Unit 2

Learn something new

1. Try the following program. The first print statement uses a **function** called `len` to find out the *length* of the string referred to by `line`. The output is 30, because there are 30 characters in the string; note that letters, spaces and punctuation all count as characters. A function is a prepackaged set of instructions; in this case it's one built in to Python to determine how many items are in a collection.

```
line = 'This is a sample line of text.'
print(len(line))
print(line.split())
print(len(line.split()))
```

   The second print statement uses a function called `split`. It divides a string into parts, breaking it at every series of space characters or other characters like tabs that display as blanks. The result—as you can see from the output you get by running the program—is a list of smaller strings. The `split` function is extremely handy when we want to process words within a long string.

   Notice that both `len` and `split` are followed by parentheses. When we **call** a function—use it—we pass it the information it needs to carry out its work by placing the information inside these parentheses. In the first print statement, we pass `line` to `len`. The `len` function computes the result and **returns** its answer, 30. This answer essentially replaces the function call. Writing `print(len(line))` has exactly the same effect here as writing `print(30)`.

   Note that `print` is also a function. In the same statement, we pass `len(line)`, or 30, to `print`. The `print` function requires this information to carry out its job; it needs to know what to display.

   In the last line of the program, we call the `split` function on `line`. The `split` function returns a list of words, which in turn is passed to `len`. The `len` function counts the number of items in the list and returns the answer, 7, passing this number to `print` which displays it. As a result, we find out how many words are in the sample text.

   One last point to note is that we call the `split` function by writing `line.split()`, not `split(line)`. The reason is that `split` is not a general function like `len` that works on many kinds of objects—lists, strings, etc. Instead it is specifically associated with strings. We say it is a string **method**. To call a string method, we connect it to any existing string with a dot (.). Strings, lists and other **data types** in Python each come with an extensive set of methods to carry out tasks naturally associated with the kind of data they store.

2. Integers are one data type we have already seen and this data type comes with the methods you would expect for numbers: addition, multiplication, etc. We can use these methods using the dot notation just introduced, but since they're so common, Python provides a much easier way. Try the following:

```
print(2+3)
print(71*(5-2))
print(36/3)
print(2**1000)
```

This displays 2 plus 3, 71 times 3, 36 divided by 3 and 2^{1000} (that's 2 times itself 1,000 times). The last answer is 302 digits long, but no problem for Python. Python follows the standard rules you learned in math class, doing multiplications and divisions before additions and subtractions; we can use parentheses, as in the second line, to force a different order if we like.

If we use names for numbers, we can sometimes make calculations clearer.

```
numberOfBoxes = 4
itemsPerBox = 20
print('Total items:', numberOfBoxes * itemsPerBox)
```

3. Now try the following.

```
x = 4
print(x*2)
print(x)
x = x*2
print(x)
```

The first line assigns the name `x` to the integer 4. The next line calculates 4 times 2 and displays the answer, 8. Note carefully that `x` is still a name for 4, as the second print demonstrates. *Using* a name does not change what it refers to.

If we want `x` to refer to a new value, we have to place it on the left of the equals sign that Python uses for assignment. The fourth line first *uses* the current value of `x` and computes 4 times 2. It then *reassigns* `x` to refer to the answer, 8.

This is a good time to stress again that the equals sign means something very different in Python than in a math class. In algebra, `x=x*2` is a statement that `x` has the same

value as a number twice as large. This means `x` must be zero. Writing the same thing in Python means to carry out a calculation using the current value of `x` and then to let `x` refer to the answer. It certainly does not tell us that `x` must be zero.[1]

4. Reassigning a name based on its current value is such a common operation that Python provides a shortcut. Rather than writing `x = x*2` to assign `x` to a value twice as large as its current one, we can write just `x *= 2`. Likewise, if we want to reassign `x` to a value one larger, we can write `x = x+1`, but Python allows us to shorten this to `x += 1`.

5. Let's use what we know to analyze some lines of text. We can write much more useful programs if we take our text from a file rather than from strings typed in by hand. The file `pap.txt` contains the complete novel *Pride and Prejudice* by Jane Austen. Make sure this file is in the same folder as your program file and then run the following program.

```
with open('pap.txt') as book:
  for line in book:
    if 'property' in line:
      print(line)
```

   The output is a series of eight lines, all the lines in the file `pap.txt` that include the word 'property'. The first line creates a **file object** associated with `pap.txt` and allows us to refer to it with the name `book`. Like a list or string, a file object can be used as the ordered collection in a `for` statement. The simplest way, as in our example, assigns a name (like `line`) one by one to a series of strings, one for each line of the file. The rest of this program uses nothing new. It checks if the current line contains the word 'property' and, if so, displays it.

   The `with` statement creates a connection with a file, maintains it while the indented instructions are carried out and then breaks it. We indent only statements that need to use the file.

6. Next, we'll run through the same file and count the total number of words used. Here's the program. When you run it, the output should be 121555.

---

[1] If you've programmed in a language like C++ or Java, you should note that Python assignment is also quite different from what you're used to. In one of these languages, a name like `x` refers to a location in memory where something is stored. When we write `x=x*2`, the `x` goes on referring to the same memory location, but the contents are changed to a new value twice as large. In Python, the same statement causes `x` to refer to a new object in an entirely different part of memory. If the original object referred to by `x` is no longer needed for other purposes, the system reclaims the memory for reuse. As a result, we can write `x=4` and then, later, `x='hello'`. This would be illegal in C++ or Java, since the memory referred to originally is just large enough for an integer and cannot hold a string.

```
count = 0
with open('pap.txt') as book:
   for line in book:
      count += len(line.split())
print('Word count:', count)
```

We use the name **count** to refer to the total number of words seen so far. Before we look at the file, it starts at zero. After creating a file object, we run through the lines one by one. For each line, we use **split** to get a list of individual words, pass the result to **len** to find out how many words are in the list and update **count**, increasing it by this value. For example, if we have seen 900 words so far and the current line is the string 'just an example', then **line.split** will return **['just', 'an', 'example']**, meaning that **len(line.split())** will return 3. The effect is as if we had written **count = count+3**, reassigning **count** to the value 903.[2]

7. With a slight variation, we can see how many times any particular word is used in the book. Here's a program that counts occurrences of the word 'the'. Run it and you'll find that the file contains this word 4,047 times.

```
count = 0
with open('pap.txt') as book:
   for line in book:
      for word in line.split():
         if word == 'the':
            count += 1
print("Number of times 'the' is used:", count)
```

This program uses just two new techniques. First, to check if two values are the same, we use **==** in Python. When you see this in a program, you should read it as *equals*. So the **if** statement here says: *if* **word** *equals* **'the'***, the value referred to by* **count** *should increase by one*. Again, **=** means something completely different in Python and you should not say *equals* when you read it.

The second new point is the use of double quotes (") in the last line. We normally enclose a string with single quotes ('), but here that would cause a problem since we use single quotes within the string.

The program itself is easy to understand. We go one by one through the lines in the file. Within each line, we go one by one through the words. Each time we find the word 'the', we count up by one.

[2] The actual number of words is slightly different than what this program reports, because we've defined words to be just strings of characters surrounded by blanks. This means that in the string 'The man--dropping his gun--ran out the door', the substring 'gun--ran' will be counted as one word.

Note that the following will *not* work.

```
count = 0
with open('pap.txt') as book:
   for line in book:
      if 'the' in line:
         count += 1
print("Number of times 'the' is used:", count)
```

This program has two errors. First, no matter how many times 'the' appears in a line, count will be increased only once. Second, count will be increased incorrectly if the word 'then' or 'other' is included in a line, since each of these contains the word 'the'.[3]

8. Finally, here's a program that finds the line with most words in it. The only thing technically new here is the use of the symbol > to mean 'greater than'. We can also write < for 'less than' and >= and <= for 'greater than or equal' and 'less than or equal.'

   More important, though, is the use of a standard programming technique. As we proceed, we keep track of the most extreme example so far—in our case the line we've seen with the most words. We compare each new example to the one we're saving and update only when we have a new champion.

```
maxcount = 0
with open('pap.txt') as book:
   for line in book:
      count = len(line.split())
      if count>maxcount:
         maxline = line
         maxcount = count
print(maxline)
```

   Here **maxcount** keeps track of the number of words in the longest line encountered at any time. Naturally, it's zero before we begin to look at any lines. For each new line, we let **count** refer to the number of words it contains. If **count** is bigger than **maxcount**, we have a new winner. In this case, we remember the line and the new maximum number of words using as **maxline** and **maxcount**.

[3] Our original version has some problems too. It will not count occurrences of 'The' with a capital T or occurrences followed by a punctuation mark: '…only the--extremely patient--women understand….'

9. By now, we've used `for` statements to go character by character, word by word and line by line. How does Python know which we mean? The answer is simple, but it's worth thinking through carefully, because misunderstanding this point is one of the most common causes of errors.

   When we write a `for` statement, Python looks at the target—the last thing on the header line—to see what sort of object it is. It then uses the type of the target to decide how to go one by one through it.

   If the target is a *list*, `for` will go item by item through the items that make up the list. So this code

```
target = ['here', 'are', 'four', 'words']
for item in target:
   print(item)
```

   will yield the output

```
here
are
four
words
```

   When the target is a *string*, Python automatically goes character by character through the string. So this code

```
target = 'here are four words'
for item in target:
   print(item)
```

   will yield the output

```
h
e
r
e

a
r
e

f
```

```
o
u
r

w
o
r
d
s
```

The split function breaks up a string into a list of strings. So if we have

```
target = 'here are four words'
```

then target.split() is a list that looks like this:

```
['here', 'are', 'four', 'words']
```

And when we use target.split() as the target in a for statement, Python will go one by one through the items in this list, that is word by word through the words in the original string.

This code

```
target = 'here are four words'
for item in target.split():
   print(item)
```

will yield the output

```
here
are
four
words
```

This is exactly the same as in our first example. In either case, the for statement is going through a *list* item by item.

Now what happens when the target of a for statement is a *file object*? In this case, Python automatically goes one by one through the *lines* in the file.

Suppose we have a file called sample.txt, which consists of the following two lines:

```
here are four words
Fred Ted Ned
```

Then this code

```
with open('sample.txt') as file:
   for item in file:
      print(item)
```

will set item one by one to each of the lines in the file. The output looks like this:

```
here are four words
Fred Ted Ned
```

Each line taken from the file is a string. For example, when this last program is working with the first line of the file, it is exactly as if we had written

```
item = 'here are four words'
```

Since item is a string, we can use it as one. In particular, we can go through it letter by letter—by using it as a target—or we can use split to break it up into a list of words and go through those. Here's code to go one by one through the lines in the file and, for each line, go character by character:

```
with open('sample.txt') as file:
   for item in file:            # item is a string, holding
      for character in item:    # a full line from the file
         print(character)
```

Here's the output:

```
h
e
r
e

a
r
e
```

```
f
o
u
r

w
o
r
d
s


F
r
e
d

T
e
d

N
e
d
```

And here's code to go word by word:

```
with open('sample.txt') as file:
   for item in file:             # item is a string
      for word in item.split():    # item.split() is a list
         print(word)
```

Here's the output:

```
here
are
four
words
Fred
Ted
Ned
```