

FUTURE VISION BIE

One Stop for All Study Materials
& Lab Programs



Future Vision

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page



Or

Visit : <https://hemanthrajhemu.github.io>

Gain Access to All Study Materials according to VTU,
CSE – Computer Science Engineering,
ISE – Information Science Engineering,
ECE - Electronics and Communication Engineering
& MORE...

Join Telegram to get Instant Updates: https://bit.ly/VTU_TELEGRAM

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

WHATSAPP SHARE: <https://bit.ly/FVBIESHARE>

5	Iteration	57
5.1	Updating variables	57
5.2	The <code>while</code> statement	57
5.3	Infinite loops	58
5.4	Finishing iterations with <code>continue</code>	59
5.5	Definite loops using <code>for</code>	60
5.6	Loop patterns	61
5.6.1	Counting and summing loops	61
5.6.2	Maximum and minimum loops	62
5.7	Debugging	64
5.8	Glossary	64
5.9	Exercises	64
6	Strings	67
6.1	A string is a sequence	67
6.2	Getting the length of a string using <code>len</code>	68
6.3	Traversal through a string with a loop	68
6.4	String slices	69
6.5	Strings are immutable	70
6.6	Looping and counting	70
6.7	The <code>in</code> operator	71
6.8	String comparison	71
6.9	String methods	71
6.10	Parsing strings	74
6.11	Format operator	74
6.12	Debugging	75
6.13	Glossary	76
6.14	Exercises	77
7	Files	79
7.1	Persistence	79
7.2	Opening files	80
7.3	Text files and lines	81
7.4	Reading files	82
7.5	Searching through a file	83

7.6	Letting the user choose the file name	85
7.7	Using <code>try</code> , <code>except</code> , and <code>open</code>	86
7.8	Writing files	87
7.9	Debugging	88
7.10	Glossary	89
7.11	Exercises	89
8	Lists	91
8.1	A list is a sequence	91
8.2	Lists are mutable	92
8.3	Traversing a list	92
8.4	List operations	93
8.5	List slices	94
8.6	List methods	94
8.7	Deleting elements	95
8.8	Lists and functions	96
8.9	Lists and strings	97
8.10	Parsing lines	98
8.11	Objects and values	99
8.12	Aliasing	100
8.13	List arguments	100
8.14	Debugging	102
8.15	Glossary	105
8.16	Exercises	105
9	Dictionaries	107
9.1	Dictionary as a set of counters	109
9.2	Dictionaries and files	110
9.3	Looping and dictionaries	111
9.4	Advanced text parsing	113
9.5	Debugging	114
9.6	Glossary	115
9.7	Exercises	115

To choose an element from a sequence at random, you can use `choice`:

```
>>> t = [1, 2, 3]
>>> random.choice(t)
2
>>> random.choice(t)
3
```

The `random` module also provides functions to generate random values from continuous distributions including Gaussian, exponential, gamma, and a few more.

4.6 Adding new functions

So far, we have only been using the functions that come with Python, but it is also possible to add new functions. A *function definition* specifies the name of a new function and the sequence of statements that execute when the function is called. Once we define a function, we can reuse the function over and over throughout our program.

Here is an example:

```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print('I sleep all night and I work all day.')
```

`def` is a keyword that indicates that this is a function definition. The name of the function is `print_lyrics`. The rules for function names are the same as for variable names: letters, numbers and some punctuation marks are legal, but the first character can't be a number. You can't use a keyword as the name of a function, and you should avoid having a variable and a function with the same name.

The empty parentheses after the name indicate that this function doesn't take any arguments. Later we will build functions that take arguments as their inputs.

The first line of the function definition is called the *header*; the rest is called the *body*. The header has to end with a colon and the body has to be indented. By convention, the indentation is always four spaces. The body can contain any number of statements.

If you type a function definition in interactive mode, the interpreter prints ellipses (...) to let you know that the definition isn't complete:

```
>>> def print_lyrics():
...     print("I'm a lumberjack, and I'm okay.")
...     print('I sleep all night and I work all day.')
... 
```

To end the function, you have to enter an empty line (this is not necessary in a script).

Defining a function creates a variable with the same name.

```
>>> print(print_lyrics)
<function print_lyrics at 0xb7e99e9c>
>>> print(type(print_lyrics))
<class 'function'>
```

The value of `print_lyrics` is a *function object*, which has type “function”.

The syntax for calling the new function is the same as for built-in functions:

```
>>> print_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

Once you have defined a function, you can use it inside another function. For example, to repeat the previous refrain, we could write a function called `repeat_lyrics`:

```
def repeat_lyrics():
    print_lyrics()
    print_lyrics()
```

And then call `repeat_lyrics`:

```
>>> repeat_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

But that’s not really how the song goes.

4.7 Definitions and uses

Pulling together the code fragments from the previous section, the whole program looks like this:

```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print('I sleep all night and I work all day.')

def repeat_lyrics():
    print_lyrics()
    print_lyrics()

repeat_lyrics()
```

Code: <http://www.py4e.com/code3/lyrics.py>

This program contains two function definitions: `print_lyrics` and `repeat_lyrics`. Function definitions get executed just like other statements, but the effect is to create function objects. The statements inside the function do not get executed until the function is called, and the function definition generates no output.

As you might expect, you have to create a function before you can execute it. In other words, the function definition has to be executed before the first time it is called.

Exercise 2: Move the last line of this program to the top, so the function call appears before the definitions. Run the program and see what error message you get.

Exercise 3: Move the function call back to the bottom and move the definition of `print_lyrics` after the definition of `repeat_lyrics`. What happens when you run this program?

4.8 Flow of execution

In order to ensure that a function is defined before its first use, you have to know the order in which statements are executed, which is called the *flow of execution*.

Execution always begins at the first statement of the program. Statements are executed one at a time, in order from top to bottom.

Function *definitions* do not alter the flow of execution of the program, but remember that statements inside the function are not executed until the function is called.

A function call is like a detour in the flow of execution. Instead of going to the next statement, the flow jumps to the body of the function, executes all the statements there, and then comes back to pick up where it left off.

That sounds simple enough, until you remember that one function can call another. While in the middle of one function, the program might have to execute the statements in another function. But while executing that new function, the program might have to execute yet another function!

Fortunately, Python is good at keeping track of where it is, so each time a function completes, the program picks up where it left off in the function that called it. When it gets to the end of the program, it terminates.

What's the moral of this sordid tale? When you read a program, you don't always want to read from top to bottom. Sometimes it makes more sense if you follow the flow of execution.

4.9 Parameters and arguments

Some of the built-in functions we have seen require arguments. For example, when you call `math.sin` you pass a number as an argument. Some functions take more than one argument: `math.pow` takes two, the base and the exponent.

Inside the function, the arguments are assigned to variables called *parameters*. Here is an example of a user-defined function that takes an argument:

```
def print_twice(bruce):
    print(bruce)
    print(bruce)
```

This function assigns the argument to a parameter named `bruce`. When the function is called, it prints the value of the parameter (whatever it is) twice.

This function works with any value that can be printed.

```
>>> print_twice('Spam')
Spam
Spam
>>> print_twice(17)
17
17
>>> import math
>>> print_twice(math.pi)
3.141592653589793
3.141592653589793
```

The same rules of composition that apply to built-in functions also apply to user-defined functions, so we can use any kind of expression as an argument for `print_twice`:

```
>>> print_twice('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> print_twice(math.cos(math.pi))
-1.0
-1.0
```

The argument is evaluated before the function is called, so in the examples the expressions `'Spam '*4` and `math.cos(math.pi)` are only evaluated once.

You can also use a variable as an argument:

```
>>> michael = 'Eric, the half a bee.'
>>> print_twice(michael)
Eric, the half a bee.
Eric, the half a bee.
```

The name of the variable we pass as an argument (`michael`) has nothing to do with the name of the parameter (`bruce`). It doesn't matter what the value was called back home (in the caller); here in `print_twice`, we call everybody `bruce`.

4.10 Fruitful functions and void functions

Some of the functions we are using, such as the math functions, yield results; for lack of a better name, I call them *fruitful functions*. Other functions, like `print_twice`, perform an action but don't return a value. They are called *void functions*.

When you call a fruitful function, you almost always want to do something with the result; for example, you might assign it to a variable or use it as part of an expression:

```
x = math.cos(radians)
golden = (math.sqrt(5) + 1) / 2
```

When you call a function in interactive mode, Python displays the result:

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

But in a script, if you call a fruitful function and do not store the result of the function in a variable, the return value vanishes into the mist!

```
math.sqrt(5)
```

This script computes the square root of 5, but since it doesn't store the result in a variable or display the result, it is not very useful.

Void functions might display something on the screen or have some other effect, but they don't have a return value. If you try to assign the result to a variable, you get a special value called `None`.

```
>>> result = print_twice('Bing')
Bing
Bing
>>> print(result)
None
```

The value `None` is not the same as the string "None". It is a special value that has its own type:

```
>>> print(type(None))
<class 'NoneType'>
```

To return a result from a function, we use the `return` statement in our function. For example, we could make a very simple function called `addtwo` that adds two numbers together and returns a result.


```
def addtwo(a, b):  
    added = a + b  
    return added
```

```
x = addtwo(3, 5)  
print(x)
```

Code: <http://www.py4e.com/code3/addtwo.py>

When this script executes, the `print` statement will print out “8” because the `addtwo` function was called with 3 and 5 as arguments. Within the function, the parameters `a` and `b` were 3 and 5 respectively. The function computed the sum of the two numbers and placed it in the local function variable named `added`. Then it used the `return` statement to send the computed value back to the calling code as the function result, which was assigned to the variable `x` and printed out.

4.11 Why functions?

It may not be clear why it is worth the trouble to divide a program into functions. There are several reasons:

- Creating a new function gives you an opportunity to name a group of statements, which makes your program easier to read, understand, and debug.
- Functions can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have to make it in one place.
- Dividing a long program into functions allows you to debug the parts one at a time and then assemble them into a working whole.
- Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

Throughout the rest of the book, often we will use a function definition to explain a concept. Part of the skill of creating and using functions is to have a function properly capture an idea such as “find the smallest value in a list of values”. Later we will show you code that finds the smallest in a list of values and we will present it to you as a function named `min` which takes a list of values as its argument and returns the smallest value in the list.

4.12 Debugging

If you are using a text editor to write your scripts, you might run into problems with spaces and tabs. The best way to avoid these problems is to use spaces exclusively (no tabs). Most text editors that know about Python do this by default, but some don’t.

Tabs and spaces are usually invisible, which makes them hard to debug, so try to find an editor that manages indentation for you.

Also, don't forget to save your program before you run it. Some development environments do this automatically, but some don't. In that case, the program you are looking at in the text editor is not the same as the program you are running.

Debugging can take a long time if you keep running the same incorrect program over and over!

Make sure that the code you are looking at is the code you are running. If you're not sure, put something like `print("hello")` at the beginning of the program and run it again. If you don't see `hello`, you're not running the right program!

4.13 Glossary

algorithm A general process for solving a category of problems.

argument A value provided to a function when the function is called. This value is assigned to the corresponding parameter in the function.

body The sequence of statements inside a function definition.

composition Using an expression as part of a larger expression, or a statement as part of a larger statement.

deterministic Pertaining to a program that does the same thing each time it runs, given the same inputs.

dot notation The syntax for calling a function in another module by specifying the module name followed by a dot (period) and the function name.

flow of execution The order in which statements are executed during a program run.

fruitful function A function that returns a value.

function A named sequence of statements that performs some useful operation. Functions may or may not take arguments and may or may not produce a result.

function call A statement that executes a function. It consists of the function name followed by an argument list.

function definition A statement that creates a new function, specifying its name, parameters, and the statements it executes.

function object A value created by a function definition. The name of the function is a variable that refers to a function object.

header The first line of a function definition.

import statement A statement that reads a module file and creates a module object.

module object A value created by an `import` statement that provides access to the data and code defined in a module.

parameter A name used inside a function to refer to the value passed as an argument.

pseudorandom Pertaining to a sequence of numbers that appear to be random, but are generated by a deterministic program.

return value The result of a function. If a function call is used as an expression, the return value is the value of the expression.

void function A function that does not return a value.

4.14 Exercises

Exercise 4: What is the purpose of the “def” keyword in Python?

- a) It is slang that means “the following code is really cool”
- b) It indicates the start of a function
- c) It indicates that the following indented section of code is to be stored for later
- d) b and c are both true
- e) None of the above

Exercise 5: What will the following Python program print out?

```
def fred():
    print("Zap")
```

```
def jane():
    print("ABC")
```

```
jane()
fred()
jane()
```

- a) Zap ABC jane fred jane
- b) Zap ABC Zap
- c) ABC Zap jane
- d) ABC Zap ABC
- e) Zap Zap Zap

Exercise 6: Rewrite your pay computation with time-and-a-half for over-time and create a function called `compute_pay` which takes two parameters (hours and rate).

```
Enter Hours: 45
Enter Rate: 10
Pay: 475.0
```

Exercise 7: Rewrite the grade program from the previous chapter using a function called `compute_grade` that takes a score as its parameter and returns a grade as a string.

Score	Grade
>= 0.9	A
>= 0.8	B
>= 0.7	C
>= 0.6	D
< 0.6	F

```
Enter score: 0.95
A
```

```
Enter score: perfect
Bad score
```

```
Enter score: 10.0
Bad score
```

```
Enter score: 0.75
C
```

```
Enter score: 0.5
F
```

Run the program repeatedly to test the various different values for input.

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:

1. Evaluate the condition, yielding `True` or `False`.
2. If the condition is false, exit the `while` statement and continue execution at the next statement.
3. 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.

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.”).

5.4 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.

5.5 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 list¹ 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:

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

```
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.

5.6 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 changing 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.

5.6.1 Counting and summing loops

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

```
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.

5.6.2 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 is None 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.

5.7 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.

5.8 Glossary

accumulator A variable used in a loop to add up or accumulate a result.

counter A variable used in a loop to count the number of times something happened. We initialize a counter to zero and then increment the counter each time we want to “count” something.

decrement An update that decreases the value of a variable.

initialize An assignment that gives an initial value to a variable that will be updated.

increment An update that increases the value of a variable (often by one).

infinite loop A loop in which the terminating condition is never satisfied or for which there is no terminating condition.

iteration Repeated execution of a set of statements using either a function that calls itself or a loop.

5.9 Exercises

Exercise 1: Write a program which repeatedly reads numbers until the user enters “done”. Once “done” is entered, print out the total, count, and average of the numbers. If the user enters anything other than a number, detect their mistake using `try` and `except` and print an error message and skip to the next number.

```
Enter a number: 4
Enter a number: 5
Enter a number: bad data
Invalid input
Enter a number: 7
Enter a number: done
16 3 5.333333333333333
```

Exercise 2: Write another program that prompts for a list of numbers as above and at the end prints out both the maximum and minimum of the numbers instead of the average.

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-th”) of “banana”, “a” is the 1th letter (“one-th”), and “n” is the 2th (“two-th”) 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
```

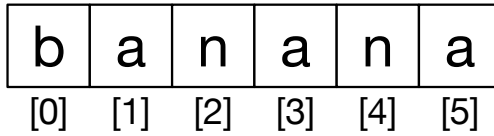



Figure 6.1: String Indexes

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 6. 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-th” character to the “m-th” 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/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/library/stdtypes.html#string-methods>

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
>>> spos = data.find(' ',atpos)
>>> print(spos)
31
>>> host = data[atpos+1:spos]
>>> 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/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 tuple¹. 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/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('> ')
    if line[0] == '#':
```

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


```

        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] == '#):
```

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/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.

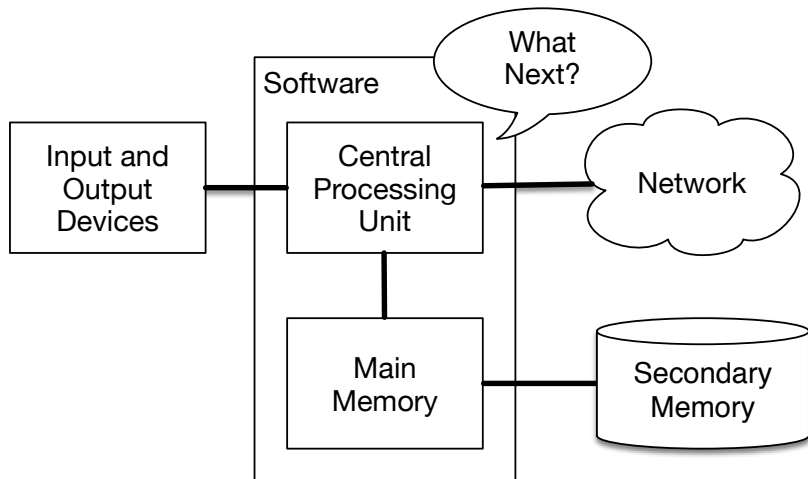


Figure 7.1: Secondary Memory

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.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.

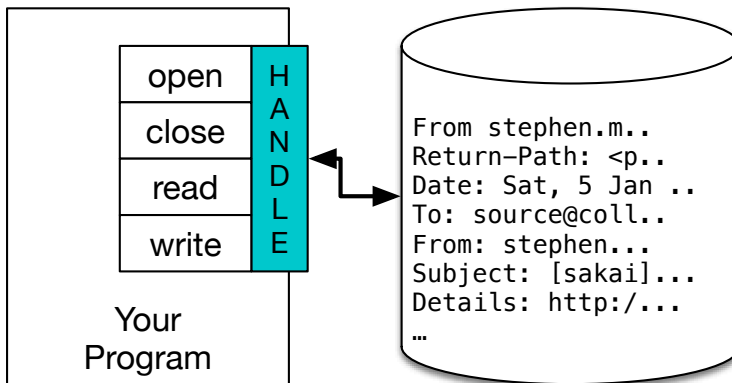


Figure 7.2: A File Handle

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 <https://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 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`. It is a good idea to store the output of `read` as a variable because each call to `read` exhausts the resource:

```
>>> fhand = open('mbox-short.txt')
>>> print(len(fhand.read()))
94626
>>> print(len(fhand.read()))
0
```

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
...
```

Here we also use the contracted form of the if statement where we put the continue on the same line as the if. This contracted form of the if functions the same as if the continue were on the next line and indented.

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:

```

python search7.py
Enter the file name: mbox.txt
There were 1797 subject lines in mbox.txt

```

```

python search7.py
Enter the file name: na na boo boo
File cannot be opened: na na boo boo

```

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)
24
```

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.

7.10 Glossary

catch To prevent an exception from terminating a program using the `try` and `except` statements.

newline A special character used in files and strings to indicate the end of a line.

Pythonic A technique that works elegantly in Python. “Using `try` and `except` is the *Pythonic* way to recover from missing files”.

Quality Assurance A person or team focused on insuring the overall quality of a software product. QA is often involved in testing a product and identifying problems before the product is released.

text file A sequence of characters stored in permanent storage like a hard drive.

7.11 Exercises

Exercise 1: Write a program to read through a file and print the contents of the file (line by line) all in upper case. Executing the program will look as follows:

```
python shout.py
Enter a file name: mbox-short.txt
FROM STEPHEN.MARQUARD@UCT.AC.ZA SAT JAN 5 09:14:16 2008
RETURN-PATH: <POSTMASTER@COLLAB.SAKAIPROJECT.ORG>
RECEIVED: FROM MURDER (MAIL.UMICH.EDU [141.211.14.90])
          BY FRANKENSTEIN.MAIL.UMICH.EDU (CYRUS V2.3.8) WITH LMTPA;
          SAT, 05 JAN 2008 09:14:16 -0500
```

You can download the file from www.py4e.com/code3/mbox-short.txt

Exercise 2: Write a program to prompt for a file name, and then read through the file and look for lines of the form:

```
X-DSPAM-Confidence: 0.8475
```

When you encounter a line that starts with “X-DSPAM-Confidence:” pull apart the line to extract the floating-point number on the line. Count these lines and then compute the total of the spam confidence values from these lines. When you reach the end of the file, print out the average spam confidence.

```
Enter the file name: mbox.txt
Average spam confidence: 0.894128046745
```

```
Enter the file name: mbox-short.txt
Average spam confidence: 0.750718518519
```

Test your file on the *mbox.txt* and *mbox-short.txt* files.

Exercise 3: Sometimes when programmers get bored or want to have a bit of fun, they add a harmless *Easter Egg* to their program. Modify the program that prompts the user for the file name so that it prints a funny message when the user types in the exact file name “na na boo boo”. The program should behave normally for all other files which exist and don’t exist. Here is a sample execution of the program:

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

```
python egg.py
Enter the file name: missing.tyxt
File cannot be opened: missing.tyxt
```

```
python egg.py
Enter the file name: na na boo boo
NA NA BOO BOO TO YOU - You have been punk'd!
```

We are not encouraging you to put Easter Eggs in your programs; this is just an exercise.