead for
such large programs is not the total number of files they read, but rather the
number of open files at any time during the run. To reduce this overhead, it is
wise to close each file when it is no longer needed, for instance

```python
F = open("ex.txt","r")
for line in F:
    myprocess(line)
F.close() # done with file F
...
```

Some function myprocess() looks at each line of the file; after the loop termi-
nates, it’s a good idea to close the file.

readline

Consider the task of merging two files ex1.txt and ex2.txt. There could be
many ways to merge two files, including a back-to-back merge, a line after line
merge, and so on. As an example, the following program takes a pair of lines, one from each file, and prints the concatenation of these two lines, in a loop. The example introduces another method for file objects, the `readline` method. Each use of `readline` returns the next line in the file. At the end of the file, `readline` returns the empty string.

```python
F, G = open("ex1.txt"), open("ex2.txt")
while True:
    line1 = F.readline()
    line2 = G.readline()
    if line1=='' and line2=='':
        break  # end of both files
    outputline = line1 + line2
    print(outputline)
F.close()
G.close()
```

When the script runs, it may be that one of the two input files has fewer lines than the other. In such an event, the `while` loop will get empty strings from `readline()` of the shorter file as it continues to read lines from the longer file. Python also has a `readlines()` method: it returns a list of all the lines in a file (unlike `read()`, which returns the entire file contents as a string).

### Writing Files

Python offers a `write()` method that adds strings to a file. To use `write()`, the file object needs to be prepared for output. These two statements are equivalent ways of creating a file object for output.

```python
F = open("out.txt","w")
G = open("out.txt","w")
```

The second argument to either `file` or `open` tells Python whether the file object will be used for reading or for writing (and if the second argument is omitted, the default is reading).

The `write()` method is deceptively simple to use: `F.write("hello")` adds the string `hello` to the file represented by file object `F`. The underlying behavior of Python, and the operating system, add some wrinkles to this simple interpretation. Reading and writing differ significantly in timing of data transfer. When a program has a statement

```python
R = F.readline()
```
the assignment of a string to variable \texttt{R} actually takes some amount of time
to perform (under a millisecond). Though this amount of time is not humanly
perceptible, it does cause Python to wait until the data is transferred from the
file to \texttt{R}. In contrast,

```python
T = "hello"
F.write(T)
print("done")
```

may not transfer \texttt{hello} to the file before printing \texttt{done}.

As far as using methods \texttt{read}, \texttt{readline}, or \texttt{readlines}, the only thing we
need to know is that these methods fetch some data when the method is called.
However, for the \texttt{write} method, the situation is not so simple. For reasons of
efficiency, the \texttt{write} method \texttt{does not instantly transfer data}. Instead, the \texttt{write}
method "schedules" the actual transfer of data to the file at some future time.
There are several motivations for scheduling future data transfer. It can be that
the file media (flash, disc, \texttt{etc}) is rather slow compared to program steps, so it
would be less efficient to make the program slow down for each \texttt{write} call; it
can be that the file media uses less power (energy) when multiple file transfers
are “bundled” together into larger size blocks of data.

### Flushing Buffers

Two file object methods can be used to \texttt{force} the transfer of data that previous
\texttt{write} calls have scheduled: they are \texttt{flush()} and \texttt{close()}. The \texttt{flush()}
method can be called at any time after a file object has been prepared for output,
up to the point where the \texttt{close()} method has been called. The \texttt{close()} method
not only finishes and scheduled data transfer, but it also releases any
memory resources entailed by the file object. After the \texttt{close()} method has
been called, the file object is no longer available for \texttt{write} operations.

Why does this matter? Suppose a program generates a large output, writing
to a file that will be many gigabytes when the program finishes. Now imagine
that the program does not finish, because of a power outage that crashes the
computer. What will be in the file after the power goes out? Hard to say.
However, if the program occasionally uses the \texttt{flush()} method, then at least
some of the scheduled data transfer will likely be added to the output file.
Such a program could run for hours or days before it closes the output file,
and using \texttt{flush()} is therefore sensible. In the small exercises of this chapter,
using \texttt{flush()} won’t be justified. Just remember that a program writing to a
file should use \texttt{close()} before the program finishes. In fact, when a Python
program does finish, the operating system automatically closes any open files.
Yet the habit of using \texttt{close()} is valuable. Consider a Python program that
acts as a web server, responding to network requests and recording something
related to those requests in files. This program might run for weeks or months,
handling millions of network requests. For this program, it would be essential to use \texttt{close()} to finish scheduled data transfer, so that the output files have all the data they should (so that these files could be used for other needs, including analysis, report generation, and so on).

\textbf{Please use close()} when finished writing.

\section*{String Preparation}

The argument of the \texttt{write()} method must be a string. Here is a demonstration of this fact:

\begin{verbatim}
>>> F = open("out.txt",'w')
>>> F.write(1.5)
  TypeError: must be string, not float
\end{verbatim}

If only a string can be written, but one would like to write numbers, then some kind of conversion to a string is needed.

\begin{verbatim}
F = open("out.txt","w")
for x in range(20):
    s = repr(x)
    F.write(s)
F.close()
\end{verbatim}

The script above writes integers converted to strings, using the built-in \texttt{repr()} function described earlier in this chapter. After running this script, the file \texttt{out.txt} consists of the single line:

\begin{verbatim}
012345678910111213141516171819
\end{verbatim}

This result may not be what one would hope for. Unlike \texttt{print}, the \texttt{write()} method doesn’t automatically insert spaces or new line (\texttt{\textbackslash n}) characters. That’s where formatting is handy.

\begin{verbatim}
F = open("out.txt","w")
for x in range(10,16):
    s = "{}\n".format(x)
    F.write(s)
F.close()
\end{verbatim}

The file \texttt{out.txt} contains these lines after the script runs:

\begin{verbatim}
012345678910111213141516171819
\end{verbatim}
Reading and Writing

Python has additional methods for file objects so that the same file can be both read from and written to, in the same program. Databases consist of files that are updated, meaning that data is read from the file, changed in memory, and then written back over the original values. Python is capable of doing this, but that’s an advanced topic beyond this chapter. Another feature worth mentioning is that, on most operating systems where Python runs, an output file can be opened in append mode: when a file object is created with append mode, the data written to the file does not replace what the file already may have — new data written to the file is concatenated to what’s already in the file. Append mode is frequently used for logging events, audit trails, or other applications where many scripts write output to the same file, and the result is a file with the output of all of them.

Printing to Files

The print statement in Python2, or the print function in Python3, is implemented internally within Python using the write() method. By default, Python directs printed output to a file object sys.stdout. Usually, the operating system sends anything intended for standard output (sys.stdout) to the console, as seen during interactive Python sessions. As with the standard input sys.stdin, it’s possible to redirect where standard output goes. Consider the basic script:

```python
for value in range(3,8):
    print(value)
```

Suppose this two-line script is in a file basic.py, and the following command is run at a Linux shell prompt:

```
> python basic.py > out.txt
>```
**Input and Output**

The ">" at the beginning of the two lines above is a marker for the command prompt, whereas the ">" before `out.txt` tells Linux to redirect standard output from the console to a file named `out.txt`. That's why you see no output from the script on the console. However, if you use some editor to look at `out.txt` after the script has run, the contents will be seen:

3
4
5
6
7

Both input redirection and output redirection can be used on the same script run.

```bash
> python countlines.py < myfile.txt > count.txt
```

The example runs the `countlines.py` script, taking standard input from the file `myfile.txt`, and sending anything printed to a file `count.txt`.

**Python2 print to File Object**

Python2 has a strange syntax for printing to a file object other than standard output (you can use the `print` statement to place output into any file that has been created for output). Here is an example printing to a file `newout.txt`.

```python
F = open("newout.txt","w")
for W in range(50):
    print >> F, W, W**2, W**3
F.close()
```

Unlike the `write()` method, the `print` statement will put spaces between the items printed and add a newline character at the end of the line.

**Python3 print to File Object**

For Python3, there is no `print` statement, there is the `print()` function. It has a keyword parameter `file` that defaults to `sys.stdout`. For any `print()` function call, the `file` argument can specify a file object for output.

```python
F = open("newout.txt","w")
for W in range(50):
    print(W, W**2, W**3, file=F)
F.close()
```
The `print()` function will put spaces between the items and add a newline (\n). The `print()` function has other keyword parameters that can change some details of behavior, including the spacing between items printed and whether the newline character is inserted at the end.

**Terminology Review**

Python2 has two techniques for getting input from the keyboard, `input()` and `raw_input()`, whereas Python3 has only `input()`. The `format` method and the formatting mini-language use a template to control how output strings convert other types into readable form. This chapter introduced `file objects` and methods `read()`, `write()`, `readline()`, `close()`, plus functions to create file objects, `open()` and `file()`. Standard input, standard output, and redirection (for both input and output) have names and special notation (`sys.stdin`, `sys.stdout`, with “<” and “>” in a Linux command console). A new kind of `for` loop was introduced in this chapter, which is used to iterate over the lines of a file. The `repr()` built-in function was introduced, and the Python3 `bytes` type was mentioned, for which there are methods `decode()` and `encode()`.

**Exercises**

1. Write a script that takes a text file, perhaps containing a news article or some short story, and makes a version of the same file in which all punctuation characters (.,;:’"?–) have been removed, as well as numeric digits (0123456789) and other non-alphabetic characters ($%^&*()[]@!$, etc). Furthermore, the script should convert uppercase to lowercase. Thus, The output file should only contain whitespace characters and letters (a–z), but have the same number of lines as the input file and the same number of “words”.

2. Write a script that produces a table with two columns, words and counts. The input for the script is a text file containing simple words, like the output file from problem (1). For each word in the input file, there should be a row in the table with that word and the number of times it occurs in the input file, so there should be as many rows in the table as there are different (distinct) words in the input file. The table’s two columns should line up, so that all words start at the beginning of the line, and all numbers line up in the same column.