Unit 3

Learn something new

1. Here's a program that introduces a new data type, the **dictionary**. A dictionary is a collection of **key-value pairs**. In this particular dictionary—created on the first line of the program—there are three pairs. The first key is `'smith'` and it is associated with the value `'apple'`. Note that we use colons (:) to connect each key with a value and commas (,) to separate key-value pairs. The whole dictionary is set off by curly brackets: `{` and `}`.

   ```
   passwords = {'smith':'apple', 'jones':'a34xx', 'brown':'zzzz'}
   username = input('username: ')
   password = input('Password: ')
   if password == passwords[username]:
      print('You are logged in.')
   else:
      print('Bad password.')
   ```

   To look up a value in a dictionary, we place a key in square brackets after the dictionary's name. So, for example, if we write `passwords['smith']` the value will be `'apple'`.

   Our program asks the user to type in a name and password. It then checks the given password against one it looks up for the user in the `passwords` dictionary. If they match, it tells the user he or she is logged in; otherwise it reports a password failure.

   This is our first use of an expanded form of the `if` statement. As usual, it starts out with a condition followed by an indented block of statements to carry out if the condition holds. But now it also has an `else` part with an indented block of statements to carry out if the condition does *not* hold. With an `if-else`, one of the two blocks is certain to be executed, the `if` block if the condition is satisfied and the `else` block otherwise.

2. Now let's build up a dictionary that allows us to quickly look up which lines in the *Pride and Prejudice* file contain any given word. The keys will be words, like `'property'` and the associated value for each will be a list of line numbers, like this: `[15, 466, 901, 2647]`. If we were creating this dictionary by hand, we might start like this:

   ```
   concordance = {'property':[ 15, 466, 901, 2647], …}
   ```

Naturally, though, we'll create our concordance automatically.[1] Here's a program to do it.

```
concordance = {}
with open('pap.txt') as book:
   linenum = 1
   for line in book:
      for word in line.split():
         if word in concordance:
            concordance[word].append(linenum)
         else:
            concordance[word] = [linenum]
      linenum += 1
print('Test:', concordance['property'])
```

Temporarily ignoring the `if-else` in the middle of this program, we can see that the rest is easy to understand. The first line creates an empty dictionary called `concordance`. Then we create the usual file object called `book`, run through it line by line and, within each line, go word by word.

As we work, the name `linenum` always refers to the number of the current line. When we start, we assign it to the number 1 and each time we finish with a line we let it refer to a number one higher.

The last line is just a test to see if the concordance was built correctly. It looks up the word `'property'` in the dictionary and prints the associated value, which should be a list of line numbers where the word occurs.

Now let's look at the `if-else` statement where the real work is carried out. When we encounter a word on a certain line, say 'apple' on line 173, what do we need to do? The answer is that it depends on whether this is the first time we've seen this word.

If it *is* the first time we have encountered the word, we need to create a new entry in the dictionary with `'apple'` as the key and the single-item list `[173]` as the value. The statement we want is: `concordance[word] = [linenum]`. In general, we add a new key-value pair to a dictionary by writing: `dictionaryname[key] = value`.

Note carefully that the value we're adding is a list. If we write `concordance['apple'] = 173`, then the value is the integer 173. If instead we write `concordance['apple'] = [173]`, then the value is a list which contains one item, the integer 173.

[1] The word 'concordance' means a reference work that tells us where we can find words or phrases in a text or collection of texts, e.g. the bible, the works of Shakespeare, etc.

Now consider what we need to do if we encounter a word that has appeared one or more times earlier in the book. In this case, `concordance[word]` already has a value—it is the list of line numbers where the word has been found so far. In this case, we just want to add a new line number to the list. We use the `append` list method to add the new item: `concordance[word].append(linenum)`.

The `if-else` distinguishes between the two cases with the following condition: `word in concordance`. This is a new use of `in`, but a natural one; the condition `item in dictionary` holds if the item can be looked up in the dictionary, i.e. if it is used as a *key*. No checking is done to see if `item` appears as a value.

To sum up, the program goes word by word through each line of the file. For each word, it either adds a new dictionary entry or expands an existing entry to include a new line number.

3. It's a bit of a waste to build an entire concordance and then use it to check just one word. Let's modify our program slightly so that once the concordance is compiled we can use it as much as we like.

   The only change is at the end. We'll *repeatedly* ask the user for a word to look up and then report back on what we find in the concordance. So far the only way we have of repeating instructions is to indent them within a `for` statement, but this limits the repetition to once for each item in a collection. Here, we'd like to allow the user to look up any arbitrary number of words, so we'll introduce a new statement: `while`. Here is the revised program.

```
concordance = {}

with open('pap.txt') as book:
    linenum = 1
    for line in book:
        for word in line.split():
            if word in concordance:
                concordance[word].append(linenum)
            else:
                concordance[word]=[linenum]
        linenum += 1

while True:
    word = input('Enter word: ')
    if word in concordance:
        print('Found on lines:', concordance[word])
    else:
        print('Not found.')
```

Like the **if** statement, **while** specifies a condition. The indented statement block that follows is carried out over and over, *while* the condition holds. As soon as it fails to hold, the interpreter skips the indented block and continues with the next statement following the **while**.

In our case, we want to ensure that the condition does *not* fail, so that the user can always enter another word. We could use any condition that is permanently true—e.g. 2<3—but there's a simpler way. Just as an expression like 2+3 **evaluates** to 5, the logical expression 2<3 evaluates in Python to True. If we use True as the condition in a **while**, we get an infinite loop, a repetition that never stops. Each time through the loop, the user enters a word and gets the appropriate report.

Of course, it *is* possible to stop the program. Just type Ctrl+C in the interpreter window.

4. If you run the concordance program and check the word 'property', the program will report that it is found on four lines. This is odd, since back in Unit 2 we found it eight times. To see what is happening, consider the following output from our program.

```
Enter word: property
Found on lines: [15, 466, 901, 2647]
Enter word: Property
Not found.
Enter word: property.
Found on lines: [2277, 2797]
Enter word: property,
Found on lines: [12119]
```

Recall that the **split** function assumes blank space should be used to delimit words. This means that punctuation will be included as part of a word. As things stand, our concordance has one entry for '**property**' another for '**property.**' and still another for '**property,**' . Moreover, the string '**Property**', with a capital P, matches none of these.

Let's take care of all these problems by cleaning up each line of text before we process it. First, we'll convert all letters to lowercase. Then we'll make up a new line that has a space substituted for any character in the original that is not a letter.

Here is the revised **code**, the text that makes up a program:

```
concordance = {}
alphabet = 'abcdefghijklmnopqrstuvwxyz'

with open('pap.txt') as book:
  linenum = 1
  for line in book:
    cleanline = ''
    for character in line.lower():
      if character in alphabet:
        cleanline += character
      else:
        cleanline += ' '
    for word in cleanline.split():
      if word in concordance:
        concordance[word].append(linenum)
      else:
        concordance[word]=[linenum]
    linenum += 1

while True:
  word = input('Enter word: ')
  if word in concordance:
    print('Found on lines:', concordance[word])
  else:
    print('Not found.')
```

Apart from the definition of the string called `alphabet`, all of the changes are contained in a few lines under the first `for`. We use the string method `lower` to make a copy of the current line with uppercase letters converted to lowercase. Most of the work, though, is done by creating a new empty string called `cleanline` and then adding characters to the end of it one by one. We go through each of the existing characters, adding it to `cleanline` if it is a letter and adding a space instead if it is not.

Think carefully about the space character highlighted in green in the code above. A very common mistake is to omit it and use '' in place of ' '. But if we replace non-alphabet characters with nothing rather than with a space, '`sample--with, dashes and commas`' will become '`samplewithdashesandcommas`' rather than what we want, which is '`sample  with  dashes and commas`'. Without spaces, the whole line will look like one long word and the `split` method will have no effect.

Note also that the symbol `+`, which signifies addition for integers, means something very different for strings. The technical term is **concatenation**, but you won't be wrong if you think of it just as 'sticking together'. In Python, '`base`'`+`'`ball`' evaluates to '`baseball`'. And if `word` refers to the string '`baseball`', then after the instruction `word += 's'`, it will refer to the string '`baseballs`'.

If you try the latest version of the concordance program, you'll see that it now correctly finds all eight occurrences of 'property'.

5. Cleaning up each line of text before processing makes our program work better, but it also makes it rather hard to understand. It's confusing to have the code for clean-up

right in the middle of the concordance-building. A much better plan would be the following. Don't try to run this now, though; it doesn't yet work.

```
concordance = {}
with open('pap.txt') as book:
   linenum = 1
   for line in book:
      for word in cleanedup(line).split():
         if word in concordance:
            concordance[word].append(linenum)
         else:
            concordance[word]=[linenum]
      linenum += 1
while True:
   word = input('Enter word: ')
   if word in concordance:
      print('Found on lines:', concordance[word])
   else:
      print('Not found.')
```

Here we've called on a *function* named `cleanedup` to do all the work of converting to lowercase and replacing non-alphabetic characters with spaces. This certainly makes the program neater and easier to follow. The problem, of course, is that such a function does not exist.

Luckily, when a function is not already part of Python, we can create our own. Here's a final revision of the concordance program that *does* work.

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

concordance = {}
with open('pap.txt') as book:
   linenum = 1
   for line in book:
      for word in cleanedup(line).split():
         if word in concordance:
            concordance[word].append(linenum)
         else:
            concordance[word]=[linenum]
      linenum += 1
```

```
while True:
   word = input('Enter word: ')
   if word in concordance:
      print('Found on lines:', concordance[word])
   else:
      print('Not found.')
```

The new part is at the top. We use a `def` statement—short for define—to create a function called `cleanedup`. Note the letter `s` in parentheses on the first line of the function definition. When the function is called, a string of text will be passed for it to clean. The name `s` will refer to this string.

For example, lower down we write `cleanedup(line)`. In this case, `line` is the string passed to the function and `s` will refer to it. Actually, as a first step in executing this function call, the interpreter just does a simple assignment: `s = line`.

So when the function begins to work, s refers to the string to be cleaned. The body of the function—other than the last line—contains nothing new. This is exactly the code we used in our previous version to build up a clean line of text.

At the end of the function, `cleantext` refers to the resulting string that the function should return as its answer. We use a `return` statement to accomplish that. When a `return` is encountered inside a function, the function stops work immediately and returns the specified result.

6. Defining functions is a central part of programming. We've just seen that using functions makes programs *easier to understand* by breaking the work into pieces. A second advantage is that small pieces are *easier to write and test* than larger ones. Very often we proceed by writing and checking a number of small functions, using these to build other functions and so on, until eventually we can accomplish the main work of our program just by making a handful of function calls. You'll see a good example of this below in Program 6.

   A final advantage is that, *once a function is written, it can be reused*. This is true in two senses. A function can be reused within a program. For example, once the `cleanedup` function is defined, it can be called at more than one place in our code. But even more importantly, if we write this function today for use in one program, there is nothing to stop us from using it tomorrow in another program. Moreover, we can share useful functions with other programmers. Actually, we've already done that—`len`, `split` and `lower` are all functions we reused from someone else's previous work!