Chapter 5

Iteration

5.1 Updating variables

A common pattern in assignment statements is an assignment statement that updates a variable, where the new value of the variable depends on the old.

```
x = x + 1
```

This means "get the current value of x, add 1, and then update x with the new value."

If you try to update a variable that doesn't exist, you get an error, because Python evaluates the right side before it assigns a value to x:

```
>>> x = x + 1
NameError: name 'x' is not defined
```

Before you can update a variable, you have to ***initialize*** it, usually with a simple assignment:

```
x = 0
x = x + 1
```

Updating a variable by adding 1 is called an ***increment***; subtracting 1 is called a ***decrement***.

5.2 The while statement

Computers are often used to automate repetitive tasks. Repeating identical or similar tasks without making errors is something that computers do well and people do poorly. Because iteration is so common, Python provides several language features to make it easier.

One form of iteration in Python is the while statement. Here is a simple program that counts down from five and then says "Blastoff!".

```
n = 5
while n > 0:
    print(n)
    n = n - 1
print('Blastoff!')
```

You can almost read the while statement as if it were English. It means, "While n is greater than 0, display the value of n and then reduce the value of n by 1. When you get to 0, exit the while statement and display the word Blastoff!"

More formally, here is the flow of execution for a while statement:

Evaluate the condition, yielding True or False.

If the condition is false, exit the while statement and continue execution at the next statement.

If the condition is true, execute the body and then go back to step 1.

This type of flow is called a ***loop*** because the third step loops back around to the top. We call each time we execute the body of the loop an ***iteration***. For the above loop, we would say, "It had five iterations", which means that the body of the loop was executed five times.

The body of the loop should change the value of one or more variables so that eventually the condition becomes false and the loop terminates. We call the variable that changes each time the loop executes and controls when the loop finishes the ***iteration variable***. If there is no iteration variable, the loop will repeat forever, resulting in an ***infinite loop***.

5.3 Infinite loops

An endless source of amusement for programmers is the observation that the directions on shampoo, "Lather, rinse, repeat," are an infinite loop because there is no ***iteration variable*** telling you how many times to execute the loop.

In the case of countdown, we can prove that the loop terminates because we know that the value of n is finite, and we can see that the value of n gets smaller each time through the loop, so eventually we have to get to 0. Other times a loop is obviously infinite because it has no iteration variable at all.

5.4 "Infinite loops" and break

Sometimes you don't know it's time to end a loop until you get half way through the body. In that case you can write an infinite loop on purpose and then use the break statement to jump out of the loop.

This loop is obviously an ***infinite loop*** because the logical expression on the while statement is simply the logical constant True:

```
n = 10
while True:
print(n, end=' ')
n = n - 1
print('Done!')
```

If you make the mistake and run this code, you will learn quickly how to stop a runaway Python process on your system or find where the power-off button is on your computer. This program will run forever or until your battery runs out because the logical expression at the top of the loop is always true by virtue of the fact that the expression is the constant value True.

While this is a dysfunctional infinite loop, we can still use this pattern to build useful loops as long as we carefully add code to the body of the loop to explicitly exit the loop using break when we have reached the exit condition.

For example, suppose you want to take input from the user until they type done.
You could write:

```
while True:
line = input('> ')
if line == 'done':
break
print(line)
print('Done!')
```

 # Code: http://www.py4e.com/code3/copytildone1.py

The loop condition is True, which is always true, so the loop runs repeatedly until it hits the break statement.

Each time through, it prompts the user with an angle bracket. If the user types done, the break statement exits the loop. Otherwise the program echoes whatever the user types and goes back to the top of the loop. Here's a sample run:

```
    hello there
    hello there
    finished
    finished
    done
Done!
```

This way of writing while loops is common because you can check the condition anywhere in the loop (not just at the top) and you can express the stop condition affirmatively ("stop when this happens") rather than negatively ("keep going until that happens.").

5.5 Finishing iterations with continue

Sometimes you are in an iteration of a loop and want to finish the current iteration and immediately jump to the next iteration. In that case you can use the continue

statement to skip to the next iteration without finishing the body of the loop for the current iteration.

Here is an example of a loop that copies its input until the user types “done”, but treats lines that start with the hash character as lines not to be printed (kind of like Python comments).

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

# Code: http://www.py4e.com/code3/copytildone2.py
```

Here is a sample run of this new program with continue added.

```
hello there
hello there
  # don't print this
  print this!
print this!
  done
Done!
```

All the lines are printed except the one that starts with the hash sign because when the continue is executed, it ends the current iteration and jumps back to the while statement to start the next iteration, thus skipping the print statement.

5.6 Definite loops using for

Sometimes we want to loop through a ***set*** of things such as a list of words, the lines in a file, or a list of numbers. When we have a list of things to loop through, we can construct a ***definite*** loop using a for statement. We call the while statement an ***indefinite*** loop because it simply loops until some condition becomes False, whereas the for loop is looping through a known set of items so it runs through as many iterations as there are items in the set.

The syntax of a for loop is similar to the while loop in that there is a for statement and a loop body:

```
friends = ['Joseph', 'Glenn', 'Sally'] for friend in
friends:
print('Happy New Year:', friend)
print('Done!')
```

In Python terms, the variable friends is a list[1] of three strings and the for loop goes through the list and executes the body once for each of the three strings in the list resulting in this output:

```
Happy New Year: Joseph
Happy New Year: Glenn
Happy New Year: Sally
Done!
```

Translating this for loop to English is not as direct as the while, but if you think of friends as a ***set***, it goes like this: "Run the statements in the body of the for loop once for each friend ***in*** the set named friends."

Looking at the for loop, ***for*** and ***in*** are reserved Python keywords, and friend and friends are variables.

```
for friend in friends:
print('Happy New Year:', friend)
```

In particular, friend is the ***iteration variable*** for the for loop. The variable friend changes for each iteration of the loop and controls when the for loop completes. The ***iteration variable*** steps successively through the three strings stored in the friends variable.

5.7 Loop patterns

Often we use a for or while loop to go through a list of items or the contents of a file and we are looking for something such as the largest or smallest value of the data we scan through.

These loops are generally constructed by:

Initializing one or more variables before the loop starts

Performing some computation on each item in the loop body, possibly chang-ing the variables in the body of the loop

Looking at the resulting variables when the loop completes

We will use a list of numbers to demonstrate the concepts and construction of these loop patterns.

5.7.1 Counting and summing loops

For example, to count the number of items in a list, we would write the following for loop:

[1] We will examine lists in more detail in a later chapter.

```
count = 0
for itervar in [3, 41, 12, 9, 74, 15]:
count = count + 1
print('Count: ', count)
```

We set the variable count to zero before the loop starts, then we write a for loop to run through the list of numbers. Our ***iteration*** variable is named itervar and while we do not use itervar in the loop, it does control the loop and cause the loop body to be executed once for each of the values in the list.

In the body of the loop, we add 1 to the current value of count for each of the values in the list. While the loop is executing, the value of count is the number of values we have seen "so far".

Once the loop completes, the value of count is the total number of items. The total number "falls in our lap" at the end of the loop. We construct the loop so that we have what we want when the loop finishes.

Another similar loop that computes the total of a set of numbers is as follows:

```
total = 0
for itervar in [3, 41, 12, 9, 74, 15]:
total = total + itervar
print('Total: ', total)
```

In this loop we ***do*** use the ***iteration variable***. Instead of simply adding one to the count as in the previous loop, we add the actual number (3, 41, 12, etc.) to the running total during each loop iteration. If you think about the variable total, it contains the "running total of the values so far". So before the loop starts total is zero because we have not yet seen any values, during the loop total is the running total, and at the end of the loop total is the overall total of all the values in the list.

As the loop executes, total accumulates the sum of the elements; a variable used this way is sometimes called an ***accumulator***.

Neither the counting loop nor the summing loop are particularly useful in practice because there are built-in functions len() and sum() that compute the number of items in a list and the total of the items in the list respectively.

5.7.2 Maximum and minimum loops

To find the largest value in a list or sequence, we construct the following loop:

```
largest = None
print('Before:', largest)
for itervar in [3, 41, 12, 9, 74, 15]:
if largest is None or itervar > largest :
largest = itervar
print('Loop:', itervar, largest)
print('Largest:', largest)
```

When the program executes, the output is as follows:

```
Before: None
Loop: 3 3
Loop: 41 41
Loop: 12 41
Loop: 9 41
Loop: 74 74
Loop: 15 74
Largest: 74
```

The variable largest is best thought of as the "largest value we have seen so far". Before the loop, we set largest to the constant None. None is a special constant value which we can store in a variable to mark the variable as "empty".

Before the loop starts, the largest value we have seen so far is None since we have not yet seen any values. While the loop is executing, if largest is None then we take the first value we see as the largest so far. You can see in the first iteration when the value of itervar is 3, since largest is None, we immediately set largest to be 3.

After the first iteration, largest is no longer None, so the second part of the compound logical expression that checks itervar > largest triggers only when we see a value that is larger than the "largest so far". When we see a new "even larger" value we take that new value for largest. You can see in the program output that largest progresses from 3 to 41 to 74.

At the end of the loop, we have scanned all of the values and the variable largest now does contain the largest value in the list.

To compute the smallest number, the code is very similar with one small change:

```
smallest = None
print('Before:', smallest)
    for itervar in [3,41, 12,  9, 74, 15]:
        if smallest isNone or itervar < smallest:
            smallest =itervar
        print('Loop:',itervar, smallest)
    print('Smallest:',smallest)
```

Again, smallest is the "smallest so far" before, during, and after the loop executes. When the loop has completed, smallest contains the minimum value in the list.

Again as in counting and summing, the built-in functions max() and min() make writing these exact loops unnecessary.

The following is a simple version of the Python built-in min() function:

```
def     min(values):
smallest = None
for         value in values:
if              smallest is None or value < smallest:
smallest = value
return          smallest
```

In the function version of the smallest code, we removed all of the print statements so as to be equivalent to the min function which is already built in to Python.

5.8 Debugging

As you start writing bigger programs, you might find yourself spending more time debugging. More code means more chances to make an error and more places for bugs to hide.

One way to cut your debugging time is “debugging by bisection.” For example, if there are 100 lines in your program and you check them one at a time, it would take 100 steps.

Instead, try to break the problem in half. Look at the middle of the program, or near it, for an intermediate value you can check. Add a print statement (or something else that has a verifiable effect) and run the program.

If the mid-point check is incorrect, the problem must be in the first half of the program. If it is correct, the problem is in the second half.

Every time you perform a check like this, you halve the number of lines you have to search. After six steps (which is much less than 100), you would be down to one or two lines of code, at least in theory.

In practice it is not always clear what the “middle of the program” is and not always possible to check it. It doesn’t make sense to count lines and find the exact midpoint. Instead, think about places in the program where there might be errors and places where it is easy to put a check. Then choose a spot where you think the chances are about the same that the bug is before or after the check.

Chapter 6

Strings

6.1 A string is a sequence

A string is a ***sequence*** of characters. You can access the characters one at a time with the bracket operator:

```
fruit = 'banana'
letter = fruit[1]
```

The second statement extracts the character at index position 1 from the fruit variable and assigns it to the letter variable.

The expression in brackets is called an ***index***. The index indicates which character in the sequence you want (hence the name).

But you might not get what you expect:

```
print(letter)
a
```

For most people, the first letter of "banana" is b, not a. But in Python, the index is an offset from the beginning of the string, and the offset of the first letter is zero.

```
letter = fruit[0]
print(letter)
b
```

So b is the 0th letter ("zero-eth") of "banana", a is the 1th letter ("one-eth"), and n is the 2th ("two-eth") letter.

You can use any expression, including variables and operators, as an index, but the value of the index has to be an integer. Otherwise you get:

```
>>> letter = fruit[1.5]
TypeError: string indices must be integers
```

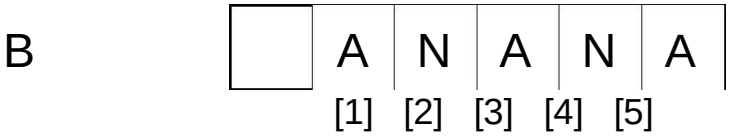

Figure 6.1: String Indexes

6.2 Getting the length of a string using len

len is a built-in function that returns the number of characters in a string:

```
fruit = 'banana'
len(fruit)
6
```

To get the last letter of a string, you might be tempted to try something like this:

```
length = len(fruit)
last = fruit[length]
IndexError: string index out of range
```

The reason for the IndexError is that there is no letter in 'banana' with the index Since we started counting at zero, the six letters are numbered 0 to 5. To get the last character, you have to subtract 1 from length:

```
last = fruit[length-1]
print(last)
a
```

Alternatively, you can use negative indices, which count backward from the end of the string. The expression fruit[-1] yields the last letter, fruit[-2] yields the second to last, and so on.

6.3 Traversal through a string with a loop

A lot of computations involve processing a string one character at a time. Often they start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a ***traversal***. One way to write a traversal is with a while loop:

```
index = 0
while index < len(fruit):
letter = fruit[index]
print(letter)
index = index + 1
```

This loop traverses the string and displays each letter on a line by itself. The loop condition is index \< len(fruit), so when index is equal to the length of the string, the condition is false, and the body of the loop is not executed. The last character accessed is the one with the index len(fruit)-1, which is the last character in the string.

Exercise 1: Write a while loop that starts at the last character in the string and works its way backwards to the first character in the string, printing each letter on a separate line, except backwards.

Another way to write a traversal is with a for loop:

```
for char in fruit:
print(char)
```

Each time through the loop, the next character in the string is assigned to the variable char. The loop continues until no characters are left.

6.4 String slices

A segment of a string is called a ***slice***. Selecting a slice is similar to selecting a character:

```
s = 'Monty Python'
print(s[0:5]) Monty
print(s[6:12]) Python
```

The operator returns the part of the string from the “n-eth” character to the “m-eth” character, including the first but excluding the last.

If you omit the first index (before the colon), the slice starts at the beginning of the string. If you omit the second index, the slice goes to the end of the string:

```
fruit = 'banana'
fruit[:3]
'ban'
fruit[3:]
'ana'
```

If the first index is greater than or equal to the second the result is an ***empty string***, represented by two quotation marks:

```
fruit = 'banana'
fruit[3:3]
''
```

An empty string contains no characters and has length 0, but other than that, it is the same as any other string.

Exercise 2: Given that fruit is a string, what does fruit[:] mean?

6.5 Strings are immutable

It is tempting to use the operator on the left side of an assignment, with the intention of changing a character in a string. For example:

```
greeting = 'Hello, world!'
greeting[0] = 'J'
TypeError: 'str' object does not support item assignment
```

The “object” in this case is the string and the “item” is the character you tried to assign. For now, an ***object*** is the same thing as a value, but we will refine that definition later. An ***item*** is one of the values in a sequence.

The reason for the error is that strings are ***immutable***, which means you can’t change an existing string. The best you can do is create a new string that is a variation on the original:

```
greeting = 'Hello, world!'
new_greeting = 'J' + greeting[1:]
print(new_greeting)
Jello, world!
```

This example concatenates a new first letter onto a slice of greeting. It has no effect on the original string.

6.6 Looping and counting

The following program counts the number of times the letter a appears in a string:

```
word = 'banana'
count = 0
for letter in word:
if letter == 'a':
count = count + 1
print(count)
```

This program demonstrates another pattern of computation called a ***counter***. The variable count is initialized to 0 and then incremented each time an a is found. When the loop exits, count contains the result: the total number of a’s.

Exercise 3:

Encapsulate this code in a function named count, and generalize it so that it accepts the string and the letter as arguments.

6.7 The in operator

The word in is a boolean operator that takes two strings and returns True if the first appears as a substring in the second:

```
'a' in 'banana'
True
'seed' in 'banana' False
```

6.8 String comparison

The comparison operators work on strings. To see if two strings are equal:

```
if word == 'banana':
print('All right, bananas.')
```

Other comparison operations are useful for putting words in alphabetical order:

```
if word < 'banana':
print('Your word,' + word + ', comes before banana.')
elif word > 'banana':
print('Your word,' + word + ', comes after banana.')
else:
print('All right, bananas.')
```

Python does not handle uppercase and lowercase letters the same way that people do. All the uppercase letters come before all the lowercase letters, so:

```
Your word, Pineapple, comes before banana.
```

A common way to address this problem is to convert strings to a standard format, such as all lowercase, before performing the comparison. Keep that in mind in case you have to defend yourself against a man armed with a Pineapple.

6.9 string methods

Strings are an example of Python ***objects***. An object contains both data (the actual string itself) and ***methods***, which are effectively functions that are built into the object and are available to any ***instance*** of the object.

Python has a function called dir which lists the methods available for an object. The type function shows the type of an object and the dir function shows the available methods.

```
    stuff = 'Hello world'
    type(stuff)
<class 'str'>
    dir(stuff)
['capitalize', 'casefold', 'center', 'count', 'encode', 'endswith' , 'expandtabs',
'find', 'format', 'format_map', 'index' , 'isalnum', 'isalpha', 'isdecimal',
'isdigit', 'isidentifier' , 'islower', 'isnumeric', 'isprintable', 'isspace' , 'istitle',
'isupper', 'join', 'ljust', 'lower', 'lstrip' , 'maketrans', 'partition', 'replace',
'rfind', 'rindex' , 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split' , 'splitlines',
'startswith', 'strip', 'swapcase', 'title' , 'translate', 'upper', 'zfill']
```

help(str.capitalize) Help on method_descriptor:

capitalize(...)
S.capitalize() -> str

Return a capitalized version of S, i.e. make the first character have upper case and the rest lower case.
>>>

While the dir function lists the methods, and you can use help to get some simple documentation on a method, a better source of documentation for string methods would be https://docs.python.org/3.5/library/stdtypes.html#string-methods.

Calling a ***method*** is similar to calling a function (it takes arguments and returns a value) but the syntax is different. We call a method by appending the method name to the variable name using the period as a delimiter.

For example, the method upper takes a string and returns a new string with all uppercase letters:

Instead of the function syntax upper(word), it uses the method syntax word.upper().

```
    word = 'banana'
    new_word = word.upper()
    print(new_word)
BANANA
```

This form of dot notation specifies the name of the method, upper, and the name of the string to apply the method to, word. The empty parentheses indicate that this method takes no argument.

A method call is called an ***invocation***; in this case, we would say that we are invoking upper on the word.

For example, there is a string method named find that searches for the position of one string within another:

```
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 parenthesis. We will cover tuples in Chapter 10

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

# Code: http://www.py4e.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] == '#':
```

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.

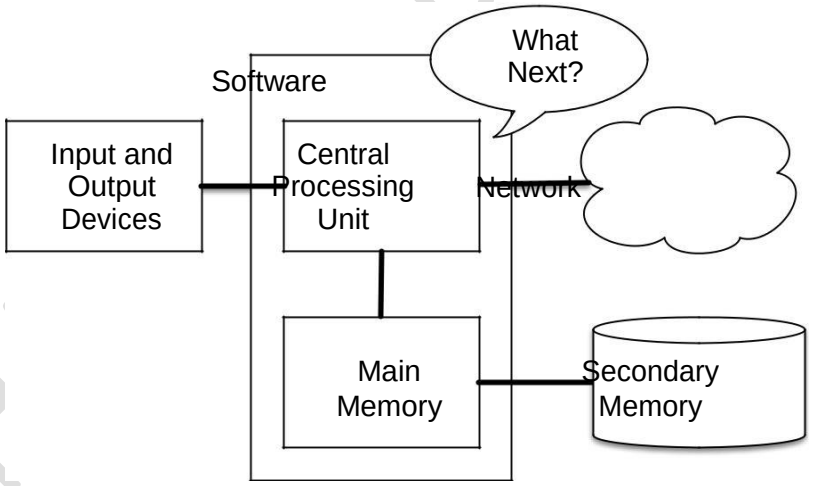

Figure 7.1: Secondary Memory

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.py4e.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.

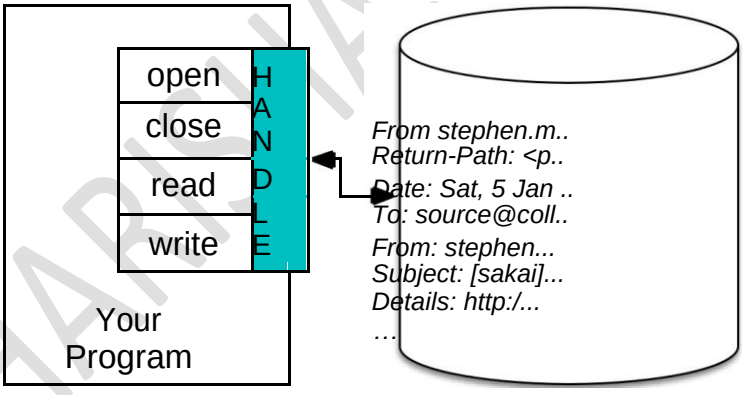

Figure 7.2: A File Handle

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.py4e.com/code3/mbox.txt

and a shortened version of the file is available from

www.py4e.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.py4e.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 slic-ing 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.py4e.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.py4e.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.py4e.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.py4e.com/code3/search4.py
```

Which produces the following output:

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
X-Authentication-Warning: set sender to stephen.marquard@uct.ac.za using -f
From: stephen.marquard@uct.ac.za
Author: stephen.marquard@uct.ac.za
From david.horwitz@uct.ac.za Fri Jan 4 07:02:32 2008
X-Authentication-Warning: set sender to david.horwitz@uct.ac.za using -f
From: david.horwitz@uct.ac.za
Author: david.horwitz@uct.ac.za
...
```

7.6 Letting the user choose the file name

We really do not want to have to edit our Python code every time we want to process a different file. It would be more usable to ask the user to enter the file name string each time the program runs so they can use our program on different files without changing the Python code.

This is quite simple to do by reading the file name from the user using input as follows:

```
fname = input('Enter the file name: ')
fhand = open(fname)
count = 0
for line in fhand:
if line.startswith('Subject:'):
count = count + 1
print('There were', count, 'subject lines in', fname)

# Code: http://www.py4e.com/code3/search6.py
```

We read the file name from the user and place it in a variable named fname and open that file. Now we can run the program repeatedly on different files.

```
python search6.py
Enter the file name: mbox.txt
There were 1797 subject lines in mbox.txt

python search6.py
Enter the file name: mbox-short.txt
There were 27 subject lines in mbox-short.txt
```

Before peeking at the next section, take a look at the above program and ask yourself, “What could go possibly wrong here?” or “What might our friendly user do that would cause our nice little program to ungracefully exit with a traceback, making us look not-so-cool in the eyes of our users?”

7.7 Using try, except, and open

I told you not to peek. This is your last chance.

What if our user types something that is not a file name?

```
python search6.py
Enter the file name: missing.txt
Traceback (most recent call last):
File "search6.py", line 2, in <module>
fhand = open(fname)
FileNotFoundError: [Errno 2] No such file or directory: 'missing.txt'

python search6.py
Enter the file name: na na boo boo
Traceback (most recent call last):
File "search6.py", line 2, in <module>
fhand = open(fname)
FileNotFoundError: [Errno 2] No such file or directory: 'na na boo boo'
```

Do not laugh. Users will eventually do every possible thing they can do to break your programs, either on purpose or with malicious intent. As a matter of fact, an important part of any software development team is a person or group called ***Quality Assurance*** (or QA for short) whose very job it is to do the craziest things possible in an attempt to break the software that the programmer has created.

The QA team is responsible for finding the flaws in programs before we have delivered the program to the end users who may be purchasing the software or paying our salary to write the software. So the QA team is the programmer's best friend.

So now that we see the flaw in the program, we can elegantly fix it using the try/except structure. We need to assume that the open call might fail and add recovery code when the open fails as follows:

```
fname = input('Enter the file name: ')
try:
fhand = open(fname)
except:
print('File cannot be opened:', fname)
exit()
count = 0
for line in fhand:
if line.startswith('Subject:'):
count = count + 1
print('There were', count, 'subject lines in', fname)

# Code: http://www.py4e.com/code3/search7.py
```

The exit function terminates the program. It is a function that we call that never returns. Now when our user (or QA team) types in silliness or bad file names, we "catch" them and recover gracefully:

Protecting the open call is a good example of the proper use of try and except in a Python program. We use the term “Pythonic” when we are doing something the “Python way”. We might say that the above example is the Pythonic way to open a file.

Once you become more skilled in Python, you can engage in repartee with other Python programmers to decide which of two equivalent solutions to a problem is “more Pythonic”. The goal to be “more Pythonic” captures the notion that programming is part engineering and part art. We are not always interested in just making something work, we also want our solution to be elegant and to be appreciated as elegant by our peers.

7.8 Writing files

To write a file, you have to open it with mode “w” as a second parameter:

```
fout = open('output.txt', 'w')
print(fout)
<_io.TextIOWrapper name='output.txt' mode='w' encoding='cp1252'>
```

If the file already exists, opening it in write mode clears out the old data and starts fresh, so be careful! If the file doesn’t exist, a new one is created.

The write method of the file handle object puts data into the file, returning the number of characters written. The default write mode is text for writing (and reading) strings.

```
line1 = "This here's the wattle,\n"
fout.write(line1)
24
```

Again, the file object keeps track of where it is, so if you call write again, it adds the new data to the end.

We must make sure to manage the ends of lines as we write to the file by explicitly inserting the newline character when we want to end a line. The print statement automatically appends a newline, but the write method does not add the newline automatically.

```
line2 = 'the emblem of our land.\n'
fout.write(line2)
```

When you are done writing, you have to close the file to make sure that the last bit of data is physically written to the disk so it will not be lost if the power goes off.

```
>>> fout.close()
```

We could close the files which we open for read as well, but we can be a little sloppy if we are only opening a few files since Python makes sure that all open files are closed when the program ends. When we are writing files, we want to explicitly close the files so as to leave nothing to chance.

7.9 Debugging

When you are reading and writing files, you might run into problems with whitespace. These errors can be hard to debug because spaces, tabs, and newlines are normally invisible:

```
    s = '1 2\t 3\n 4'
    print(s)
1 2  3
4
```

The built-in function repr can help. It takes any object as an argument and returns a string representation of the object. For strings, it represents whitespace characters with backslash sequences:

```
print(repr(s)) '1 2\t
3\n 4'
```

This can be helpful for debugging.

One other problem you might run into is that different systems use different characters to indicate the end of a line. Some systems use a newline, represented \n. Others use a return character, represented \r. Some use both. If you move files between different systems, these inconsistencies might cause problems.

For most systems, there are applications to convert from one format to another. You can find them (and read more about this issue) at wikipedia.org/wiki/Newline. Or, of course, you could write one yourself.