Unit 4

Learn something new

1. Once we've developed useful functions like `average` and `cleanedup`, we naturally want to reuse them. So far we've been doing that just by copying and pasting the function definitions into a new program file. But since this kind of reuse is so common and important, Python provides a much better method.

   Create a program file called `my.py` containing exactly the following:

```
def cleanedup(s):
   alphabet = 'abcdefghijklmnopqrstuvwxyz'
   cleantext = ''
   for character in s.lower():
      if character in alphabet:
         cleantext += character
      else:
         cleantext += ' '
   return cleantext

def average(numbers):
   total = 0
   for number in numbers:
      total += number
   return total/len(numbers)
```

   Create a second program file using any name you like (but ending in `.py` as usual) and edit it as follows:

```
import my

print(my.average([1,2,3,4]))
print(my.cleanedup('This--WOW--is ready for split()'))
```

   Be sure this file and `my.py` are saved in the same folder. Then run the second file. You should see the following output, showing that we've used the functions `average` and `cleanedup` without having to copy their definitions into the new program file.

```
2.5
this  wow  is ready for split
```

The my.py file is called a **module**. It is prepackaged code meant to be used by other programs. To use a module, we employ an import statement. This executes the instructions in the module and makes available any names assigned by these instructions. Typically, as in this case, some of these are names of functions.

Note carefully two essential details.

1.1 In the import statement, we do not use the .py part of the filename.

1.2 When we use a name defined in a module, we need to add the module name and a dot at the beginning. For example, we refer to the function average imported from the module my as my.average.

As we proceed, we can add new functions to my.py, making it a handy collection of useful tools we've developed. Then we can get the use of the whole collection in a new program with just one line: import my. After we've been working for some time, we might even want to have several different modules, each with a related set of functions. Then we can import just the ones we need for a given project.

If we build up a particularly useful module, we can also give a copy to a friend or colleague, making their programming easier as well.

2. Of course, the really good news is that we can *get* prepackaged code as easily as we can share ours. In fact, Python comes with a huge number of handy modules. Try the following:

```
import random

colors = ['red', 'blue', 'green', 'yellow', 'orange']
print(random.choice(colors))

random.shuffle(colors)
print(colors)
```

The random module defines many useful functions for the computer equivalent of doing something 'at random'. The choice function takes in a collection of objects and randomly chooses one. The shuffle function takes a collection and mixes it up into a random order. If you run this program several times, you'll get a different output each time—one of the five given colors and then a list of all five in a random order.

3. We can use these new functions to answer some interesting questions. Suppose we play a betting game. We start out with 10 dollars and flip a coin over and over. Each time it comes up heads we win a dollar; each time it comes up tails we lose a dollar.

We keep playing until either we lose all of our money or double our initial bankroll, ending up with 20 dollars. On average, how many flips will this game take?

Let's start by writing a program that plays the game once.

```
import random

countFlips = 0
initial = 10
bankroll = initial
while 0 < bankroll < 2*initial:
   flip = random.choice(['heads', 'tails'])
   countFlips += 1
   if flip == 'heads':
      bankroll += 1
   else:
      bankroll -= 1

print(countFlips)
```

The statements under the `while` carry out one flip of the coin. We use the `random` module's `choice` function to choose at random between two ways the coin toss might come out. We increase `countFlips` by one to show that one more toss has been made. And we either add or subtract a dollar from our bankroll, depending on the result of the flip.

The condition of the `while` is designed to keep the program flipping so long as we haven't either lost all of our money or doubled our initial bankroll. Since `initial` refers to 10—the amount we started with—the value of `2*initial` is `20`. So the condition boils down to `0 < bankroll < 20`, that is 'bankroll is between 0 and 20'. If we keep flipping *while our bankroll is between 0 and 20*, we'll stop only when it finally gets to either 0 or 20.

Run this program several times. You'll find that the number of flips it takes to get to either 0 or 20 is quite variable. Sometimes it takes only 15 to 20 flips, other times it may take 130 or more. So we still don't know, on average, how long the game lasts.

What we need to do is play a large number of games automatically and then average the results. As a first step, let's turn the code we've already written into a function that, when called, will give the number of flips it takes to play one game.

Here's the revised code, along with a quick test that calls the function three times

```
import random
```

```
def oneGame(initial):
    countFlips = 0
    bankroll = initial
    while 0 < bankroll < 2*initial:
        flip = random.choice(['heads', 'tails'])
        countFlips += 1
        if flip == 'heads':
            bankroll += 1
        else:
            bankroll -= 1
    return countFlips

print(oneGame(10))
print(oneGame(10))
print(oneGame(10))
```

The only things that have changed are (a) we pass in the starting amount (e.g. 10 dollars) rather than setting it within the function and (b) we return the number of flips the game lasts rather than printing it. Because the starting amount is passed to the function, we can use this same function to see what happens if we start with 20 or 50 dollars or any other amount rather than just 10.

Suppose now that we'd like to call the `oneGame` function a large number of times, say 1,000 times. We've seen how to write a `while` statement that accomplishes this, but it's a little complicated. We also know that we could do the job with a simple `for` like this:

```
for number in [1, 2, 3, ..., 1000]:
```

The only problem with this approach is that we don't want to have to create a list of 1,000 numbers by typing it out. Luckily, since this is a common situation, Python provides an easy solution. Here's how the revised code looks.

```
for number in range(1000):
```

The `range` function returns something very nearly like the list of numbers we want. One slight difference is that it counts from 0 rather than 1. So the items we'll go through if we write a `for` with `range(5)`—for example—are 0, 1, 2, 3, and 4. Notice that there are still five items, so we'll still execute the body of the `for` statement five times. We'll talk about another difference between `range(1000)` and the list `[1, 2, 3, ..., 1000]` later in this unit, but it doesn't affect us now.

Using `range`, we can easily write code that calls `oneGame` 1,000 times and gives the average number of flips.

```
import random

def oneGame(initial):
    countFlips = 0
    bankroll = initial
    while 0 < bankroll < 2*initial:
        flip = random.choice(['heads', 'tails'])
        countFlips += 1
        if flip == 'heads':
            bankroll += 1
        else:
            bankroll -= 1
    return countFlips

totalFlips = 0
for number in range(1000):
    totalFlips += oneGame(10)

print('Average number of flips:', totalFlips/1000)
```

If you run this program, you'll get a good answer to the original question. It takes about 100 flips on average to go broke or double your money, if you start with 10 dollars. At this point, though, it's also very easy to find out what happens if you start out with a different amount of money.

Let's move our experiment-running code into a function and have the function carry out a *series* of experiments. We'll give the function a list of different amounts of money to start with and how many games to try for each starting amount. Then it will carry out all the experiments and report back on the results. Here's our final program.

```
import random

def oneGame(initial):
    countFlips = 0
    bankroll = initial
    while 0 < bankroll < 2*initial:
        flip = random.choice(['heads', 'tails'])
        countFlips += 1
        if flip == 'heads':
            bankroll += 1
        else:
            bankroll -= 1
    return countFlips
```

```
def experiment(initials, repetitions):
  for initial in initials:
    print('Initial bankroll:', initial)
    totalFlips = 0
    for number in range(repetitions):
      totalFlips += oneGame(initial)
    print('Average number of flips:', totalFlips/repetitions)
    print()

experiment([10, 20, 40], 2000)
```

The first thing to notice about this code is that the main work of the program is specified in just one line, the last one. Here we call on the `experiment` function to try 2,000 games each for three different starting amounts: 10 dollars, 20 dollars and 40 dollars.

Next, note that we have written the `experiment` function so that takes in two items of information. Actually, we can write functions that take in as many items as we like. What happens when we pass several items is simple. In our case, the first item passed is the list `[10, 20, 40]` and the second is the integer `2000`. These are matched with the names given in the first line of the function definition. When we make the function call, the interpreter starts by executing two assignments: `initials = [10, 20, 40]` and `repetitions = 2000`. Then it continues with the statements in the function body.

In general, assignments in a function call are made *by position*: the first name in the definition is assigned to the first item passed in the call, the second name in the definition is assigned to the second item in the call and so on.

The rest of `experiment` is nearly identical to what we had before we made it into a function. For each starting amount, the function calls `oneGame` a set number of times (the number passed in as `repetitions`) and reports the average result. Of course, when we call `oneGame`, we pass it `initial`, which is the number of dollars to start with.

When you run this program, you should notice a definite—and surprisingly simple—relationship between the initial bankroll and the average number of flips.

4. Here's another tricky question. Suppose a teacher collects exam papers from a class of students and then gives them back out at random, so that the students can grade each other's work. What's the chance that someone will end up grading his or her own paper? We'll look at this first for a class of 30 students, but when we're done it will be interesting to see what happens if the size of the class gets larger. For example, what's the chance that some student will get his or her own paper in a class of 300 students?

We'll start by defining and testing a function for giving the papers out one time. The function will return '**warning**' if any student gets his or her own paper and '**okay**' if not.

```
import random

def paperStatus(classSize):
   papers = list(range(classSize))
   random.shuffle(papers)
   for student in range(classSize):
      if papers[student] == student:
         return 'warning'
   return 'okay'

print(paperStatus(30))
print(paperStatus(30))
print(paperStatus(30))
```

This function is short, but there are a number of points to cover before it will be possible to understand how it works. The first is that, as we said before, the **range** function does not quite return a list of numbers. If it did, then a statement like

```
for number in range(10000000):
```

would require your computer to store a list containing ten million numbers. This would be a waste of memory, since the program only actually *uses* one number at a time. To save memory, **range** actually returns a **generator**, a technical term for an object that produces a series of values one at a time. When **range** is used in the header line of a **for** statement—e.g. **for number in range(10000000)**—this is exactly what we want. One value is produced each time **number** is assigned and no memory is wasted storing all the other values that are not in use.

Sometimes, though, we actually want to produce a list of numbers. In this case, we need a conversion like the one we used to change a string like '**123**' into the integer **123**. Here, the conversion function is called **list**. We pass it something that can be used to create a list and it returns one. Here are two examples:

```
print(list(range(5)))
print(list('abcde'))
```

The output looks like this:

```
[0, 1, 2, 3, 4]
```

```
['a', 'b', 'c', 'd', 'e']
```

Now we can understand the beginning of the `paperStatus` function. Assuming that `classSize` is 30, the line

```
papers = list(range(classSize))
```

sets `papers` to `[0, 1, 2, ..., 29]` by converting the generator returned by `range` to an actual list. The next line

```
random.shuffle(papers)
```

rearranges the list at random. It might end up like this: `[17, 0, 8, 3, ..., 14]`.

Our idea is that we'll refer to both students and their papers by number. We'll call the first one 0, the next one 1 and so on—up to 29 if there are 30 students in the class. Before the teacher collects the papers, the situation is like this—with Paper 0 in the hands of Student 0, Paper 1 in the hands of Student 1 and so on.

Paper 0	Paper 1	Paper 2	Paper 3	…	Paper 29
Student 0	Student 1	Student 2	Student 3	…	Student 29

After the papers are collected and returned, the situation might look like this.

Paper 17	Paper 0	Paper 8	Paper 3	…	Paper 14
Student 0	Student 1	Student 2	Student 3	…	Student 29

The list of numbers referred to by `papers`—e.g. `[17, 0, 8, 3, ..., 14]`—simply represents the top row in this diagram.

In short, the first two lines of the `paperStatus` function take care of redistributing the papers at random. What remains is to check if anyone got his or her own paper. To do that we'll go one by one through the students—that is the numbers 0 through 29—checking in each case if the associated paper number is the same as the student number.

We've seen that entries in a dictionary can be accessed using keys. In the same way, items in a list can be accessed by **index**, the numerical position of the item, counting the first position as 0, the next as 1 and so on. If `papers` is the list `[17, 0, 8, 3, ..., 14]`, then `papers[0]` is `17`, `papers[1]` is `0` and so on, up through `papers[29]` which is `14`. In this example, the key fact is that `papers[3]` is `3`, showing that Student 3 got his or her own paper.

Once `papers` has been shuffled, the following code does the checking we want:

```
    for student in range(classSize):
      if papers[student] == student:
        return 'warning'
    return 'okay'
```

This checks if `papers[0]`—the paper given back to Student 0—is `0`; then if `papers[1]` is `1` and so on. If a match is found, the function `paperStatus` immediately stops—without finishing the `for` statement—and returns the answer '`warning`'. If the `for` statement completes, it must be because no match was found. In this case, `paperStatus` returns '`okay`'.

5. The function `paperStatus` gives papers out once and gives a 'warning' if any student will be grading his or her own. Now let's write a function that runs a series of experiments—in which `paperStatus` is called many times—and summarizes the results. Here is a complete program that will tell us the number of times we get a warning for classes of 30, 300 and 3,000 students. For each class size, we'll give out papers 1,000 times.

```
import random

def paperStatus(classSize):
  papers = list(range(classSize))
  random.shuffle(papers)
  for student in range(classSize):
    if papers[student] == student:
      return 'warning'
  return 'okay'

def experiment(classSizes, repetitions):
  for classSize in classSizes:
    print('Class size: ', classSize)
    warnings = 0
    for number in range(repetitions):
      if paperStatus(classSize) == 'warning':
        warnings += 1
    print('Warnings:', warnings, 'out of', repetitions)
    print()

experiment([30, 300, 3000], 1000)
```

The `experiment` function uses nothing new and is very similar to the analogous function we wrote earlier in this unit.

Run the program. You may be surprised to find out what happens to the number of warnings as the size of the class increases.