Good Habits, Dictionaries, and Files

Congratulations on making it to Week 4, you're over half way now!

Things are hotting up now, the problems are starting to get harder and the solutions are growing longer and more complex. So it is time to get into on a few good problem solving and programming habits before things get nasty.

Also, we are going to introduce you to one of the most cunning ideas in Computer Science — *dictionaries*. With dictionaries there are all kinds of fancy new programs you can write, and usually in very few lines of code.

Finally, we're going to quickly show you how to read text from a file. So this week's notes are packed with goodies!

1 Understanding the problem

One problem that tricked lots of people (at least for a while) in Week 1 was Introducing Eliza, so much so that many people emailed us asking if we had made a mistake! Like that ever happens :)

Now, we admit the question was a bit vague, and in many cases you need to look at the example input and output to really work out what the problem is. But, we aren't going to apologise for that — with good reason. It turns out this is the nature of defining what computer programs are meant to do. Working out exactly what tasks a user wants their software to perform is very (very) difficult.

In fact, a whole area of software engineering called *requirements analysis* exists just to solve this problem. And just like in the real world, it turns out that examples, often called *use cases*, are the best way of defining exactly what the program should do.

Hint

First read the problem and try to make sense of it. Then, use the example input and output to test whether your understanding of the question matches what we are expecting you to produce.

2 Attention to detail

One thing that many of you have battled with over the last couple of weeks is getting your program to produce output that our automatic marking system will pass. It might seem to you that our pesky marking system is overly pedantic since it require the output to be *exactly* as we have described it in the problems.

However, we have done this for a reason — computers are very unforgiving when it comes to even the most minor differences! What appears to be close enough for a human, e.g. these two lines:

```
3 6 9 12 15 18 21 24 27 30 33 36
[3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36]
```

are not at all similar to a machine that doesn't *know* to interpret them both as lists of numbers. This means that *attention to detail is a key skill* for computer programmers which you will develop as the challenge progresses.

Hint

Make sure, *character by character* and *line by line*, that your program produces the correct output for the example cases with the problem.

3 Solving Problems

Programs can be very complicated things to think about because there are lots and lots of little details that you need to remember and get right. Sometimes it gets a bit overwhelming and it seems very hard to know where to start on a program.

The best thing to do in these cases is to break the problem down into little pieces and think about solving them one at a time. Let's use Googlewhackblatts as an example.

A good place to start is to read in a word and just `print` it out. Try to copy as much code from the notes as possible and modify it. Since we are experimenting, we might as well type straight into the shell:

```
>>> word = raw_input('Enter a word: ')
Enter a word: dog
>>> print word
dog
```

Now let's try a separate experiment to multiply a string by an integer:

```
>>> word = 'dog'
>>> print word*5
dogdogdogdogdog
```

Ok, this is looking good. Now we just need to read an integer from the user. Again using the example in the notes for reading in a number is a good place to start.

```
>>> num = int(raw_input('Enter a number: '))
Enter a number: 4
>>> print num
4
```

You can then use this number to multiply the string:

```
>>> print word*num
dogdogdogdog
```

And we're practically there. All you need to do is copy the bits you've just experimented with into a single program and test that they do the right thing.

Hopefully, this approach might help you get unstuck on a bigger problem. But it is just one way and if you have something else that works for you then go for it!

Hint

- break each problem down into manageable little pieces
- reuse as much code from the notes as possible
- use `print` statements to see what's going on

4 Comments

As your programs get larger and take longer to write it can be easy to forget what a particular part was meant to do. Imagine the case where you hadn't looked at a piece of code for a year, would you be able to work out exactly what it did at a glance? We sure couldn't! And that's where *comments* come in.

Comments are helpful messages in the code to yourself and other programmers. Comments in Python start with `#` and continue to the end of the line. Commenting in the right places with enough detail takes practice.

Newbies often add comments like `# add one to i` which is fine for learning to program. However, you probably already (as most programmers do) know what `i += 1` does without the help of the comment. If there is a comment like this for every line of code, the program would become twice as long, making it much harder to read. On the other hand, comments like `# solve problem` are way too general, to the point where they don't provide any information at all.

Striking a balance is the key. A good rule of thumb is to write a comment to say what each major chunk of code does and to add comments for any line of code which does something complicated.

5 Dictionaries

An (English) dictionary is a book in which each word is associated with a definition. Looking up a word's definition is pretty fast because the words are in alphabetical order. A phone book is like a dictionary too — each name is associated with a phone number and again lookup is quick. Looking up definitions or phone numbers would be painfully slow if you had to read *every* entry to find the one you wanted.

A Python *dictionary* is a collection of *keys* which are associated with *values*. In our examples above the words and names are *keys*, and the definitions and phone numbers are *values*. Python dictionaries are also very fast and convenient to lookup compared with searching a list from the beginning.

Python dictionaries are created using braces `{}` (a.k.a. squiggly brackets):

```
>>> defs = {'dog': 'a barker', 'cat': 'a meower', 'elephant': 'a trumpeter'}
>>> defs
{'elephant': 'a trumpeter', 'dog': 'a barker', 'cat': 'a meower'}
```

This example creates a small English dictionary with silly definitions. The Python dictionary associates the key `'dog'` with the value `'a barker'`. A colon `:` is used to separate the keys and the values. There are three key-value pairs (called *items*) in this dictionary, one for `'dog'`, `'cat'` and `'elephant'`. Notice that when the dictionary is displayed, the key-value pairs are in a different order. That's because dictionaries don't keep track of the order things are added (unlike a list) so you can't rely on them being in a particular order.

6 Accessing Dictionaries

We can access items stored in a dictionary using square brackets (like lists and strings):

```
>>> defs['dog']
'a barker'
```

Here, `defs['dog']` returns the corresponding value `'a barker'` since `'dog'` is in the dictionary. Don't be confused by the different uses for square brackets. Item access for *any* Python type uses square brackets, but only lists are *created* using square brackets.

What happens if the key is not in the dictionary?

```
>>> defs['mouse']
Traceback (most recent call last):
  File ‘‘<pyshell#2>’’, line 1, in -toplevel-
    definitions['mouse']
KeyError: 'mouse'
```

Here, `'mouse'` is not in `defs` so a `KeyError` exception is thrown. So how can we tell if a key is in a dictionary? Easy, using the `in` operator (just like in strings and lists):

```
>>> 'cat' in defs
True
>>> 'mouse' in defs
False
```

Now we have everything we need to write a complete English dictionary lookup program:

```
defs = {
  'dog': 'a barker',
  'cat': 'a meower',
  'elephant': 'a trumpeter'
}

word = raw_input('Enter a word? ')

if word in defs:
  print word, 'means', defs[word]
else:
  print 'There is no such word as', word
```

Try doing the same thing for names and phone numbers.

7 Modifying Dictionaries

You can modify existing items in a dictionary using the same notation as for lists:

```
>>> phonebook = {'john': 11111, 'judy': 22222, 'jeff': 33333}
>>> phonebook['john']
11111
>>> phonebook['john'] = 44444
>>> phonebook['john']
44444
```

Initially, `'john'` has the phone number `11111` but we change it to `44444`. This example also shows that you can use numbers for values rather than strings.

You can also use the same trick to add more items to the dictionary. For example, to add a new item with the key `'bill'` and the value `55555` you use:

```
>>> phonebook['bill'] = 55555
>>> phonebook
{'judy': 22222, 'john': 44444, 'bill': 55555, 'jeff': 33333}
```

As well as writing out the entire dictionary, it is often easier to start with an empty dictionary `{}` and add elements (like `append` for lists):

```
>>> months = {}
>>> month['January'] = 1
>>> month['February'] = 2
>>> month['March'] = 3
```

Try writing a program which takes a date string like `'1 March 2005'` and converts it to `'1/3/05'`.

8 Dictionaries from input

Starting from an empty dictionary is often used with reading in key-value pairs from the user:

```
phonebook = {}

line = raw_input()
while line:
  (name, number) = line.split()
  phonebook[name] = int(number)
  line = raw_input()

print phonebook
```

When we run this program, we get the following:

There is lots more to learn about dictionaries, but this is all you will need to write some interesting programs of your own and complete this week's problems.

9 Reading from files

Python makes reading input from a *file* (as opposed to the Python shell) very simple. In fact, reading multiple lines from files is *even easier* than reading them from the shell using `raw_input`.

Before a program can read data from a file, it must tell the operating system that it wants to access that file. Files sitting in the same directory as the running program in IDLE can be referred to just using the *filename*, e.g. `test.txt`. This is the setup we will use in the Challenge.

10 Directories and Paths

A *directory* is the technical name for a folder. Accessing files in other directories requires extra directions to the file called a *path*. There are two kinds of paths: *absolute* paths and *relative* paths.

An *absolute path* starts from a fixed (or absolute) location on disk called the *root* (which on Windows is the drive name, like `C:`). This means absolute paths can be accessed in the same way from any directory. On Windows, absolute paths therefore start with a drive letter e.g. `C:Python25python.exe`. On Mac OS X/Linux it means an absolute path starts with a backslash e.g. `/usr/bin/python2.5`.

A *relative path* starts from (that is, is relative to) the current directory goes from there. The relative path `..` refers to the parent directory, so `..\..\data.txt` goes up two levels (to the grandparent directory) to find `data.txt`. Whereas `data/colours/rainbow.txt` goes down two levels from the current directory, inside `data` and then `colours` to find `rainbow.txt`.

An absolute path is like a set of directions that start from somewhere recognisable (e.g. the Opera House), while a relative path is like a set of directions that start from where the person is standing now.

Hint

This means that any text files that you download from us or, you have typed in yourself, will need to be copied into the same directory as the Python programs you are writing.

11 Opening a file

The process of requesting access to a file is called *opening* a file (just like it is for users) and is done using the builtin function `open` (or `file`). This creates a Python *file* object which we can then read from. Try the following after downloading `test.txt` from our website (or creating one yourself):

```
>>> f = open('test.txt')
>>> f
<open file 'test.txt', mode 'r' at 0x4a458>
```

This angle bracket representation is used for types in Python that it doesn't make sense to print (like the whole contents of the file). But it lets you know that you successfully opened the file.

What happens when you try to open a file that doesn't exist or isn't in the location you expect it to be:

```
>>> f = open('missing.txt')
Traceback (most recent call last):
  File ``<pyshell#0>'', line 1, in -toplevel-
    open('missing.txt')
IOError: [Errno 2] No such file or directory: 'missing.txt'
```

You get an `IOError` exception. More importantly you also get the message `No such file or directory` which tells you what particular problem has occurred.

Now, there is one more thing that is a bit of a problem: DOS/Windows, Mac 9/OS X and Linux all use a different convention for representing end of lines in text files. Windows uses two characters **\r\n**, Macs use one character **\r** and Unix operating (such as Linux) use **\n**. To help overcome this problem Python has a special mode for reading files which converts all of these different standards to **\n** (which makes our lives much much easier). This mode is called universal newline support and is set when the file is opened with a second argument **'rU'**:

```
>>> f = open('test.txt', 'rU')
```

12 Looping over a file

Ok, now after all that it's easy to read from an opened file because you can treat it just like a list in a **for** loop:

```
>>> f = open('test.txt', 'rU')
>>> for line in f:
...   print line,
...
This is a test file.
It contains 3 lines of input.
This is the last line!
```

That's all there is to reading in a file! As you can see it is actually shorter than using **raw_input** in a **while** loop. Notice that we have put a comma after **print line**. That's because line still has the newline character on the end. Try removing it to see what happens to the output. If you want to reproduce the example above exactly then you can download **test.txt**.

Now we can actually make this a bit shorter because we don't need to put the file in a separate **file** variable, so we can substitute it straight into the **for** loop itself:

```
>>> for line in open('test.txt', 'rU'):
...   print line,
```

This produces the same output as the example above and is the recommended approach to read input from files.

Let's combine file reading with some of our existing data structures.

13 Reading a list from a file

Reading from a file into a list is also much simpler than it is at the Python shell:

```
colours = []
for line in open('rainbow.txt', 'rU'):
  colours.append(line)

print colours
```

The file **rainbow.txt** used in this example is available from here.

When this runs we get:

```
['red\n', 'orange\n', 'yellow\n', 'green\n', 'blue\n', 'indigo\n', 'violet\n']
```

Notice that the newline characters are still attached to the ends of the string. To remove the newline characters we would have to chop them off either with a slice **line[:-1]** or with a **line.rstrip()** which removes whitespace from the right hand side of the string.

14 Reading a dictionary from a file

Reading from a file into a dictionary is a little more complicated because we not only want to read the lines, but we also need to split each line into the key and the value:

```
numbers = {}
for line in open('french.txt', 'rU'):
  (english, french) = line.split()
  numbers[english] = french

print numbers['one'], numbers['two'], numbers['three']
```

The file **french.txt** is also available from our website. This example prints out one, two, three in French:

```
un deux trois
```

The extra work that goes on here is that **line.split()** firstly breaks the line into two strings, which are stored in the variables **english** and **french**. These are then used to add new items to the dictionary which translate between English numbers and their French equivalents. Try changing this program so that it takes the numbers to translate from the user.