Chapter 4, Repetition and Iteration

John M. Morrison

April 16, 2021

Contents

0 Introduction

The goal of this chapter is to bring you to the point where Python is *Turing-Complete*, which means that, given sufficient time and memory, it can solve any solvable computational problem.

In the beginning, all Python programs were lists of worker statements that executed *in seriatum*. Then we started realizing, "If we keep doing the same thing over and over again, can't we store a procedure under a single name so we can reuse it?" This brought us to functions. Functions are objects that store sets of instructions. In fact, they are first-class objects of type `<class 'function'>`.

We then decided that our programs should be able to make decisions based on visible variables; this brings us conditional logic.

The flow of programs is no longer linear. However, if we do things right, it is structured. And the use of functions can help make programs more understandable if we choose names for our functions that are evocative of their actions.

We have actually taken the final step on the road to Turing-completeness: all repetition in programs can be done by recursion. However, the appearance of our code might be somewhat recondite and opaque. Take note, however, that recursion can be a very handy tool for solving problems that initially appear to be unwieldy impossible snarls.

Python provides two programming constructs for repetition: `while` and `for`. It also provides objects called *iterators* that walk through collections and show us objects in succession, and some python objecs are *iterables*, which means they can be walked through with a definite loop.

Once we master these ideas, Python becomes a Turing-complete language; it, given sufficient memory and time, can be used to solve any computational problems that is solvable. Let us now set out on this next exploration.

1 Iterables and Definite Loops

Iterables show us a collection of objects in succession. A very simple iterable is called a `range` object. If you cast a `range` object as a list, you can see all of the values it exposes.

```
>>> range(10)
range(0, 10)
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1,5)
```

```
range(1, 5)
>>> list(range(1,5))
[1, 2, 3, 4]
>>> list(range(2,101,7))
[2, 9, 16, 23, 30, 37, 44, 51, 58, 65, 72, 79, 86, 93, 100]
```

To do something with each value of an iterable, you use the `for` keyword as follows.

```
>>> for quack in range(10):
...     print(f"{quack}\t\t{quack*quack}")
...
0       0
1       1
2       4
3       9
4       16
5       25
6       36
7       49
8       64
9       81
```

Lurking inside of any collection (list/tuple/string/dictionary/set) is an iterator that serves its items up in some order; this object is what makes these collections iterable. For a hashed collection, the iterator serves up the items in no particualar discernable order. A dictionary iterates through its keys. The use of the `for` loop automatically brings out that feature.

```
>>> for f in os.listdir():
...     print(f"{f}\t\t{os.path.getsize(f)}\t{os.path.abspath(f)}"
...
docstring.py        200 /Users/morrison/book/ppp/p1Code/docstring.py
evil\_scope.py        78  /Users/morrison/book/ppp/p1Code/evil\_scope.py
grades.py       671 /Users/morrison/book/ppp/p1Code/grades.py
rectangle.py        215 /Users/morrison/book/ppp/p1Code/rectangle.py
```

Here we see the name, size and absolute path of each file in a directory.

Watch the `for` loop work on tuples and strings.

```
>>> t = (1,2,3,4,5)
>>> tot = 0
>>> for k in t:
...     tot += k
...
```

```
>>> print(tot)
15
```

We just found the sum of the entries in the tuple. A string's iterator walks through the string one character at a time.

```
>>> for k in s:
...     print(k*6)
...
ffffff
oooooo
oooooo
mmmmmm
eeeeee
nnnnnn
tttttt
```

A Cautionary Tale Watch this attempt to zero out a list.

```
>>> x = [1,2,3,4,5,6]
>>> for k in x:
...     k = 0
...
>>> x
[1, 2, 3, 4, 5, 6]
```

What happened? What the iterator did is assign each element in succession to the temporary name `k`, so reassigning `k` has no effect on the list itself.

Contrast that to this.

```
>>> for k in range(len(x)):
...     x[k] = 0
...
>>> x
[0, 0, 0, 0, 0, 0]
```

Here we are indexing into the list using copies of the integers starting at `0` and ending before `len(x)`. Note that the indexed entries of the list are lvalues, so we can assign to them.

The `for` loop is a *definite* loop; its purpose is to walk through a specified collection of objects, or visit all of the objects offered up by an iterable. Its "food" is an iteraable. Objects of type `range` are iterables. Lists, tuples, and strings all automatically offer their iterators when used in a `for` loop.

Two Useful Modifiers Here is clunkiness of the first class.

```
>>> for k in range(len(x) - 1, -1, -1):
...     print(x[k])
...
elephant
dingo
carical
bat
aardvark
>>>
```

This is a far better way. Use it.

```
>>> for k in reversed(x):
...     print(k)
...
elephant
dingo
carical
bat
aardvark
```

So, the `reversed` function hands you an iterator that walks backward through a collection.

Now consider this. It's kinda ugly.

```
>>> for k in range(len(x)):
...     print(f"x[{k}] = {x[k]}")
...
x[0] = aardvark
x[1] = bat
x[2] = carical
x[3] = dingo
x[4] = elephant
>>>
```

Now let's see the better way.

```
>>> for k, item in enumerate(x):
...     print(f"x[{k}] = {item}")
...
x[0] = aardvark
x[1] = bat
x[2] = carical
```

```
x[3] = dingo
x[4] = elephant
```

In general, you should only rarely walk through a list or tuple by traversing its indices. These two tools will help make that occasion rare. The underlying collection is never altered by either of them.

Programming Exercises

1. Can you use `reversed` and `enumerate` on a `range` object?
2. Write a loop that will produce this output

   ```
   5. aardvark
   4. bat
   3. carical
   2. dingo
   1. elephant
   ```

   Can you come up with two reasonable solutions?

2 File IO

Reading and writing text files in Python is achieved using the built-in `open` function. This function has one required argument, a filename. The second, optional, argument is the mode for opening the file; its default value opens a file for reading. It is recommended you open a file for reading explicitly; remember, "explicit is better than implicit." When we open a file, there is an iterator in it that can read the file line-by-line.

r	This is read mode. It is the default mode as well. The file you are reading from needs to exist, and you need to have read permission, or your program will error out.
w	This is write mode. It clobbers any existing If the file exists and you lack write permission, your program will error out. file you open. If the file does not exist, it is created.
a	This is append mode. It causes additional text to be appended to the end of an existing file. If the file does not exist, it gets created.
b	This is binary mode. It opens a file as raw bytes and can be combined with read or write mode.
t	This is text mode. It opens a file as text; it is the default.
x	This opens a file for writing but throws a `FileExistsError` if the file exists. if the file exists.

In read mode, there are several ways to access the contents of the file. Create this text file.

```
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789
!@#$%^&*()_+
,./;'[]\<>?:"{}|+
```

Now we will demonstrate some features in a live session. Let us open the file for reading.

```
>>> fp = open("sampler.txt", "r")
>>> fp
<_io.TextIOWrapper name='sampler.txt' mode='r' encoding='UTF-8'>
>>>
```

Now we take a byte.

```
>>> fp.read(1)
'a'
>>> fp.tell()
1
```

We pass the number of bytes we want to read to `read` and they are returned. In addition, the file object has inside it a pointer to the next byte it is to read. You can be told that byte by using `tell`. You can move to any byte by using `seek`.

```
>>> fp.seek(10)
10
>>> fp.read(1)
'k'
```

To go back to the beginning of the file, use `seek(0)`. To read the rest of the file, pass no argument like so.

```
>>> fp.seek(0)
0
>>> fp.read()
'abcdefghijklmnopqrstuvwxy... 789\n!@#%$^&*()_+\n,./;\'[]\\<>?:"{}|+\n'
```

In this case, you get the entire file in a single string. If the file is large, you might not want to do that. In addition to having the file pointer, the file object has its own iterator. Watch this.

```
>>> for line in fp: print(line)
...
abcdefghijklmnopqrstuvwxyz

ABCDEFGHIJKLMNOPQRSTUVWXYZ

0123456789

!@#%$^&*()_+

,./;'[]\<>?:"{}|+
```

Hey, why did this double space? Remember, `print` puts a newline at the end by default. But each line of the file has a newline at the end as well. We can suppress this annoyance as follows.

```
>>> for line in fp: print(line, end="")
...
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789
!@#%$^&*()_+
,./;'[]\<>?:"{}|+
```

Here is one other nifty trick.

```
>>> fp.seek(0)
0
```

```
>>> stuff = fp.readlines()
>>> stuff[0]
'abcdefghijklmnopqrstuvwxyz\n'
>>> stuff[1]
'ABCDEFGHIJKLMNOPQRSTUVWXYZ\n'
>>> len(stuff)
5
```

The `readlines` method returns a list of strings each containing a line of the file in seriatum.

Now, let us turn to writing files. The `w` mode corresponds to C's and UNIX's write mode for `open` and `fopen`. If you open an existing file for writing it will be clobbered. You must use append mode (`a`) to open a file and to add text to it. Both write and append mode will create the file if it does not yet exist. Let us now create a file in an interactive session.

```
>>> out\_file = open("bilge.txt", "w")
>>> out\_file.write("quack")
5
>>> out\_file.write("moo")
3
>>> out\_file.write("baa")
3
>>> out\_file.close()
>>> in\_file = open("bilge.txt", "r")
>>> print(in\_file.read())
quackmoobaa
```

What can be gleaned from this session? Firstly, the `write` method returns the number of bytes written to the file. Notice that it *does not* put a newline at the end of the byte sequence you are entering. You have to do this yourself or you will end up with a file with one long line. Also beware that the file is not saved until you close it. Why is this? FileIO in Python is buffered so that python is not pestering the kernel every time it wants to write a character to a file. It has a temporary storage place for the characters you write, and when that storage place gets full, Python ships the buffer to the file. Similarly, when you read from a file, you are actually reading from a buffer. Most kernels will do file operation a disk sector at a time. Closing the file causes the buffere to be flushed into the file, where it belongs, and it discontinues the use of certian system resources.

Know when to `flush` You can trigger this manually with the `flush` method. Observe this.

```
>>> fout = open("dragons.slay", "w")
>>> fout.write("Bart, eat my shorts")
19
>>> fout.write("NOW")
3
```

Now, during this session, do this

```
unix> cat dragons.slay
```

and you will see that the file is empty! Now do this.

```
>>> fout.flush()
```

```
unix> cat dragons.slay
Bart, eat my shortsNOW
unix>
```

Most of the time you never need to flush, because if you do this

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

the buffer is automatically flushed.

2.1 A Helpful Tool: Raw Strings

Python supports a version of strings called *raw strings*. To make a raw string literal, just prepend with an `r`. When Python encounters a raw string, all backslashes are read literally. No special meaning is given them by the language. This interactive session shows how it works.

```
>>> path = 'C:\nasty\mean\oogly'
>>> print (path)
C:
asty\mean\oogly
>>> path = r'C:\nasty\mean\oogly'
>>> print (path)
C:\nasty\mean\ugly
>>>
```

Notice that in the raw string, the `\n` did not expand to a newline; it was a literal backslash-n. This is a great convenience when dealing with file paths in Windoze and for writing regular expressions. You can also make a triple-quoted string a raw string.

Warning! You may not end a raw string with a \. This causes the close-quote to be escaped to a literal character and causes a string-delimiter leak. Think for a moment: there is an easy work-around for this!

3 Some FileIO Applications

Let us try to imitate the UNIX command `cat`, which puts a file to the screen. We begin by developing an outline for what we want to do.

```
# The filename we wish to cat should be a command-line argument
# It will be argv[1].
# We will open this file for reading
# read the contents
# put them to the screen
# finish up by closing the file.
```

Now, let's get started with the command-line arguments.

```
# The filename we wish to cat should be a command-line argument
from sys import argv
filename = argv[1]
# We will open this file for reading
# read the contents
# put them to the screen
# finish up by closing the file.
```

Now, let's open the file and aspirate its contents.

```
# The filename we wish to cat should be a command-line argument
from sys import argv
filename = argv[1]
# We will open this file for reading using the \texttt{with} statement
with open(filename, "r") as fp:
# read the contents
    s = fp.read()
# put them to the screen
```

Now let's finish up.

```
# The filename we wish to cat should be a command-line argument
from sys import argv
filename = argv[1]
# We will open this file for reading
```

```
with open(filename, "r") as fp:
# read the contents
    s = fp.read()
# put them to the screen
print(s)
```

You think we are done? Think again. It's time to run this raw program.

```
from sys import argv
filename = argv[1]
with open(filename, "r") as fp:
    s = fp.read()
print(s)
```

Create this little test file, `simple.txt`

```
Here is a file
with a little text in it.
We are going to use this to see how
well our at program works.
```

Now run our program on it.

```
unix> python cat.py simple.txt
Here is a file
with a little text in it.
We are going to use this to see how
well our at program works.
```

It looks great, eh? Now let us do this.

```
unix> python cat.py
Traceback (most recent call last):
  File "cat.py", line 2, in <module>
    filename = argv[1]
IndexError: list index out of range
```

Uh oh. That idiotic end user. Let us now fend off this form of folly with a little parry.

```
from sys import argv
if len(argv) < 2:
    print("Usage:  python cat.py filename; enter a filename")
    quit()
```

```
filename = argv[1]
with open(filename, "r") as fp:
    s = fp.read()
print(s)
```

Now let's run this.

```
python cat.py
Usage:  python cat.py filename; enter a filename
```

Think we are in the clear? Check this out.

```
MAC:Thu Dec 06:14:23:ppp> python cat.py not.real
Traceback (most recent call last):
  File "cat.py", line 6, in <module>
    fp = open(filename, "r")
FileNotFoundError: [Errno 2] No such file or directory: 'not.real'
```

We ran this, giving a file that fails to exist. There are two ways to handle this. One way is to take exception.

```
from sys import argv
if len(argv) < 2:
    print("Usage:  python cat.py filename; enter a filename")
    quit()
filename = argv[1]
try:
    with open(filename, "r") as fp:
        s = fp.read()
    print(s)
except FileNotFoundError:
    print("File {0} does not exist.".format(filename))
    quit()
```

```
unix> python cat.py not.real
File not.real does not exist.
```

Another way is to use `os.path.exists`

```
from sys import argv
import os
if len(argv) < 2:
    print("Usage:  python cat.py filename; enter a filename")
    quit()
```

```
filename = argv[1]
if not os.path.exists(filename):
    print("File {0} does not exist.".format(filename))
    quit()
with open(filename, "r") as fp:
    s = fp.read()
print(s)
```

Feel safe? Not yet. One more form of nonsense can occur.

```
unix> chmod 000 simple.txt
```

We just revoked all read, write, and execute privileges for this file. Now watch this.

```
unix> python cat.py simple.txt
Traceback (most recent call last):
  File "cat.py", line 10, in <module>
    fp = open(filename, "r")
PermissionError: [Errno 13] Permission denied: 'simple.txt'
```

We could take exception and handle this that way. Or, here is another useful too, `os.access`. Let's do this.

```
unix> chmod 644 simple.txt
```

Now start Python in interactive mode.

```
>>> os.access("simple.txt", os.F_OK)
True
>>> os.access("simple.txt", os.R_OK)
True
>>> os.access("simple.txt", os.W_OK)
True
>>> os.access("simple.txt", os.X_OK)
False
```

The first line tests if the file exists. It does. The second, third and fourth check for read, write and execute permission.

```
from sys import argv
import os
if len(argv) < 2:
    print("Usage:  python cat.py filename; enter a filename")
```

```
    quit()
filename = argv[1]
if not os.path.exists(filename):
    print("File {0} does not exist.".format(filename))
    quit()
if not os.access(filename, os.R_OK):
    print("You lack permission to read file {0}.  Bailing....")
    quit()
with  open(filename, "r") as fp:
    s = fp.read()
print(s)
```

Now revoke all permissions and run

```
unix> chmod 000 simple.txt
MAC:Thu Dec 06:15:09:ppp> python cat.py simple.txt
You lack permission to read file {0}.  Bailing....
```

Our suit of armor is complete.

Programming Exercises In these exercises, you will learn how to imitate the behavior of various UNIX commands for file processing. You will need to prowl the `os` and `os.path` documentaton to solve this problems.

1. Write a program named `ls.py` which accepts a filename as a command line argument. If the file is a regular file, display the file's name. If the file is a directory, list the directory's contents. If the file does not exist or cannot be read, emit an appropriate nastygram.
2. Write a function named `p_string(filename)` which shows the permission string for a file, and which takes appropriate action if the file does not exist. Example: if a file has 644 permissions and is not a directory its permission string is `"-rw-r-r-"`. If a file is a directory and it has 711 permissions, its permission string is `"drwx-x-x"`.
3. Write a program show_sizes.py that accepts a commmand-line argument that is a file (regular or directory) and which displays the file name with its size if it is a regular file and which displays the contents and their sizes it it is a directory.
4. Write a program `copy.py` that copies a donor file to a recipient file. Give appropriate error warnings if something goes awry.
5. Open a file for reading and the invoke the `readlines` method. What does it do?
6. Write a program named wc.py that counts the number of characters, lines, and words in a file. Test it against UNIX's wc.

7. Write a program named `grep.py` that takes as arguments a string and a filename and which puts all of the lines of the file containing the specified string to `stdout`.

4 `while` and Indefinite Looping

Python has a second looping construct, `while`. Let us begin by showing an very simple example.

```
def main():
    x = input("Enter a number  ")
    x = int(x)
    while x < 100:
        print("The number you entered, {}, is less than 100".format(x))
        x = int(input("Enter a number  "))

    print("You finally entered {} and it is at least 100.".format(x))
main()
```

This is an example of a "nag" loop that will keep asking until the user does the desired thing.

Indefinite loops can be dangerous. It is very easy to have an "infinite loop," which just keeps running until the OS or the user calls the process running it to halt. Look at this little program.

```
x = 5
while x < 10:
    print(x)
    x -= 1
```

The value of `x` will keep marching farther and farther from resolution. This loop will just keep printing a countdown to the screen. If you run this, use control-C to stop it. This loop causes "spewing;" to wit, it causes great volumes of text to keep streaming into `stdout`.

Loops can also "hang;" in this event, the program simply sits there doing nothing. You can kill a hung or spewing program with control-C.

Here is a loop that is guaranteed to hang.

```
x = 1
while x > 0:
    x += 1
```

Programming Exercises Wrap these solutions in functions.

1. Write a `while` loop that prints out the entries in a list.
2. Repeatedly roll a pair of dice until you get doubles. Print the rolls as they occur. If a double 1 occurs, print `"Snake Eyes!` and if a double 6 occurs, print `"Boxcars!`
3. Repeatedly toss a coin until you get five heads in a row. return the tosses in a string like this: `"HTHHHTTTHTTTHHHTTTTHHHHH"`.

5 Programming Projects

Here you will use your skills to perform simulations of two stochastic systems. One will introduce you to `stopping times`, which entail experiments that are repeated until a specified event occurs.

The other is an analysis of risk in the Parker Brothers' game Monopoly. Everyone should love the orange monopoly.

Project 1: System Simulation for a Waiting Time In the first project, you will perform a simulation of a stochastic (random) system. In this project, you are going to perform a simulation in which you toss a fair coin until a head appears. You are going to run this experiment one million times and maintain a tally of how many times the first head came up on the first toss, second toss..., etc. An outline of suggested functions for you to implement is shown.

1. Implement the function toss_coin, which produces an "H" or a "T" at random.
2. Implement the function first_head, which tosses a fair coin until a head appears and returns the number of that trial.
3. Implement the function perform_sim(num_trials) that returns a dictionary whose keys are integers and whose values are the number of times that result was returned by repeated trials of `first_Head`. This dictionary just amouts to a tally of the trials.
4. Run this first_Head for one million trials. What do you see? What seems to happen as you double the number of trials?

Project 2: The Orange Monopoly Simulation In this next project, you are playing Monopoly and your client is on the square Just Visiting/Jail, and is just visiting. His opponent has hotels on the New York Avenue Monopoly. You have the actuarial duty of determining the price for an insurance policy to indemnify your client against the cost of the visiting the hotels for one turn.

To do this project, you will need to find an image of a Monopoly board. Here are the pertinent rules. The rent for New York Avenue is $1000. The rents for St. James and Tennessee Ave are $950. You do not need to concern yourself with the other squares, save for Go to Jail, which ends your turn and puts you in jail.

To start a turn, you roll a fair pair of dice (6-sided) and advance that number of squares. If you land on a property with a hotel, you must pay the applicable rent. If you roll doubles, you roll again and the same rules apply. If you roll doubles three times in a row, you go to straight to jail for speeding and your turn ends. In this case, you never land on the square you rolled for and are not obliged to pay rent theere.

So it's possible to have `#$!$!@#` luck and roll double threes, hit St. James, then roll a 1 and a 2 and hit New York Avenue for a total damage of $1,950.

Here are some suggestions for how to proceed.

1. Write a function that produces a tuple with two fair die rolls in it.
2. Make a tuple that represents all reachable squares in one turn.
3. Write a function that peforms the rolls in a single turn and which returns the total damage from the hotels in that turn.
4. Write a function `doTrials(n)` that accepts an integer as an argument, and which computes the average damage from the hotels in `n` trials.
5. Run a ton of trials and see what the average damage is. How would you price this policy?
6. Can you do a big trial, keep running averages in a file, and plot the results?

Project 3: Gambler's Ruin Suppose we have two gamblers and they have a total of $M (an integer) between them. They repeatedly play a game until one of them goes bust; each trial of the game has win probablility p for Player One and $1-p$ for Player Two. For $M = 10$, perform simulations of this experiment. What do the probabilities of bust look like for Player one if his initial stake is $ k, $1 < k < 10$? What is the average time to bust in each of these cases?

6 Function Flexibility

When you define a function such as this one

```
def f(x, y, z):
    return x + 2*y + 3*z
```

the arguments `x`, `y`, and `z` are called `positional arguments`. This is so because when you call the function like this

```
print(f(1,2,3))
```

the arguments are sent, in order to `f`. Notice that the position of each argument is critical. Permute them and the result is not the same.

You have seen some nifty stuff that make functions flexible and which expand their purpose, but now the time has arrived to for you to be able to use these things yourself.

Let us begin with an example. Consider the function `math.log`. This function can be called as follows.

```
>>> math.log(10)  #natural log
2.302585092994046
>>> math.log(1000, 2)
9.965784284662087 #log base 2
```

You see an optional second argument. How did they do this? We can make it happen. We will create a function called `lincoln` that does the same thing.

```
import math
def lincoln(x, b = math.e):
    return math.log(x)/math.log(b)
print(lincoln(1000,2))
print(math.log(1000, 2))
```

The second argument has a default value of `math.e`. Notice that at least one argument is required.

Here is another example. We create the illusion that this function can have as many as five arguments.

```
def product(a = 1, b = 1, c = 1, d = 1, e = 1):
    return a*b*c*d*e
print("product()  = ", product())
print("product(3)  = ", product(3))
print("product(3, 4) =", product(3, 4))
print("product(3, 4, 5) =", product(3, 4, 5))
print("product(3, 4, 5, 6) =", product(3, 4, 5, 6))
print("product(3, 4, 5, 6, 7) =", product(3, 4, 5, 6, 7))
```

If you take the default values away from the first two arguments, then at least two arguments are required to call this function. You should try this now.

End of List Rule All default arguments for any function must be grouped together at the end of the argument list. Something like this is illegal.

```
def dumb(x, y = "cows", z):
    pass
```

Run this program and see the error message. This is because these arguments are really polymorphic. If you pass a value to them, they behave as positional arguments. If you don't the default value is used. It does not take a great deal of imagination to see why the End of List Rule is necessary.

6.1 A Star is Born

Now let us consider the `print` function. You have noticed this sort of flexibility in its use.

```
    print("Foo", "Bar", "Baz", sep="moo", end="cats\n")
```

It seems that this function accepts an unlimited number of comma-separated arguments, and they allows you to specify behavior at the end of the argument list using keywords. We might like to have this particular arrow in our quiver, so let us set about getting it.

Consider the problem of finding writing a function to find the sum of a a bunch of numbers whose call looks like this.

```
>>> total(2,3,6,8)
19
```

To solve it, let us see if we can plagiarize `print`'s mechanism. Here is what `print`'s header looks like.

```
def print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False):
```

Its first argument is a *star argument* or *stargument*. A stargument must appear after all other positional arguments.

Now let's make `total`. We will use a stargument.

```
def total(*x):
    out = 0
    for k in x:
        out += k
    return out
print(total(2, 3, 6, 8))
```

You might want to require at least one argument be passed to `total`. We can enforce this by adding a postional argument at the beginning. We do this in function `totall`.

```
def total(*x):
    out = 0
    for k in x:
        out += k
    return out
def total1(y, *x):
    out = y
    for k in x:
        out += k
    return out

print(total(2,3,6, 8))
print(total1(2,3,6, 8))
print(total())
print(total1())
```

Note the opprobrious ululation after the last call. At least one number is required.

```
unix> python keywords.py
19
19
0
Traceback (most recent call last):
  File "keywords.py", line 15, in <module>
    print(total1())
TypeError: total1() missing 1 required positional argument: 'y'
unix>
```

Let us now make a simple function to count the number of files in a directory with a given extension that uses all default arguments. For defaults, we will have the directory be the `cwd` and the extension be `.txt`.

```
import os
from sys import argv
def countFiles(directory=".", end="txt"):
    files = os.listdir(directory)
    out = 0
    for item in files:
        if item.endswith("." + end):
            out += 1
    return out
folder = "." if len(argv)== 1 else argv[1]
ext = "" if len(argv) < 3 else argv[2]
print(folder)
print("There are {0} files in {1} with extension .{2}".format( countFiles(directory=folder,
```

Programming Exercise Modify the output routine so that no mention of extension is made if `argv[2]` does not exist and all files are counted.

6.2 Keyword Arguments

Now let us see how to use keyword arguments. Recall that `print`'s header looks like this.

```
def print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False):
```

Its first argument is a stargument. Following it are several keyword arguments. Each has the form `keyword=value`. Each of the specified values is the default for that argument. If no value is passed to a keyword argument, its default value is used.

Order in the court! You can have positional, star, and keyword arguments in a function. You must obey these rules

1. Positional arguments come first.
2. One stargument can come next.
3. Keyword arguments must all occur at the end.

Here is a cheesy example of these rules at work.

```
def f(*x, y="cows", z = "horses"):
    return "{}{}{}".format(sum(x), y ,z)
def g(a, b, *x, y="cows", z = "horses"):
    return "{}{}{}{}{}".format(a, b, sum(x), y, z)
print (f(2,3,4,5,y="rhinos", z="pigs"))
print (g("moo", "baa", 2,3,4,5,y="rhinos", z="pigs"))
```

7 Comprehensions

Comprehensions provide a succinct and convenient way to filter and transform collections and other iterables. These operations produce a transformed copy of the original iterable; they do not modify the original iterable. The transformed collection can be a list, set or dictionary. If you wish it to be a tuple, make it a list, then convert the result into a tuple since mutability is needed for comprehensions to work.

They can eliminate the need for a lot of error-prone looping and conditional logic. What is nice is that you just say what you want and it happens. You

have to get used to their quirky "backwardness," but this is a miniscule price to pay in exchage for this new and succinct form of expression.

Let us begin with a simple example on lists.

```
>>> x = [1,2,3,4,5]
>>> y = [k*k for k in x]
>>> y
[1, 4, 9, 16, 25]
>>> x
[1, 2, 3, 4, 5]
>>>
```

This operation took every item in the list `x` and squared it. You can use this technique to "butter" a function over a list. Suppose we have a function `f` and a list `x` and we want to call the function on every element on the list and return in in an new list. Then all you need do is this.

```
[f(k) for k in x]
```

Compare this to the loop you might use.

```
>>> x = [1,2,3,4,5]
>>> y = []
>>> for k in x:
...     y.append(k*k)
>>> y
[1, 4, 9, 16, 25]
```

You can transform tuples and range objects as well, since they are also iterables. Look at these examples. Experiment and create a few of your own.

```
>>> t = (1,2,3,4,5)
>>> items = [k*k for k in t]
>>> items
[1, 4, 9, 16, 25]
>>> sequence = [k*k for k in range(1,11)]
>>> sequence
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
>>> nums = [ord(k) for k in "abcdefghABCDEFGH"]
>>> nums
[97, 98, 99, 100, 101, 102, 103, 104, 65, 66, 67, 68, 69, 70, 71, 72]
>>>
```

You can also filter items in a list using this technique.

```
>>> names = ["smith", "jones", "sims", "boyarsky", "teague", "miller", "doyle"]
>>> has_an_e = [k for k in names if "e" in k]
>>> has_an_e
['jones', 'teague', 'miller', 'doyle']
>>>
```

This operation created a new list with the names containing the letter `e`. The general form of this construct is as follows.

```
[f(k) if k predicate(x)]
```

The item `predicate` is a boolean-valued experession involving `x`. The items that pass the filter are those for which `predicate(x)` evaluates to `True`.

Compare this to conditioanl logic where you might write the following.

```
>>> names = ["smith", "jones", "sims", "boyarsky", "teague", "miller", "doyle"]
>>> has_an_e = []
>>> for k in has_an_e:
            if "e" in k:
            has_an_e.append(k)
>>> has_an_e
['jones', 'teague', 'miller', 'doyle']
```

Two levels of indent are now obviated.


```
[f(k) if k predicate(x)]
```

The item `predicate` is a boolean-valued experession involving `x`. The items that pass the filter are those for which `predicate(x)` evaluates to `True`.

Compare this to conditioanl logic where you might write the following.

```
>>> names = ["smith", "jones", "sims", "boyarsky", "teague", "miller", "doyle"]
>>> has_an_e = []
>>> for k in has_an_e:
            if "e" in k:
            has_an_e.append(k)
>>> has_an_e
['jones', 'teague', 'miller', 'doyle']
```

Two levels of indent are now obviated.

Programming Exercises In this series of problems, you will learn about set comprehensions.

1. In an interactive session, type

   ```
   s = {k*k for k in range(-10, 11)}
   ```

   What do you see? What type of object is s?

2. The range object in the last problem has 20 elements in it. Why doesn't s?

3. Now try this.

   ```
   s = {k*k for k in range(-10, 11) if k*k > 20}
   ```

Now let's try this on dictionaries. Two tools are extremely handy. One is the `items` method for dictioanries. We demonstrate it here.

```
>>> customers = {'Jones': 4, 'Peters': 10, 'Aiken': 0, 'Morton': 3}
>>> customers.items()
dict_items([('Jones', 4), ('Peters', 10), ('Aiken', 0), ('Morton', 3)])
>>> for k in customers.items(): print(k)
...
('Jones', 4)
('Peters', 10)
('Aiken', 0)
('Morton', 3)
>>>
```

What is produced is an iterable that walks through all of the key-value pairs as tuples. Now we will find all of our idle customers.

```
>>> idle = {k: v for (k, v) in customers.items() if v == 0}
>>> idle
{'Aiken': 0}
>>> idle_names = [k for (k,v) in customers.items() if v == 0]
>>> idle_names
['Aiken']
>>>
```

One other useful method we should point out is `zip`, which can be used to create dictionaries.

```
>>> x = [1,2,3,4]
>>> y = ["I", "II", "III", "IV"]
>>> zip(x,y)
<zip object at 0x7f9ba0b34dc0>
>>> for k in zip(x,y):
...     print(k)
```

```
...
(1, 'I')
(2, 'II')
(3, 'III')
(4, 'IV')
>>> dictionary = {k:v for k,v in zip(x,y)}
>>> dictionary
{1: 'I', 2: 'II', 3: 'III', 4: 'IV'}
>>>
```

Python will hiss if you try to zip lists of unequal length. This function will also zip iterables.

```
>>> for k in zip(range(1,5), y):
...     print(k)
...
(1, 'I')
(2, 'II')
(3, 'III')
(4, 'IV')
>>>
```

Programming Exercises Opening a file gives us an iterator that walks through the file a line at a time. List comprehensions can make short work of a lot of tasks where you sift through a file.

1. Write a function `ferret(search_string, file_name)` that prints out all lines of the named file containing `search_string`.
2. Write a function that sums the number of characters residing in lines that are of a specified length or longer.

8 Generators

A *generator* is a stateful function that remembers its local symbol table between calls. You will meet a new keyword `yield` when creating genearators, which returns a value to the caller without destroying the stack frame containing the function. Between calls, this object remembers where it left off and it remembers the values of local variables. Note that the scope of these local variables is still confined to the body of the generator. Let us see this at work in a very simple example.

Each time a generator is called it can either yield a value, in which case it can be called again, or it can return a value, which termimnates its execution. We begin with a super-simple example, and we see how a generator is an iterable.

```
def simple():
    yield "quack"
    yield "moo"
    yield "baa"
    yield "neigh"
    yield "woof"
s = simple()
for k in s:
    print(k)
```

Now we run it.

```
unix> python simple.py
quack
moo
baa
neigh
woof
```

Generators can iterate through finite or infinite sets, as we shall soon see. Let's make one that starts counting at 1. Generators are a handy form of iterable.

```
def counter(n):
    k = 0
    while k <  n:
        k += 1
        yield k

for k in counter(10):
    print(k)
```

Now run it.

```
unix> python counter.py
1
2
3
4
5
6
7
8
9
10
```

Here is an imitation of `range`.

```
def strange(start=0, stop = 10, skip = 1):
    while start <  stop:
        yield start
        start += skip

for k in strange(0, 11, 2):
    print(k)
for k in strange(0, 1, .1):
    print("{0:.2f}".format(k))
```

```
unix> python strange.py
0
2
4
6
8
10
0.00
0.10
0.20
0.30
0.40
0.50
0.60
0.70
0.80
0.90
1.00
```

Generators can serve up infinite sequences. For example, we will create a generator here that serves up the nonnegative integers.

```
def z():
    out = 0
    while True:
        out += 1
        yield out
progression = z()
for k in progression:
    print(k)
```

Now we run it. Be prepared to hit control-C to stop the spewage.

```
unix> python z.py
0
1
2
...
```

```
(big number)
```

You will see that the value of the local variable `out` is rememembered between calls. This will iterate through all of the positive integers.

This beast requires some taming. You could put logic into `z` to tell it to stop, but here is a more flexible way. We pass our generator to a function that does the dirty work for us.

```
def z():
    out = 0
    while True:
        out += 1
        yield out
def stopper(generator, n):
    well = z()
    for k in well:
        if k < n:
            yield k
        else:
            return

progression = z()
for k in stopper(progression, 10):
    print(k)
```

Now we run it.

Now we run it.

```
unix> python z.py
1
```

```
2
3
4
5
6
7
8
9
```

The *stopper* routine can be recycled.

```
def z():
    out = 0
    while True:
        out += 1
        yield out
def squares():
    out = 0
    while True:
        out += 1
        yield out*out
def stopper(generator, n):
    well = generator
    for k in well:
        if k < n:
            yield k
        else:
            return

progression = squares()
for k in stopper(progression, 200):
    print(k)
```

Now we run it.

```
unix> python z.py
1
4
9
16
25
36
49
64
81
```

```
100
121
144
169
196
```

8.1 Holy Iterable, Batman!

A generator is an iterable! Here is proof.

```
def squares():
    out = 0
    while True:
        out += 1
        yield out*out
def stopper(generator, n):
    well = generator
    for k in well:
        if k < n:
            yield k
        else:
            return
s = squares()
for k in range(10):
    print(next(s))
```

Now we run it.

```
unix> python z.py
1
4
9
16
25
36
49
64
81
100
121
144
169
196
```

You can also make generators on-the-fly using a mechanism akin to a comprehension. Here is an example.

```
>>> s = (x*x*x for x in range(1,10))
>>> for k in s:
...     print(k)
...
1
8
27
64
125
216
343
512
729
```

Take note that once you do this, the generator is "used up." Watch this; we are iterating with it again.

```
>>> for k in s:
...     print(k)
...
>>>
```

The solution? Just make another one. Here is another way to use it. We will begin by reconstituting it.

```
>>> s = (x*x*x for x in range(1,10))
>>> next(s)
1
>>> next(s)
8
>>> next(s)
27
>>> next(s)
64
>>>
```

What happens if you go too far? Watch.

```
>>> next(s)
125
>>> next(s)
```

```
216
>>> next(s)
343
>>> next(s)
512
>>> next(s)
729
>>> next(s)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

This `next` function is handy if a generator generates an infinite progression. We create an example with Fibonacci numbers.

```
def fib():
    little = 1
    big = 0
    while True:
        little, big = big, little + big
        yield little
f = fib()
for k in range(10):
    print(next(f))
```

Now run this.

```
unix> python fibonacci.py
0
1
1
2
3
5
8
13
21
34
```

9 Terminology Roundup

- **definite loop** This is a loop that walks through a collection or an iterable, executing a block of code for each item. Python implements this with the `for` construct.

- **indefinite loop** This is a loop that repeats until its predicate becomes false. Python implements this with `while`
- **iterable** This is an object that can be walked through using a `for` loop. All Python collections are iterables that are walked through an item at a time. Python strings are walked through a character at a time. Generators and range objects are also iterables.
- **keyword argument** This is named argument given at the end of a function's argument list. Examples include `sep` and `end` in `print`.
- **star argument** This is an argument in a function that is preceded by a star (*), which behaves like an array inside of a function.
- **stargument** Synonym of for star argument.
- **Turing-Complete** This describes a full-featured computer language capable of solving any computational problem, given sufficient time and memory.

References