Chapter 8, Iteration and Collections

John M. Morrison

June 26, 2024

Contents

0 Introduction

We are now going to meet our second tool for doing repetitive tasks in Python. The first tool was recursion. We are now going to learn about a new way of traversing a collection, `iteration`. In this process, a block of code is executed for each item in the collection. We will learn about a new boss statement, the `for` loop, which controls iteration.

This chapter will also add two new data structures to your arsenal: dictionaries and sets. Dictionaries store key-value pairs and allow you to "index" on objects other than integers. Sets are data structures that do not admit duplicates. Both, using the magic of hashing, allow for rapid retrieval of data. You will also learn how to iterate through these.

1 Iterables and Iterators

Consider using Netflix. When you stream a movie from Netflix, what happens is your computer downloads a segment of the movie at a time by copying (buffering) that segment to its memory. As the segments play, they are discarded and new ones are loaded in the background. The original source of the movie is completely untouched. This is what allows many clients to view the same movie in different places at the same time. When you are streaming a movie, you are iterating through its frames.

An `iterator` is an object that takes a one-way walk through a collection. It is like a tube of toothpaste. Once you use an iterator it is emptied. The iterator acts like your Netflix stream. An `iterable` is an object that can hand you an iterator; you can think of the iterable as being like the file on the Netflix server; it produces an iterator on demand from a subscriber.

So far, the collections we have met are sequences, which are collections we can index into using the `[]` operator. How do we know if they are iterables?

```
>>> x = "cows"
>>> y = iter(x)
>>> y
<str_ascii_iterator object at 0x104adf160>
>>> x = [1,2,3]
>>> y = iter(x)
>>> y
<list_iterator object at 0x104ade170>
>>> x = (1,2,3)
>>> y = iter(x)
>>> y
<tuple_iterator object at 0x104ade440>
```

```
>>> x = 5
>>> y = iter(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
```

The string, list and tuple handed you an iterator. When we tried to get an integer's iterator, Python hissed. An integer is not an iterable. You can get an iterator from an iterable by calling the built-in `iter` function on it.

1.1 for

Now let's use this mechanism. This is done with the aid of the `for` boss statement.

```
>>> x = [1,2,"cow", "horse", True, 4.5]
>>> y = iter(x)
>>> for k in y: print(k)
...
1
2
cow
horse
True
4.5
```

Now let's try that again.

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

Uh oh. The tube is empty and you are out of toothpaste. An iterator can be used once, then it is exhausted. However, we can do this.

```
>>> for k in x: print(k)
...
1
2
cow
horse
True
4.5
```

You can use the `for` loop on an iterable. The iterable just hands an iterator to the `for` loop behind the scenes. You can do this repeatedly; each time a new iterator is generated for the `for` statement. For tuples and lists, the iterator walks through the collection one item at a time. Order is determined by the index; the iterator goes from 0 to the end of the collection.

What about a string's iterator?

```
>>> s = "spaghetti"
>>> for k in s:
...     print(k)
...
s
p
a
g
h
e
t
t
i
```

A string's iterator walks through a string one character at a time.

2 Home on the `range`

We are going to study the `range` function, which hands us an iterable that will walk through an arithmetic sequence of integers.

So let us open a Python3 session and create a `range` iterable.

```
>>> x = range(10)
>>> x
range(0, 10)
```

If you want to see the contents of any iterator, you can cast it as a list as follows.

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

Now let us play with the `[]` index operator.

```
>>> x[0]
0
```

```
>>> x[1]
1
>>> x[2]
2
>>>
```

Ooh, this looks like a list, In fact, you saw we can cast it to a list if you wish and see its contents. You can also index into it as you would index into a list. The range iterable yields up the integers 0-9. It then runs out of numbers and stops.

```
for k in x:
    print(f"<tr><td>{k}</td><td>{k*k}</td></tr>")
```

xx You can see that "for k in x" is an incomplete clause, as you would expect in a boss statement. The `for` statement must own a block of code. The object `k` here represents a generic item in the list of items specified by the iterator created by `x` now being streamed. It shows us a *copy* of the memory address of each item specified by `x`. The process terminates when the iterator's list runs out of elements. Now let us see the output.

```
>>> for k in x:
...     print(f"<tr><td>{k}</td><td>{k*k}</td></tr>")
...
<tr><td>0</td><td>0</td></tr>
<tr><td>1</td><td>1</td></tr>
<tr><td>2</td><td>4</td></tr>
<tr><td>3</td><td>9</td></tr>
<tr><td>4</td><td>16</td></tr>
<tr><td>5</td><td>25</td></tr>
<tr><td>6</td><td>36</td></tr>
<tr><td>7</td><td>49</td></tr>
<tr><td>8</td><td>64</td></tr>
<tr><td>9</td><td>81</td></tr>
>>>
```

Python 2 Note The `range` function in Python 2 returns a list instead of an iterable. The functionality with the `for` loop, however, is identical to that of Python 3. Make a Python 2 session right now, and you can see that for yourself. The advantage to an iterator is that it is a rule. Imagine creating `range(1000000)` in memory. You now have a million integers stored in memory. The rule for generating these entries is basically a simple little function, which has a much smaller memory footprint. For you doubting Thomases out there, here is a little demonstration.

```
>>> import sys
>>> x = range(1000000)
>>> sys.getsizeof(x)
48
>>> sys.getsizeof(list(range(1000000))
... )
8000056
>>>
```

Look at that, five orders of magnitude, a mere bagatelle.

2.1 More `range` capabilities

The call `range(n)` iterable that walks through the integers starting at 0 and ending before `n`. The `range` iterable has a couple of other guises.

The call `range(m, n)` will produce an iterator that will walk through all integers starting at `m` and ending before `n`. If `n` $\leq$ `m` the iterable returned stops immediately, except as shown below.

You can specify an optional third argument for `range` which causes the iterator to skip by numbers other than `1`. Be aware that you cannot use floating point arguments for `range`. We cast the ranges to lists, so we can see their contents.

```
>>> list(range(1,100,10))
[1, 11, 21, 31, 41, 51, 61, 71, 81, 91]
>>> list(range(100,0,-10))
[100, 90, 80, 70, 60, 50, 40, 30, 20, 10]
```

Think of range as `range(beginAt, endBefore, skip)`; you begin at `beginAt`, end before `endBefore` and skip by (well...) `skip`.

2.2 Sequences and Iterators

Every Python sequence is iterable; hand it to a `for` loop and the loop will walk through its elements. Here is a simple example.

```
>>> names = ["Alice", "Bob", "Carol", "Ted"]
>>> for k in names:
...     print(k.upper())
...
ALICE
BOB
```

```
CAROL
TED
>>>
```

The object `k` represents the object currently being shown by the list's iterator. It offers up a copy of each item in the list in the order in which they reside in the list.

How do you tell if an object is an iterable? Here is a very simple way. Just call the built-in function `iter` on it.

Programming Exercises

1. Write a function `square(ch, n)` which prints out an `n` by `n` array of the character `ch`.
2. Use a `for` loop and a range iterator to write the function `superSum(f, n)`, which returns
$$\sum_{k=1}^{n} f(k).$$

3 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 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. Place this code in a file named `starry.py`.

```python
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 positional argument at the beginning. We do this in function `total1`.

```python
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 starry.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`.

```python
import os
from sys import argv
def count_files(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)
n = count_files(directory=folder)
print(f"There are {n} files in {folder} with extension .{ext}")
```

Let's run it. First we list the contents of the directory. Then we see our program at work.

```
unix> ls
a.txt          b.txt          c.txt          cat.py         dolphin.html
    starry.py
aardvark.html bobcat.html    carical.html   count.py       echidna.html
    test.txt
unix> python count.py
/Users/morrison/book/procedural_python/recut/pn8code
There are 4 files in . with extension .txt
There are 5 files in . with extension .html
```

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.

3.1 Keyword Arguments, Again

Recall that `print`'s header looks like this.

```python
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"))
```

3.2 Spillage

Let us consider our old friend `total` in the file `starry.py`. Now make a list and pass it to `total`.

```
def total(*x):
    out = 0
    for k in x:
        out += k
    return out
print(total(2,3,6, 8))
nums = [1,2,3,4,5,6]
print(total(nums))
```

Uh oh. Punishment.

```
unix> python starry.py
19
Traceback (most recent call last):
  File "/Users/morrison/book/procedural_python/recut/pn8code/starry.py
    ", line 8, in <module>
    print(total(nums))
          ^^^^^^^^^^^
  File "/Users/morrison/book/procedural_python/recut/pn8code/starry.py
    ", line 4, in total
    out += k
TypeError: unsupported operand type(s) for +=: 'int' and 'list'
```

How can we "unpack" the elements of the list so `total` won't get dyspepsia? The answer: Our old friend *. It is also the "spill operator" that spills the contents of a list (or tuple) to a stargument. Videte et spectate! A * is prepended to `nums`.

```python
def total(*x):
    out = 0
    for k in x:
        out += k
    return out
print(total(2,3,6, 8))
nums = [1,2,3,4,5,6]
print(total(*nums))
```

Videte et spectate!

```
unix> python starry.py
19
21
```

4 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 walks through 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.

This is a guide to errors you might encounter during fileIO opearations.

`FileNotFoundError`	This is emitted if the a file you attempt to read does not exist.
`FileExistsError`	This is emitted if the a file you attempt to open exists if you open it in `"x"` mode.
`PermissionError`	This is emitted if you lack permission to access this file.

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
```

Note to Windoze Users Yoou should use a raw string when specifying a full file path to have a
interpreted as a backslash. Otherwise Python will interpret it as an escape character, causing all manner of annoying confusion. See this example.

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

Now back to our main thread. 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. Note that newline characters are present in each entry of the list that is returned.

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 can halt the process by opening the file in `"x"` mode and an error will be omitted if an existing file is opened for writing. 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 called a *buffer* , and when the ther buffer 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 buffer to be flushed into the file, where it belongs, and it discontinues the use of certain system resources.

Know when to `flush` You can cause the buffer to flush at any time manually by using 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.

5 Programming with Files

We have taken a tour of Python's file opening mechanisms in an interactive context. Now we will turn to the job of writing programs that access files. Part of the process is ensuring that our programs do not crash because an eror is emitted during execution which is not handled. There are two major means of dealing with these situations, using `os` and `os.path` to look before leaping, and Python's exception handling mechanism `try-except-finally`.

5.1 Get `with` it!

When writing programs that contain file operations, you will want to use a *context manager* and you will also want to handle exceptions, such as an attempt to access a nonexistent file or trying to write in a place where you do not have permission to write.

We being by introducting a new statement, the `with` statement. This statement manages resources for you and automatically closes a file for you on exit. It can make your code less complex. Consider the example of putting a file to the screen. We will create a file `cat.py` that does this and which takes a filename as a required command-line argument.

Let's start with an outline.

1. Require the user to specify a filename as a command-line argument. Bail if the user fails to do so.
2. Open the file for reading, and bail with an error message if the file fails to exist.
3. Put the file's contents to the screen.
4. What we won't have to do: close the file. The context manager automatically does this for us.

Now for a little code. Begin by bailing if no command-line argument is specified.

```
from sys import argv
if len(argv) == 1:
    print("Usage python cat.py filename")
    print("Bailing...")
    quit()
```

We will use the `with` statement.

```
try:
    with open(argv[1]) as fp:
        contents = fp.read()
        print(contents)
except FileNotFoundError:
    print(f"File {argv[1]} not found.")
    print("Bailing...")
except PermissionError:
    print(f"You do not have permission to access {argv[1]}.")
    print("Bailing...")
```

Programming Project: Copying Write a program named `cp` that copies a file. The usage shoud be

```
python cp.py donor recipient
```

Here are some considerations.

- The user must specify two command-line arguments for `donor` and `recipient`.
- You must have read permission for `donor`.
- If `recipient` exists, it should be clobbered; if not it should be created.

Programming Project: grepping Write a program named `grep.py` that takes two command-line arguments `search` and `file`. This program will print out all lines in the file `file` that contain the search string `search`. Check for the proper number of arguments and use exception handling where appopriate. Use the file object's iterator to walk through the file line-by-line.

Programming Project: gawking Write a program named `gawk.py` that takes command-line arguments as follows.

1. The first command-line argument is a file containing data in columns separated by whitespace.
2. The remaining command-line arguments are integers representing column numbers of the data in the datafile. Bail with error if any of these integers is larger than the number of columns in the file.
3. The program should put the selected columns to `stdout`.

Suppose you have the file data.txt that looks like this.

```
Smith Joe 30 40 25
Carson John 40 40 5
Obama Barrack 30 20 10
```

Here is a sample run.

```
unix> python gawk.py data.txt 1 4 5
Smith 40 25
Carson 40 5
Obama 20 20
```

6 Functions of Iterators

Python supplies two built-in functions for iterators, `reversed` and `enumerate`. As you might guess, `reversed` will cause an iterator to walk backwards

```
>>> for k in reversed(range(6)):
...     print(k)
...
5
4
3
2
1
0
```

For clarity and simplicity this beats

```
>>> for k in range(5, -1, -1):
...     print(k)
...
5
4
3
2
1
0
```

with a big stick. Suppose we make a list and would like it to display like this.

```
stuff[0] = foo
stuff[1] = bar
stuff[2] = baz
stuff[3] = murphy
```

One way is to do this.

```
>>> for k in range(len(stuff)):
...     print(f"stuff[{k}] = {stuff[k]}")
...
stuff[0] = foo
stuff[1] = bar
stuff[2] = baz
stuff[3] = murphy
```

This is not very Pythonic. In fact, use of `range(len(x))` is described as a "code smell." Check out this function `enumerate`.

```
>>> for k in enumerate(stuff):
...     print(f"stuff[{k[0]}] = {k[1]}")
...
stuff[0] = foo
stuff[1] = bar
stuff[2] = baz
stuff[3] = murphy
```

You might say, "Nice, but what if I want to start numbering at 1 or some other number." Watch this.

```
>>> for k in enumerate(stuff, 1):
...     print(f"{k[0]}.  {k[1]}")
...
1.  foo
2.  bar
3.  baz
4.  murphy
```

The `enumerate` function returns an iterator that walks through a collection of tuples each consisting of an item number (starting at zero by default) and each item in the collection.

Programming Exercises

1. Can you use `enumerate` to zero out a numerical list?

7 Indefinite Looping

Python provides two mechanisms for performing repetitive procedures, `for` and `while`. The `for` loop performs the same action for each item yielded by an iterator. Since an iterator knows where it stops and ends, we say that the `for` loop is a *definite* loop.

Now you will meet the `while` loop, which keeps going until its predicate is falsy. It operates an indefinite number of times, we we say that it is an *indefinite* loop. These mechanisms are a standard part of a programmer's toolbox.

The syntax for the *while* looks like this.

```
while predicate:
    block_of_code
```

Notice that `while`, like `for`, is a boss statement. The `predicate` is a predicate. The loop begins by evaluating the predicate; if it is false, `block_of_code` is ignored and the action of the loop is over; the flow of control passes just beyond the end of the loop. If the predicate is true, the loop's block executes. This occurs repeatedly until the predicate becomes false.

Since it is a boss statement, it is an error to have a `while` statement without a block of code. At minimum a `pass` statement must be present. Make sure you indent all statements below the `while` statement that you want to run repeatedly.

Here is a simple program with a `while` loop in it.

```
#!/usr/bin/python
k = 10
while k > 0:
    print(k)
    k -= 1
print "Blastoff!"
```

Run this program. Once you do this, remove the statement `k -= 1`. You will see the loop get stuck. Hit Control-C to hang up and stop it. This is a common logic error when writing a loop. If the predicate never becomes false, the loop will contentedly run (and possibly do nothing) until you force the program to quit. This results in two common types of behavior. Programs can *hang*, in which the program sits there and appears to do nothing, or they can *spew*, in which the program keeps doing the same thing over and over until it is forced to quit. Spewing is gets its name because it often results in great amounts of text being put to the screen.

Here is another use case for indefinite looping, nagging a user until the user enters something valid.

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

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

Another use case is performing a *stochastic simulation*, in which you carry out a random experiment programmatically. We will do a simple example here, tossing a fair coin until a head appears, and count how many tosses it takes. First, we do the coin toss.

```
import random
```

```python
def toss_coin():
    return random.choice("H", "T")
```

Now here is a key idea. You should realize that "until" is the same as `while not`. So now let us write our function to do the experiment.

```python
import random
def toss_coin():
    return random.choice("H", "T")
def until_a_head():
    count = 1
    while toss_coin() != "H":  #while not a head
        count += 1
    return count
```

One thing that is guaranteed to be true is that the `while`'s predicate is `False` when the loop terminates. So, that last toss is guaranteed to be a head.

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: `"HTHHHTTTHTTTHHHTTTTTHHHHH"`.