Python for Everybody

Exploring Data Using Python 3

Charles R. Severance

```
>>> word = 'banana'
>>> index = word.find('a')
>>> print(index)
1
```

In this example, we invoke `find` on `word` and pass the letter we are looking for as a parameter.

The `find` method can find substrings as well as characters:

```
>>> word.find('na')
2
```

It can take as a second argument the index where it should start:

```
>>> word.find('na', 3)
4
```

One common task is to remove white space (spaces, tabs, or newlines) from the beginning and end of a string using the `strip` method:

```
>>> line = '  Here we go  '
>>> line.strip()
'Here we go'
```

Some methods such as *startswith* return boolean values.

```
>>> line = 'Have a nice day'
>>> line.startswith('Have')
True
>>> line.startswith('h')
False
```

You will note that `startswith` requires case to match, so sometimes we take a line and map it all to lowercase before we do any checking using the `lower` method.

```
>>> line = 'Have a nice day'
>>> line.startswith('h')
False
>>> line.lower()
'have a nice day'
>>> line.lower().startswith('h')
True
```

In the last example, the method `lower` is called and then we use `startswith` to see if the resulting lowercase string starts with the letter "h". As long as we are careful with the order, we can make multiple method calls in a single expression.

Exercise 4:

There is a string method called `count` that is similar to the function in the previous exercise. Read the documentation of this method at https://docs.python.org/3.5/library/stdtypes.html#string-methods and write an invocation that counts the number of times the letter a occurs in "banana".

6.10 Parsing strings

Often, we want to look into a string and find a substring. For example if we were presented a series of lines formatted as follows:

`From stephen.marquard@ uct.ac.za Sat Jan  5 09:14:16 2008`

and we wanted to pull out only the second half of the address (i.e., `uct.ac.za`) from each line, we can do this by using the `find` method and string slicing.

First, we will find the position of the at-sign in the string. Then we will find the position of the first space *after* the at-sign. And then we will use string slicing to extract the portion of the string which we are looking for.

```
>>> data = 'From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008'
>>> atpos = data.find('@')
>>> print(atpos)
21
>>> sppos = data.find(' ',atpos)
>>> print(sppos)
31
>>> host = data[atpos+1:sppos]
>>> print(host)
uct.ac.za
>>>
```

We use a version of the `find` method which allows us to specify a position in the string where we want `find` to start looking. When we slice, we extract the characters from "one beyond the at-sign through up to *but not including* the space character".

The documentation for the `find` method is available at

https://docs.python.org/3.5/library/stdtypes.html#string-methods.

6.11 Format operator

The *format operator*, `%` allows us to construct strings, replacing parts of the strings with the data stored in variables. When applied to integers, `%` is the modulus operator. But when the first operand is a string, `%` is the format operator.

The first operand is the *format string*, which contains one or more *format sequences* that specify how the second operand is formatted. The result is a string.

For example, the format sequence "%d" means that the second operand should be formatted as an integer (`d` stands for "decimal"):

```
>>> camels = 42
>>> '%d' % camels
'42'
```

The result is the string "42", which is not to be confused with the integer value 42.

A format sequence can appear anywhere in the string, so you can embed a value in a sentence:

```
>>> camels = 42
>>> 'I have spotted %d camels.' % camels
'I have spotted 42 camels.'
```

If there is more than one format sequence in the string, the second argument has to be a tuple[1]. Each format sequence is matched with an element of the tuple, in order.

The following example uses "%d" to format an integer, "%g" to format a floating-point number (don't ask why), and "%s" to format a string:

```
>>> 'In %d years I have spotted %g %s.' % (3, 0.1, 'camels')
'In 3 years I have spotted 0.1 camels.'
```

The number of elements in the tuple must match the number of format sequences in the string. The types of the elements also must match the format sequences:

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'dollars'
TypeError: %d format: a number is required, not str
```

In the first example, there aren't enough elements; in the second, the element is the wrong type.

The format operator is powerful, but it can be difficult to use. You can read more about it at

https://docs.python.org/3.5/library/stdtypes.html#printf-style-string-formatting.

6.12 Debugging

A skill that you should cultivate as you program is always asking yourself, "What could go wrong here?" or alternatively, "What crazy thing might our user do to crash our (seemingly) perfect program?"

For example, look at the program which we used to demonstrate the `while` loop in the chapter on iteration:

```
while True:
    line = input('> ')
```

[1] A tuple is a sequence of comma-separated values inside a pair of brackets. We will cover tuples in Chapter 10

```
    if line[0] == '#':
        continue
    if line == 'done':
        break
    print(line)
print('Done!')

# Code: http://www.pythonlearn.com/code3/copytildone2.py
```

Look what happens when the user enters an empty line of input:

```
> hello there
hello there
> # don't print this
> print this!
print this!
>
Traceback (most recent call last):
  File "copytildone.py", line 3, in <module>
    if line[0] == '#':
IndexError: string index out of range
```

The code works fine until it is presented an empty line. Then there is no zero-th character, so we get a traceback. There are two solutions to this to make line three "safe" even if the line is empty.

One possibility is to simply use the `startswith` method which returns `False` if the string is empty.

```
    if line.startswith('#'):
```

Another way is to safely write the `if` statement using the *guardian* pattern and make sure the second logical expression is evaluated only where there is at least one character in the string.:

```
    if len(line) > 0 and line[0] == '#':
```

6.13 Glossary

counter A variable used to count something, usually initialized to zero and then incremented.

empty string A string with no characters and length 0, represented by two quotation marks.

format operator An operator, %, that takes a format string and a tuple and generates a string that includes the elements of the tuple formatted as specified by the format string.

format sequence A sequence of characters in a format string, like `%d`, that specifies how a value should be formatted.

format string A string, used with the format operator, that contains format sequences.

flag A boolean variable used to indicate whether a condition is true or false.

invocation A statement that calls a method.

immutable The property of a sequence whose items cannot be assigned.

index An integer value used to select an item in a sequence, such as a character in a string.

item One of the values in a sequence.

method A function that is associated with an object and called using dot notation.

object Something a variable can refer to. For now, you can use "object" and "value" interchangeably.

search A pattern of traversal that stops when it finds what it is looking for.

sequence An ordered set; that is, a set of values where each value is identified by an integer index.

slice A part of a string specified by a range of indices.

traverse To iterate through the items in a sequence, performing a similar operation on each.

6.14 Exercises

Exercise 5: Take the following Python code that stores a string:‘

```
str = 'X-DSPAM-Confidence:0.8475'
```

Use `find` and string slicing to extract the portion of the string after the colon character and then use the `float` function to convert the extracted string into a floating point number.

Exercise 6:

Read the documentation of the string methods at

https://docs.python.org/3.5/library/stdtypes.html#string-methods

You might want to experiment with some of them to make sure you understand how they work. `strip` and `replace` are particularly useful.

The documentation uses a syntax that might be confusing. For example, in `find(sub[, start[, end]])`, the brackets indicate optional arguments. So `sub` is required, but `start` is optional, and if you include `start`, then `end` is optional.

Chapter 7

Files

7.1 Persistence

So far, we have learned how to write programs and communicate our intentions to the *Central Processing Unit* using conditional execution, functions, and iterations. We have learned how to create and use data structures in the *Main Memory*. The CPU and memory are where our software works and runs. It is where all of the "thinking" happens.

But if you recall from our hardware architecture discussions, once the power is turned off, anything stored in either the CPU or main memory is erased. So up to now, our programs have just been transient fun exercises to learn Python.

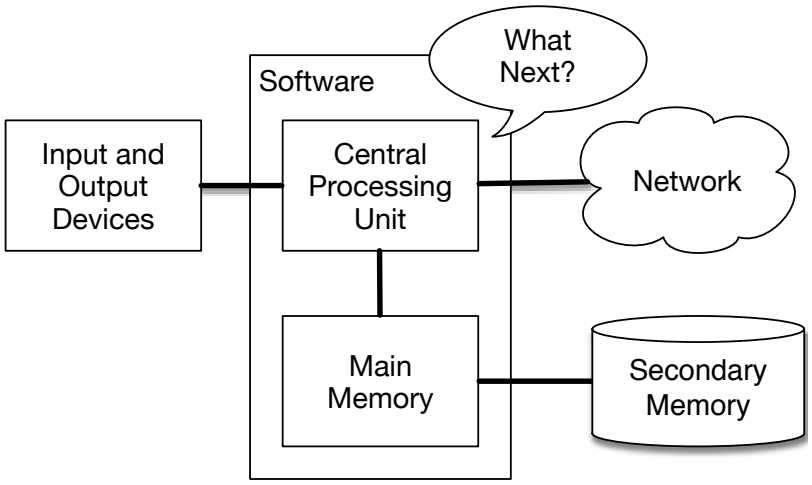

Figure 7.1: Secondary Memory

In this chapter, we start to work with *Secondary Memory* (or files). Secondary memory is not erased when the power is turned off. Or in the case of a USB flash drive, the data we write from our programs can be removed from the system and transported to another system.

We will primarily focus on reading and writing text files such as those we create in a text editor. Later we will see how to work with database files which are binary files, specifically designed to be read and written through database software.

7.2 Opening files

When we want to read or write a file (say on your hard drive), we first must *open* the file. Opening the file communicates with your operating system, which knows where the data for each file is stored. When you open a file, you are asking the operating system to find the file by name and make sure the file exists. In this example, we open the file `mbox.txt`, which should be stored in the same folder that you are in when you start Python. You can download this file from www.pythonlearn.com/code3/mbox.txt

```
>>> fhand = open('mbox.txt')
>>> print(fhand)
<_io.TextIOWrapper name='mbox.txt' mode='r' encoding='cp1252'>
```

If the `open` is successful, the operating system returns us a *file handle*. The file handle is not the actual data contained in the file, but instead it is a "handle" that we can use to read the data. You are given a handle if the requested file exists and you have the proper permissions to read the file.

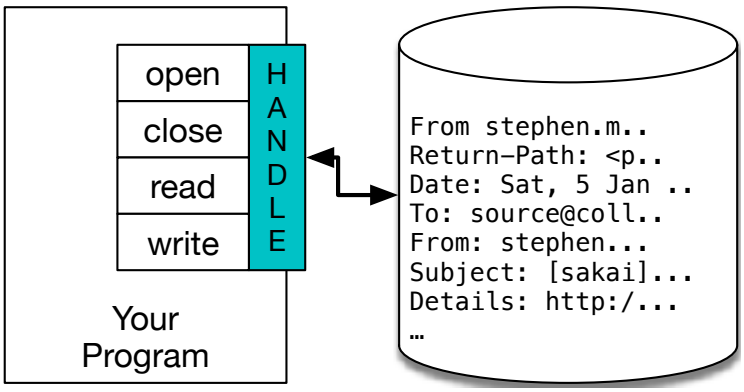

Figure 7.2: A File Handle

If the file does not exist, `open` will fail with a traceback and you will not get a handle to access the contents of the file:

```
>>> fhand = open('stuff.txt')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'stuff.txt'
```

Later we will use `try` and `except` to deal more gracefully with the situation where we attempt to open a file that does not exist.

7.3 Text files and lines

A text file can be thought of as a sequence of lines, much like a Python string can be thought of as a sequence of characters. For example, this is a sample of a text file which records mail activity from various individuals in an open source project development team:

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
Return-Path:  <postmaster@collab.sakaiproject.org>
Date:  Sat, 5 Jan 2008 09:12:18 -0500
To:  source@collab.sakaiproject.org
From:  stephen.marquard@uct.ac.za
Subject:  [sakai] svn commit:  r39772 - content/branches/
Details:  http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
...
```

The entire file of mail interactions is available from

www.pythonlearn.com/code3/mbox.txt

and a shortened version of the file is available from

www.pythonlearn.com/code3/mbox-short.txt

These files are in a standard format for a file containing multiple mail messages. The lines which start with "From" separate the messages and the lines which start with "From:" are part of the messages. For more information about the mbox format, see en.wikipedia.org/wiki/Mbox.

To break the file into lines, there is a special character that represents the "end of the line" called the *newline* character.

In Python, we represent the *newline* character as a backslash-n in string constants. Even though this looks like two characters, it is actually a single character. When we look at the variable by entering "stuff" in the interpreter, it shows us the `\n` in the string, but when we use `print` to show the string, we see the string broken into two lines by the newline character.

```
>>> stuff = 'Hello\nWorld!'
>>> stuff
'Hello\nWorld!'
>>> print(stuff)
Hello
World!
>>> stuff = 'X\nY'
>>> print(stuff)
X
Y
>>> len(stuff)
3
```

You can also see that the length of the string `X\nY` is *three* characters because the newline character is a single character.

So when we look at the lines in a file, we need to *imagine* that there is a special invisible character called the newline at the end of each line that marks the end of the line.

So the newline character separates the characters in the file into lines.

7.4 Reading files

While the *file handle* does not contain the data for the file, it is quite easy to construct a `for` loop to read through and count each of the lines in a file:

```
fhand = open('mbox-short.txt')
count = 0
for line in fhand:
    count = count + 1
print('Line Count:', count)

# Code: http://www.pythonlearn.com/code3/open.py
```

We can use the file handle as the sequence in our `for` loop. Our `for` loop simply counts the number of lines in the file and prints them out. The rough translation of the `for` loop into English is, "for each line in the file represented by the file handle, add one to the `count` variable."

The reason that the `open` function does not read the entire file is that the file might be quite large with many gigabytes of data. The `open` statement takes the same amount of time regardless of the size of the file. The `for` loop actually causes the data to be read from the file.

When the file is read using a `for` loop in this manner, Python takes care of splitting the data in the file into separate lines using the newline character. Python reads each line through the newline and includes the newline as the last character in the `line` variable for each iteration of the `for` loop.

Because the `for` loop reads the data one line at a time, it can efficiently read and count the lines in very large files without running out of main memory to store the data. The above program can count the lines in any size file using very little memory since each line is read, counted, and then discarded.

If you know the file is relatively small compared to the size of your main memory, you can read the whole file into one string using the `read` method on the file handle.

```
>>> fhand = open('mbox-short.txt')
>>> inp = fhand.read()
>>> print(len(inp))
94626
>>> print(inp[:20])
From stephen.marquar
```

In this example, the entire contents (all 94,626 characters) of the file `mbox-short.txt` are read directly into the variable `inp`. We use string slicing to print out the first 20 characters of the string data stored in `inp`.

When the file is read in this manner, all the characters including all of the lines and newline characters are one big string in the variable *inp*. Remember that this

form of the `open` function should only be used if the file data will fit comfortably in the main memory of your computer.

If the file is too large to fit in main memory, you should write your program to read the file in chunks using a `for` or `while` loop.

7.5 Searching through a file

When you are searching through data in a file, it is a very common pattern to read through a file, ignoring most of the lines and only processing lines which meet a particular condition. We can combine the pattern for reading a file with string methods to build simple search mechanisms.

For example, if we wanted to read a file and only print out lines which started with the prefix "From:", we could use the string method *startswith* to select only those lines with the desired prefix:

```
fhand = open('mbox-short.txt')
count = 0
for line in fhand:
    if line.startswith('From:'):
        print(line)

# Code: http://www.pythonlearn.com/code3/search1.py
```

When this program runs, we get the following output:

```
From: stephen.marquard@uct.ac.za

From: louis@media.berkeley.edu

From: zqian@umich.edu

From: rjlowe@iupui.edu
...
```

The output looks great since the only lines we are seeing are those which start with "From:", but why are we seeing the extra blank lines? This is due to that invisible *newline* character. Each of the lines ends with a newline, so the `print` statement prints the string in the variable *line* which includes a newline and then `print` adds *another* newline, resulting in the double spacing effect we see.

We could use line slicing to print all but the last character, but a simpler approach is to use the *rstrip* method which strips whitespace from the right side of a string as follows:

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if line.startswith('From:'):
```

```
        print(line)

# Code: http://www.pythonlearn.com/code3/search2.py
```

When this program runs, we get the following output:

```
From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
From: cwen@iupui.edu
...
```

As your file processing programs get more complicated, you may want to structure your search loops using `continue`. The basic idea of the search loop is that you are looking for "interesting" lines and effectively skipping "uninteresting" lines. And then when we find an interesting line, we do something with that line.

We can structure the loop to follow the pattern of skipping uninteresting lines as follows:

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    # Skip 'uninteresting lines'
    if not line.startswith('From:'):
        continue
    # Process our 'interesting' line
    print(line)

# Code: http://www.pythonlearn.com/code3/search3.py
```

The output of the program is the same. In English, the uninteresting lines are those which do not start with "From:", which we skip using `continue`. For the "interesting" lines (i.e., those that start with "From:") we perform the processing on those lines.

We can use the `find` string method to simulate a text editor search that finds lines where the search string is anywhere in the line. Since `find` looks for an occurrence of a string within another string and either returns the position of the string or -1 if the string was not found, we can write the following loop to show lines which contain the string "@uct.ac.za" (i.e., they come from the University of Cape Town in South Africa):

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if line.find('@uct.ac.za') == -1: continue
    print(line)

# Code: http://www.pythonlearn.com/code3/search4.py
```