Marching Up and Down the Code

Learning Python 3 on the Raspberry Pi

Dr. D.W. Joyce, M.S. Joyce

Pre-release 0.2.4.5b44ecd

DON'T PANIC!

Copyright © 2015 - 2016 by David & Matthew Joyce

Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0):

You are free to:

- **Share** – Copy and redistribute the material in any medium or format.
- **Adapt** – Remix, transform, and build upon the material.

The licensor cannot revoke these freedoms as long as you follow the license terms:

- **Attribution** – You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
- **NonCommercial** – You may not use the material for commercial purposes.
- **ShareAlike** – If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
- **No additional restrictions** – You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

Read the full licence at https://creativecommons.org/licenses/by-nc-sa/4.0/legalcode.

Get the latest version of this book at https://github.com/dwjoyce/pythonbook/releases/latest and the source at https://github.com/dwjoyce/pythonbook.

CONTENTS

CHAPTER

ZERO

STARTING WITH PYTHON'S IDLE

Most of the good programmers do programming not because they expect to get paid or get adulation by the public, but because it is fun to program.

— Linus Torvalds

0.0 Introduction

Welcome to our guide on learning Python!

Programming is the art of logical thinking. It is the process of breaking down a problem or task into simple steps so even a dumb computer can follow them. These steps are written down according to a set notation, so that a computer can translate them into its own machine code. Thankfully, people no longer have to program in machine code themselves, made up of numbers, but in a higher form, constructed mainly from English words. Computer languages are different from human languages in that they are formal and unambiguous. The formal aspect means they are specified in advance with a set notation and the use of keywords. They are unambiguous so that the program will always do the same thing every time, given the same circumstances.

Knowing how to program a computer is a great skill to have, even if you are not a professional programmer. You can program in almost any field of work - especially when you need to transform data (the basis of information) from one form to another. It means you are not simply using other people's work, but you have the ability create for new programs by yourself. If you can never program a computer, it is like living in a world that you can only read other people's writing, unable to write for yourself. Programming is very much a creative process - you are instructing a computer to follow your what you are thinking, down to the letter!

Python is a fantastic first computer language to learn. It is easy to pick up, but at the same time is very useful to computer professionals so is not just a beginner's language. It is used by large companies such as NASA, Disney, Google, YouTube, Microsoft, Yahoo and Dropbox. It is also used by many educational institutions when learning programming, such as Cambridge University. It also has the advantage of being free and open source software, so you can download it at home free of charge. This also means you can study how it works.

It was created in the early 1990s by the Dutch programmer Guido van Rossum. It has been updated many times since then, and in this book we will be using Python 3. The exact version of Python 3 does not matter so much, as long as your version is Python 3.something. This is for when you are trying to program at home.

Code written in Python is very readable. It lacks much of the cryptic notation that other languages use to express themselves. It almost reads like *pseudo-code* - a description of how a program should operate as written in lines of plain English.

Python is also very interactive. This means that you can experiment with the language, getting to know it better, without a cumbersome process getting in the way. Although we will write proper programs later (from *chapter 6* - First Program), initially we will use Python's interactive interpreter to start off. It is called an *interactive interpreter* because you type lines of code (known as *statements*) and get the answer straight away, i.e. interactively. It is an interpreter because it translates the lines of Python into a form that the computer can understand directly.

0.1 What we will be learning

We will be learning the basics of the Python language, just enough for us to write interesting and challenging programs (and have a bit of fun).

We will cover data types, organising our data and code, opening and reading files on disk, handling errors. We introduce the three ways code can flow - sequentially, selectively and iteratively. We will not cover more advanced features in any detail such as object orientated programming, dictionaries, list comprehensions, lambdas, generators, decorators, and the like. We may use some of these features, but not code them ourselves. In fact, what we do teach will not be covered in great depth, only enough for you to start programming yourself. Not only do we not have time for this, but these topics are not entirely necessary to do the programming we want to achieve.

This book unfolds in a way that should be familiar to school children. We first use Python to perform calculations and call functions, just as you have learnt in your maths class. The arithmetic is calculated in a way familiar to those using proper scientific calculators, not basic ones. We use the language to store values in memory, again just like a calculator, but in more flexible manner. Only then do we move onto writing complete programs and using concepts that are less familiar to those following a high school curriculum. We use the Raspberry Pi as our hardware platform, as reflected in the pictures, but any computer could be used instead, particularly at home.

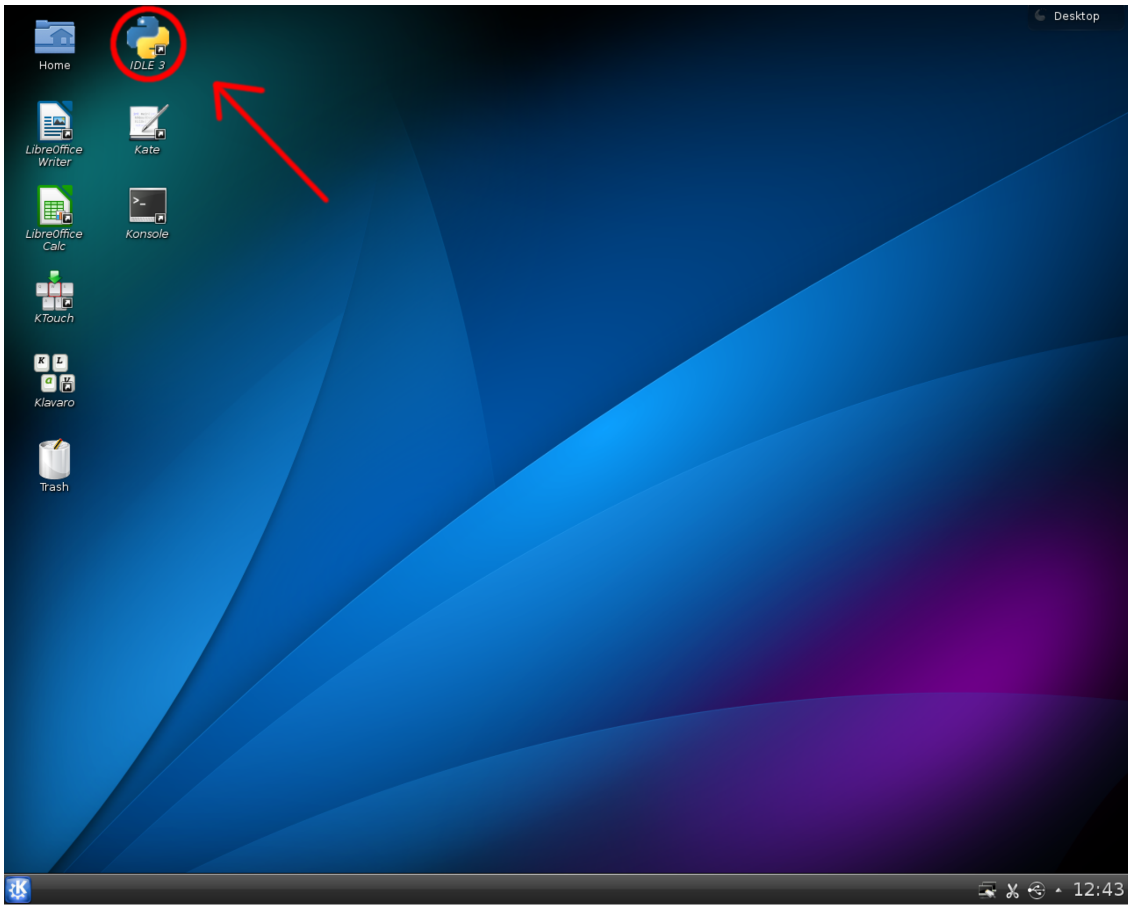
It is encouraged to use this book as a starting point, and then use other resources to continue your practice in the art of programming. Remember, it is all about being creative and getting computers to follow our instructions!

0.2 How to get going

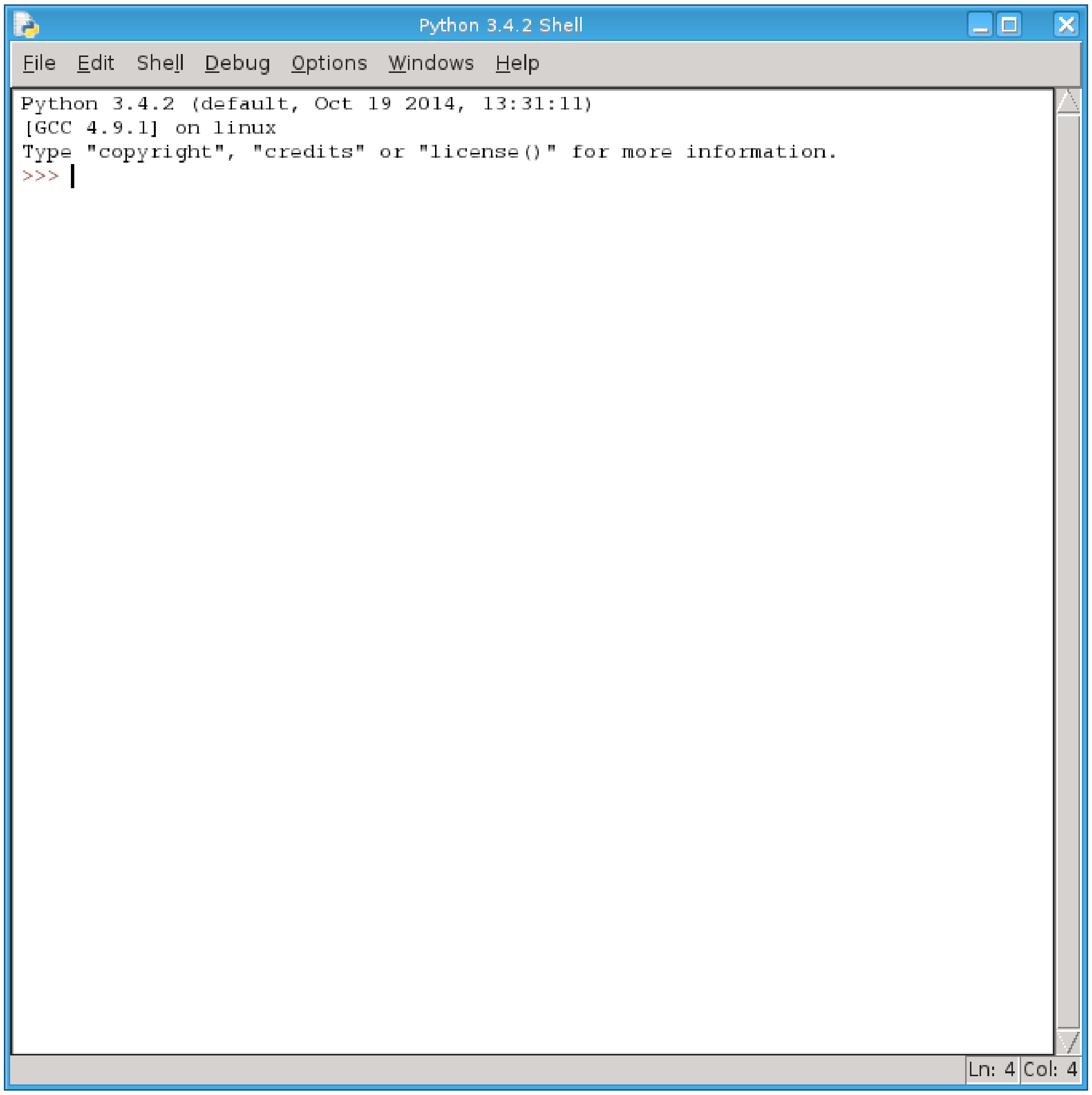
We first use the interactive interpreter to start with Python. Even when introducing more advanced language features, we will still use this shell to experiment before moving onto a proper program. A proper program is one typed into its own file, and then the program is started separately. This is similar to how we use other programs (e.g. LibreOffice or Firefox) - we use the final result. This is known as *running* or executing the program. However, at this stage, we only use the interactive interpreter.

Python's interactive interpreter is known as *IDLE*. This name comes from the acronym *IDE*, which stands for Integrated Development Environment - a program that allows you to develop or create other programs. It contains an editor, used to type your code, and a way of running your programs. IDLE also contains the interactive interpreter that allows us to experiment line by line.

To start IDLE, either click on the IDLE icon on the desktop or select the IDLE program from the Applications menu in the bottom left:



IDLE will then start, and you will have a window on your screen which looks like this:



```
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>>
```

You are now ready to go.

Later, in *chapter 6* on your First Program, you will position this IDLE window slightly differently in order for you to see both the program you are writing and the interactive interpreter at the same time. Until then, this is all you need to know about starting Python's shell. Next chapter - typing in some code and seeing what happens!

0.3 Things to remember

1. Click on the IDLE desktop or panel icon to start Python's interactive interpreter.
2. Type away in the interactive interpreter and see what results you get. Don't be afraid to experiment, you can't break the computer that easily. Fortune rewards the brave!

3. To recall something already typed in, use the up arrow to move the cursor onto the line you wish to use. Press the `Return` or `Enter` key - this brings it down onto your current line. You can modify what has been copied down. To run the line again, press the `Return` or `Enter` key a second time. This will save you lots of typing!
4. Read these *Things to remember* sections in each chapter very carefully and remember what they say!

CHAPTER

ONE

PYTHON AS A CALCULATOR

There, it should work now.

— All programmers

1.0 First steps (or sums)

Let us start our journey by taking inspiration from something we all know well - your pocket calculator.

To perform a sum on a calculator, such as 10 plus 20, you could simply hit the following buttons:

$$10 + 20$$

and then hit the $\boxed{=}$ (equals) button. The result, 30, will then appear on your screen. We could do other operations as well, such as subtraction, division and multiplication, like so:

$$10 + 20 - 4 \div 2 \times 3$$

and then finally hitting the equals button, you will get a result of 24 displayed on the calculator's screen. This is assuming you are using a proper calculator, not a simple one which performs the calculation as it goes along, one step at a time! In other words, it performs the division first, then the multiplication, and then subtracts this answer from the result of the addition.

So given their name, we should be able to use *computers* to do some *computing*, that is, working with numbers. Particularly, we should be able to use our new programming language, Python, to do this for us.

Using the first example, the Python code is very simple.

Bring up your Python interactive interpreter, as described in *chapter 0* (i.e. by clicking on the IDLE icon on your desktop), and type the following:

```
>>> 10 + 20
30
```

and press the `Return` or `Enter` key on your keyboard. The `>>>` (chevrons) appear automatically, so do not type these! The chevrons just mean that IDLE is ready for you to type something new. You should see the number 30 displayed below the line you typed, as in the example above.

How about the second example? Let us try this:

```
>>> 10 + 20 - 4 / 2 * 3
24.0
```

The answer is the same as with our calculator example above. However, what are these `/` and `*` symbols? Well, the ÷ doesn't actually appear on your computer keyboard, so we use another symbol `/` (forward slash) instead. And the × is too much like the letter `x`, so we use the asterisk `*` symbol instead. These *signs* or symbols in computer programming are called *operators*, and we have leant four so far - `+` (addition), `-` (subtraction), `*` (multiplication) and `/` (division). The value it is operating on or using is called an *operand*.

Type in it, press the `Return` or `Enter` key, and see what the result is. The result should read 24.0, the same as when we were using the calculator earlier. This is not a whole number, but a fractional number - it has a decimal point included. In Python, we will deal with two *types* of numbers - whole numbers (*integers*) and fractional numbers (*floating point* or real numbers).

Remember, like in mathematics, a computer language does not necessarily work out the sum from left to right - it gives priority or precedence - to some operations over others. It actually performs the division first (4 divided by 2, equalling 2), then the multiplication (2 times 3, equalling 6), then the addition (10 plus 20 equalling 30) and finally the subtraction (30 minus 6 equalling 24). To see the full list of operator precedence - which operator is processed before others - then refer to *Appendix B.1* on Operators.

To enforce a different order, you can use brackets `(` and `)` - just like in maths. Put a pair of brackets around each part of the sum you want done separately. So if you wanted to do the addition and subtraction section first, then type the following:

```
>>> (10 + 20 - 4) / 2 * 3
39.0
```

In programming (and maths too), this way of expressing a value to form a result is called an *expression*.

1.1 Operator overload

On your calculator there are more than 4 buttons to do things. There is one labelled x^2. And x^3. Python has this built-in as well and it is the `**` (power) operator. For example, the number 9 to the power of 2 is as simple as:

```
>>> 9 ** 2
81
```

Which is 81. 2 to the power of 3 is:

```
>>> 2 ** 3
8
```

Which is 8. This works for any power. How about working out large numbers, such as 19^8? To do this, type in the following:

```
>>> 19 ** 8
16983563041
```

Your answer should state 16983563041. Negative numbers work as well. Remember that 2^{-1} is the same as $1 \div 2$? Doing this in Python is similar:

```
>>> 2 ** -1
0.5
```

Your answer should read 0.5.

Note: Operators almost always have a value either side of them, such as `10 + 20`, or `2 ** 8`. The general exception when using the minus sign `-` or the plus sign `+`, in which case it appears you are using two appears in a row, such as `30 + -10`, which would result in a value of 20. Apart from negating a value, you should use these arithmetic operators with two values (one to the left, one to the right), not just one.

We can now move onto something your calculator cannot do. Remember when you were in primary school, and you learnt that 7 divided by 3 was 2 remainder 1 (or to put it another way, $7 \div 3 = 2\,r\,1$)? To get the quotient (in this case 2), use the `//` (floor or *integer division*) operator:

```
>>> 7 // 3
2
```

Which should be 2. And for the remainder, use the `%` (*modulus*) operator:

```
>>> 7 % 3
1
```

Resulting in 1. Reading both answers together, we have got 2 remainder 1. To practice further:

```
>>> 10 // 6
1
>>> 10 % 6
4
>>> 29 // 8
3
>>> 29 % 8
5
>>> 24 % 2
0
>>> 25 % 2
1
```

The last two demonstrate that 24 is even (no remainder when divided by 2), and 25 is odd (a remainder of 1)!

1.2 Exercises

1. Type the Python line to work out 3 plus 5 divided by 2.
2. Type the Python line to work out 4 times 2 minus 7.
3. Type the Python line to work out 7 to the power of 6.
4. Type the Python lines to work out the quotient and remainder of 11 divided by 4.
5. To convert from Celsius to Fahrenheit temperature, you multiply the Celsius by 9 divided by 5, and add 32. If the Celsius is 40, what is the Fahrenheit reading?

1.3 Things to remember

1. Remember your arithmetic operators, using the values 2 and 5 as an example:

Name	Operator	Example	Maths equivalent
Addition	`+`	`5 + 2`	$5 + 2$
Subtraction	`-`	`5 - 2`	$5 - 2$
Division	`/`	`5 / 2`	$5 \div 2$
Multiplication	`*`	`5 * 2`	5×2
Power	`**`	`5 ** 2`	5^2
Floor division (integer division)	`//`	`5 // 2`	$\lfloor \frac{5}{2} \rfloor$
Modulus (remainder)	`%`	`5 % 2`	$5 \ mod \ 2$

   See *Appendix B.1* for a full list of operators, over and above arithmetic.

2. Use brackets to force Python to do a calculation in a particular order
3. Whole numbers are referred to as integers, fractional numbers are referred to as floats (floating point).
4. In programming, values are known as *expressions*, potentially made up of other values, operators and even other expressions that need to be evaluated to form a result. They express a value.

CHAPTER

TWO

NAMING YOUR DATA

A computer once beat me at chess, but it was no match for me at kick boxing.

— Emo Philips

2.0 Saving and recalling

Remember in the previous chapter we looked at how your calculator works and started using the Python programming language to copy and build on this.

One feature of a calculator is the Memory button, usually marked by the letter `M`. This allows you to store a number, add onto or subtract from it, and recall it later for use in a new calculation. It effectively makes the calculator remember a number for you, so that you do not have to.

All programming languages, including Python, have a similar feature. Instead of having a memory button, Python has *variables*. It is really just a way of putting a name against the data (e.g. a number) you are working with, giving your program the ability to store data, change it (i.e. vary its value), and recall it later.

For example, the initial sum we worked on in the previous chapter:

```
>>> 10 + 20
30
```

Python will *evaluate* this sum, give you the result, but it will not save it anywhere. It effectively throws the answer away.

How can we save the result in memory?

Similar to a calculator, we can store it in a variable called `M`:

```
>>> M = 10 + 20
```

What this does is take the integer values 10 and 20, with the addition operator it adds them together and then assigns them, using the *assignment* operator `=` (equals), to the variable name `M`.

Note: In Python, variables names are like labels attached to boxes which contain data. If you put a box of things away in the attic or in storage, the label will describe what is inside the box, with the box containing the things themselves. In Python, the label is the variable name, the box contains the data, e.g. an integer number.

We can then recall the value stored against this name later. For example:

```
>>> M + 5
35
```

You will see it gives an answer of 35. The variable name `M` refers to 30 (from the addition of 10 and 20 above), and then we are adding a further 5 to it, giving 35.

If we now try:

```
>>> M + 10
40
```

You will notice it results in 40. It does not take the previous result of 35 and add on 10, as we did not store that back into `M`, but simply recalled the value of `M` and used it as before.

To change the value of `M`, we have to put it back on the left hand side of the assignment operator:

```
>>> M = M + 10
```

This changes `M` once more, using it on the right (recalling `M`, being 30, and adding on 10), and putting that back into the variable name `M`. We are effectively reusing the same name, or giving it a new value.

To see what the new value of `M` is, just type `M`:

```
>>> M
40
```

This gives 40, as expected (30 plus 10, as above).

2.1 More and more

Your calculator will have just the one memory to store numbers, but in a Python program you can have have any number of variables in your program. And they do not have to be called `M` either! In fact, if you want to store your data separately, you need different names for each value.

For example:

```
>>> M1 = 5
>>> M2 = 10
>>> M3 = 15
```

Try this, so one variable makes use of two others:

```
>>> a = 10
>>> b = 20
>>> c = a + b
>>> c
30
```

And this:

```
>>> first_num = 123
>>> second_num = 456
>>> third_num = first_num * second_num
>>> third_num
56088
```

And arithmetic operators can be used that we learnt in our previous chapter:

```
>>> A1 = 10 + 20
>>> B2 = A1 - 4
>>> C3 = B2 / 2 * 3
>>> C3 + A1
69.0
```

That last line displays the value of `C3`, calculated from the value of `B2`, and adds on the value of `A1`.

You can name your variables with any combination of letters (UPPER and lowercase) and numbers along with the `_` (underscore, not minus sign) character, as long as the name does not start with a number. You can use the underscore to separate words in your variable names to make them more readable (e.g. `first_time_entry` instead of `firsttimeentry`) - spaces are not allowed inside names as this would appear as two names, not one!

2.2 Exercises

1. Define a variable called `age` and set it to your age (use an *integer* number).
2. Use your `age` variable to calculate how many days old you are (assume each year has 365 days).
3. Again, use your `age` variable to calculate in what year you will be 100 years old. You will need to take the value of `age` from the present year, 2015, and then add on 100.

2.3 Things to remember

1. You define a variable by giving it a name, and using the *assignment* operator to give it a value. The value can be evaluated (calculated) from other variables.

2. Begin your variable names with a letter from the alphabet (upper or lowercase) or the `_` (underscore) character. To use the underscore character, you need to press the **`Shift`** key down whilst pressing the key to the right of the 0 (zero), the **`-`** key.
3. Use the `_` (underscore) character to divide up words in your variable names to make them more readable, e.g. `first_num`.

CHAPTER

THREE

FUNCTIONS AND MATHS

Ah. I'd like to have an argument, please.

— Michael Palin

The Argument Sketch - Monty Python's Flying Circus

3.0 Basic functions

In Maths, you learn about *functions*. An example of a simple function is:

$$f(x) = x + 1$$

This function takes in a number, x, and adds one to it. So if $x = 1$, then $f(x) = 2$. In Python-speak, x is an *argument*, and the result, $f(x)$ is the *return value*. Every function gives back a value, its return value.

Python also has *functions*. You can make your own functions, but in this chapter, we will focus on using functions that Python already has, called *built-in functions*. The first function we'll use is called `abs`, which stands for absolute value:

```
>>> abs
<built-in function abs>
```

What this function does is, given any number, positive or negative, it returns a positive number with a magnitude equal to the number you give it. This means that if you give it `6`, it will return `6`, but if you give it `-9`, it will return `9`. The way you use this function is you write the function name, `abs`, then a opening parenthesis, `(`, then the number you want, let's say `-42`, and finally a closing parenthesis, `)`, to tell Python that we are finished:

```
>>> abs(-42)
42
```

It works just as you would expect. We can do this with other numbers, including numbers with decimal places:

```
>>> abs(2)
2
>>> abs(-123.45)
123.45
>>> abs(0)
0
```

We can say that `abs` is a function that takes one argument, a number, and *returns* the positive version of that number. In the above example, we can say that we *called* `abs` with `-123.45` as an *argument*.

3.1 More arguments

But not all functions take one argument. An example of a function that takes two arguments is `round`. It takes a number to be rounded, usually a fractional number, and a whole number of decimal places to round the first number to. So rounding `123.12` to `1` decimal place should give `123.1`. How do we call a function with more than one argument? We separate the arguments with commas. To call round as we described, we write `round`, then `(`, our first argument, `123.12`, then a comma, `,`. We then write our second argument, `1` and a closing parenthesis, `)`:

```
>>> round(123.12, 1)
123.1
```

`round` can round to any number of decimal places to round to:

```
>>> round(98765.4321, 3)
98765.432
>>> round(0.123456789, 7)
0.1234568
>>> round(0.123456789, 0)
0.0
>>> round(0.123456789, 20)
0.123456789
```

When the number of decimal places to round to is greater then the precision of the number to round, `round` does nothing. `round` can also take a negative number of decimal places, like scientific notation (or standard form):

```
>>> round(12345.6, -1)
12350.0
>>> round(12345.6, -3)
12000.0
```

If you call `round` with `-2`, it makes the last two non-fractional digits zeros. Another two-argument function is `pow` (power). `pow(x, y)` is equivalent to `x ** y`:

```
>>> pow(3, 4)
81
>>> pow(-2, 5)
-32
>>> pow(64, 0.5)
8.0
```

3.2 How many arguments can one function have?

That depends! Some functions take any number of arguments. `min` is a function that takes two or more arguments, and returns the smallest one:

```
>>> min(1, 8)
1
>>> min(4, 1, 9)
1
>>> min(-2, 5, -256, 7, 2, -5, -10, 100)
-256
>>> min(0.5, 0.125)
0.125
```

If you don't give `min` enough arguments, Python gives an error:

```
>>> min()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: min expected 1 arguments, got 0
```

Other functions also give errors if you don't give the right number of arguments:

```
>>> abs()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: abs() takes exactly one argument (0 given)
>>> abs(1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: abs() takes exactly one argument (2 given)
>>> round(5, 9, 1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: round() takes at most 2 arguments (3 given)
```

`max` is a similar function to `min`, except that it returns the largest argument:

```
>>> max(1, 8)
8
>>> max(4, 1, 9)
9
>>> max(-2, 5, -256, 7, 2, -5, -10, 100)
100
>>> max(0.5, 0.125)
0.5
```

3.3 Functions functioning

You can use the return value of a function as an argument to another function, assign the return value to a variable and use variables as arguments:

```
>>> max(-10, abs(-20))
20
>>> the_biggest_num = max(4, 9, 23, 56, 12, 5)
>>> the_biggest_num
56
>>> a = 3
>>> b = -4
>>> c = 5
>>> min(a, b, c)
-4
```

Functions are also variables, so you can assign functions to new variables:

```
>>> func = abs
>>> func(-8)
8
>>> func
<built-in function abs>
>>> abs
<built-in function abs>
```

3.4 Exercises

1. Use `abs` to find the absolute value of `-35.5`.
2. Use `round` to round `-22.8364926` to `4` decimal places.
3. Use both `round` and `abs` to find the absolute value of `-7495.184758` to `2` decimal places.
4. Use `max` and `min` to find the smallest and largest number from `7`, `-8`, `4`, `-12` and `1`.

3.5 Things to remember

1. *Functions* are called with *arguments* to give a *return value*.
2. To call a function `func` with no arguments do `func()`.
3. To call a function `func` with an argument `arg` do `func(arg)`.

4. To call a function `func` with more than one argument, separate the arguments by commas: `func(arg1, arg2, arg3, etc)`.
5. Functions return values that can be assigned to variables, e.g. `num = abs(-10)`, and variables and return values can be used as arguments, e.g. `abs(num)` and `abs(round(-10.75))`.

CHAPTER

FOUR

DRAWING TURTLES

Theory is when you know something, but it doesn't work. Practice is when something works, but you don't know why. Programmers combine theory and practice: Nothing works and they don't know why.

— Anonymous

4.0 Importing

Let us move from using Python to do maths and switch to doodling instead!

What we will do is to use a Python program called `turtle` to move a shape around the screen, leaving a trail behind in the process. Think of it like using a piece of graph paper, with the origin in the centre, and the pen being moved by your instructions.

To use this separate turtle program, we have to use a new command called `import`. What *import* does is to bring in, or include, a separate program - called a *module* in Python - into your own program. We cannot ever hope to write every piece of code ourselves, so often we depend on programs that others have been written and build on them.

It was Isaac Newton who said:

> *"If I have seen further, it is by standing on the shoulders of giants."*

In other words, he only made the advances in the fields of mathematics and physics he did by building on the work of those who came before him.

Programming is similar. If we make use of the work of others, we can go far. We can build more interesting programs much faster by using other people's code which we can depend upon. Modules that Python itself offers as standard, ready to be included into your program, are often very well written and tested.

So, to include another program, we must use the `import` command, and give it the name of the module to import. Although this program will have the `.py` filename extension (e.g. `turtle.py`), we do not include that part when naming the module.

Therefore, to use the `math` (for mathematics) module, you would type:

```
>>> import math
```

and then you can use it thereafter, such as the square root function:

```
>>> math.sqrt(64)
8.0
```

which should give us the answer of 8 (8.0 to be exact), as you would expect. You cannot use a module until you have performed the import, not before! Notice the `.` (period) character - it separates the module name (`math`) from the function being used (`sqrt`). We must write the name of the module first, followed by the `.` period, and then the function name that is to be found inside the module. The period is used to say that this function is found inside this module - we will see more of this later when talking about functions that belong to a particular type.

Here is another example - using the value of π as defined as a variable by the `math` module:

```
>>> math.pi
3.141592653589793
```

Using the name `math.pi` is not calling a function, it is referring to a floating point (i.e. fractional) variable inside the `math` module, therefore we do not need to use parentheses. Although we refer to `math.pi` as a *variable*, we do not expect it change in value. We call this type of value a *constant*.

4.1 Stick your head out of the shell

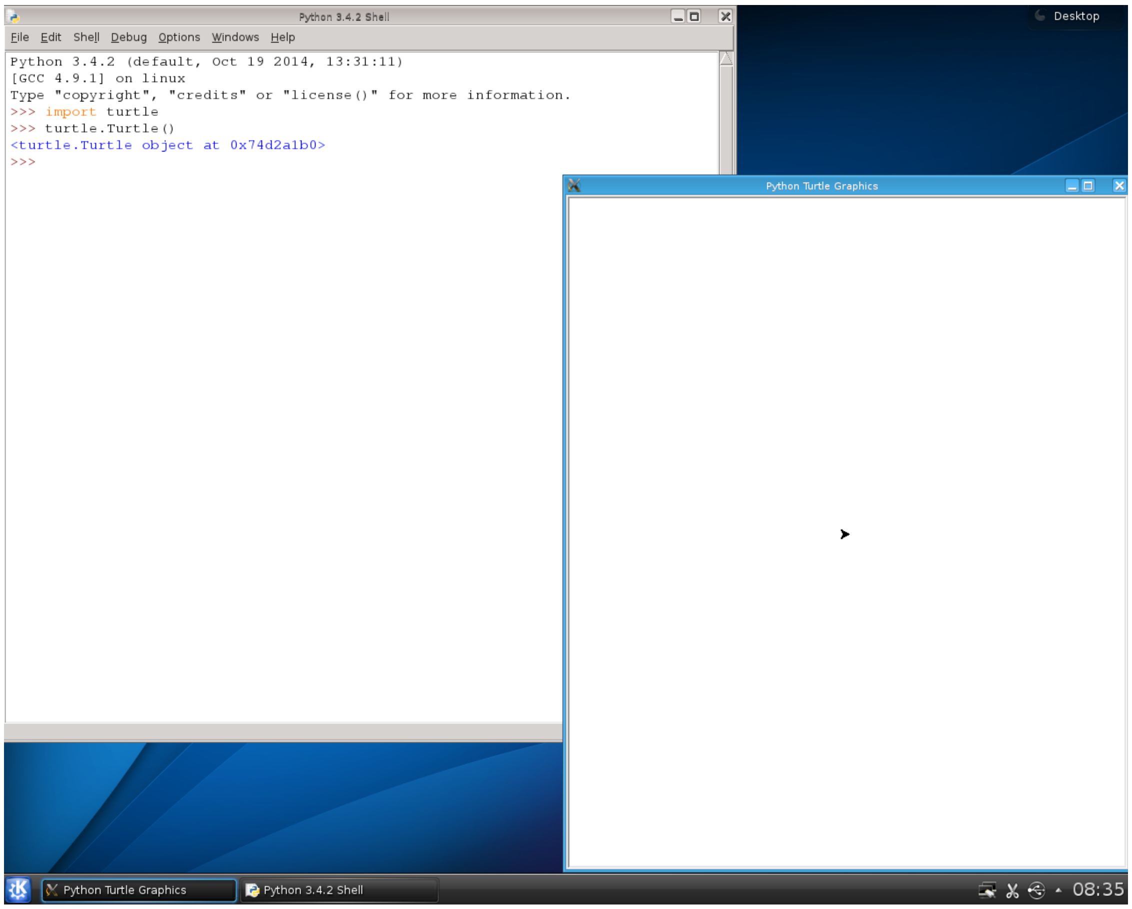
So let us get back to our drawing. To import the turtle module, you just need to type the `import` command followed by the name of the module, as follows:

```
>>> import turtle
```

Now we can begin to use it. The first time we use a function inside the turtle module, a window (similar to a canvas or graph paper) will pop up so that we can draw in it. Let us do this by drawing something, so type in the following:

```
>>> turtle.forward(100)
```

You should see a new window pop onto the screen. Move it to the right of your screen so it does not obscure what you are typing, like so:



```
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> import turtle
>>> turtle.Turtle()
<turtle.Turtle object at 0x74d2a1b0>
>>>
```

See how the small shape leaves a trail behind as it moves. Let us carry on:

```
>>> turtle.left(90)
>>> turtle.forward(100)
>>> turtle.left(90)
>>> turtle.forward(100)
>>> turtle.left(90)
>>> turtle.forward(100)
>>> turtle.left(90)
```

We have drawn a box! We have effectively done the same thing four times - moved forward 100 places (measured in *pixels*, which stands for picture elements), and then turned left 90 degrees each time.

If you make a mistake, you can go back a turn, or *undo* your previous move, by typing the following:

```
>>> turtle.undo()
```

There are lots of other functions to call as well. Try the following:

```
>>> turtle.circle(75)
>>> turtle.right(30)
>>> turtle.forward(50)
```

Cont...

```
>>> turtle.begin_fill()
>>> turtle.circle(40)
>>> turtle.end_fill()
```

The `begin_fill` must be called before you start drawing your shape, and the `end_fill` function is called when the shape is complete. The turtle program then knows what to fill in.

Here is a list of turtle functions you may find useful: `forward`, `left`, `right`, `up`, `down`, `goto`, `begin_fill`, `end_fill` and `undo`.

4.2 Exercises

1. Draw a hexagon - a six sided shape, where the angle of turn is 120 degrees (180 minus 60 degrees).
2. Draw a star that has been filled in. Hint: try turning 144 degrees and 72 degrees, alternatively, each time you draw a spike.
3. Draw a house, complete with roof, windows and door. You will need to use `turtle.up` and `turtle.down` to pick the pen up and put it down, respectfully, so that you do not draw a line everywhere.

4.3 Things to remember

1. You can use another program by using the `import` command and the module's name (without the `.py` extension).
2. You cannot use variables or functions from a separate module until you have imported it.
3. Use the `.` character to dip inside a module, with the module name first, and the variable or function from inside the module second.

CHAPTER

FIVE

GETTING HELP

Don't worry if it doesn't work right. If everything did, you'd be out of a job.

— Mosher's Law of Software Engineering

5.0 Save Our Sanity

Python is a very helpful programming language when we need to find out more information. And it is all built-in so we do not even need to go elsewhere to find it.

On a general level, you can use the interactive interpreter to enter the help utility like so:

```
>>> help()
```

So if you remember what we taught in *chapter 2*, this means that `help` is a function, and it is being called by placing parentheses `()` (round brackets) after its name.

When we are in the help utility, the prompt changes from `>>>` to `help>`, to avoid confusing the two. The first thing to learn is how to exit the help system:

```
help> quit
>>>
```

In actual fact, just pressing the **Enter** key without any text will do the same!

Now, if you re-enter the help utility, you can type any command or function to get further information on that item. For example:

```
>>> help()
help> round
```

This will display some information on the `round` function. If the help utility does not recognise what you have typed, it will say so.

You need not enter the help utility to get further information, though. You can do it from the interactive prompt as well:

```
>>> help(round)
```

This will display the same information as before, but it takes less effort to get to it!

You can also get help on values or types of data:

```
>>> help(10)
>>> help(12.8)
```

These will display information on integers and floating point numbers, respectively. Do not worry about what is displayed at this point - much of it will not make much sense yet!

Using the proper names of the types of data will display the same information:

```
>>> help(int)
>>> help(float)
```

To get a list of what is built into Python, you can call the *directory function* in this way:

```
>>> dir(__builtins__)
['__class__', '__contains__', '__delattr__', '__delitem__', '__dir__',
 '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
 '__getitem__', '__gt__', '__hash__', '__init__', '__iter__', '__le__',
 '__len__', '__lt__', '__ne__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__setattr__', '__setitem__',
 '__sizeof__', '__str__', '__subclasshook__', 'clear', 'copy',
 'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem', 'setdefault',
 'update', 'values']
```

This will list a number of error codes, followed by a number of built-in functions. These functions can be used just by referring to their name, followed by parentheses to call them. For example, one of the built-in functions is `round`, so we can simply call it as follows:

```
>>> round(10.75)
11
```

which returns a value of 11, as you would expect.

You can get more help on these functions by using the help function again. For example:

```
>>> help(round)
```

will give more information on the round function, such as what it accepts (a number, and optionally the number of digits to round the number to), and what it returns back (another number, the rounded result).

In the previous chapter, we introduced the notion of bringing in a separate program (often called a *module*) into our own using the **`import`** command. But how do we know what available modules there are to import? To find this out, we can type the following:

```
>>> help('modules')
```

This takes a few seconds for Python to work out, but it will list every available module that can be imported by your program, including the `math` or `turtle` modules we used previously.

To see what is inside a module, once you have imported it, you can perform a `dir` on the module name, for example:

```
>>> import turtle
>>> dir(turtle)
```

To delve deeper, we can find out more information on functions inside these other modules by using the dot notation as introduced last time. So, for example, if we want to find out more information on the square root function inside the `math` module, we could do the following:

```
>>> import math
>>> help(math.sqrt)
```

Note: You must import the module before you can get help on its contents. Otherwise Python does not know what is inside it.

For further information and a guide on the language, please refer to *The Python Tutorial* at `Computing/Python 3.4.2 docs/tutorial/index.html`, which includes many more examples and covers more language features than we will be doing in this guide!

5.1 Things to remember

1. To get help, use the `help()` function in the interactive interpreter with the item on which your require further information, and the `dir()` function to get a directory listing on the `__builtins__` (double underscore at each end) or a particular module.

CHAPTER

SIX

FIRST PROGRAM

Computers are good at following instructions, but not at reading your mind.

— Donald Knuth

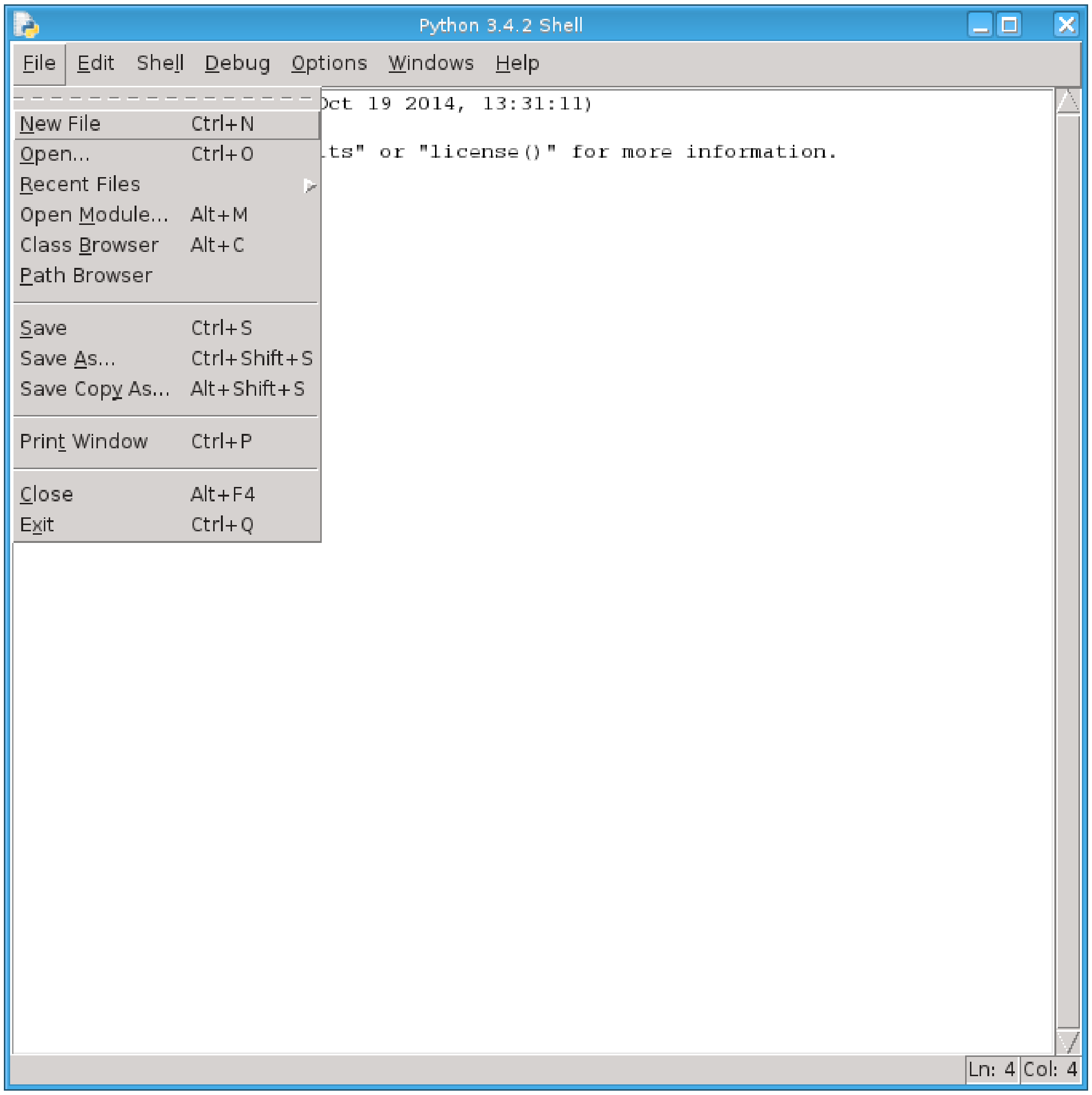
So far we have been using Python's interactive interpreter, built into the IDLE program, to experiment with Python in a way that is familiar to us - performing calculations, calling functions, giving values in memory a name. We have started to step out of this familiar territory by using other programs (known as modules), such as the `turtle` module, to extend what Python can do by itself. Using the interactive interpreter allows us get to know the language, but it limits us effectively to one statement of code - a line that when you press the **`Return`** or **`Enter`** key, will be run straight away and gives you back an answer (if there is one).

We will now learn how to use IDLE in order to write a full program, one that can be run on its own again and again. This means we can write the program once, and run it many times without the need to re-write the program each time. We only need to change our program to modify its behaviour, or correct errors, not having to it from scratch each time. It also means we can write longer, more interesting programs!

6.0 Ed

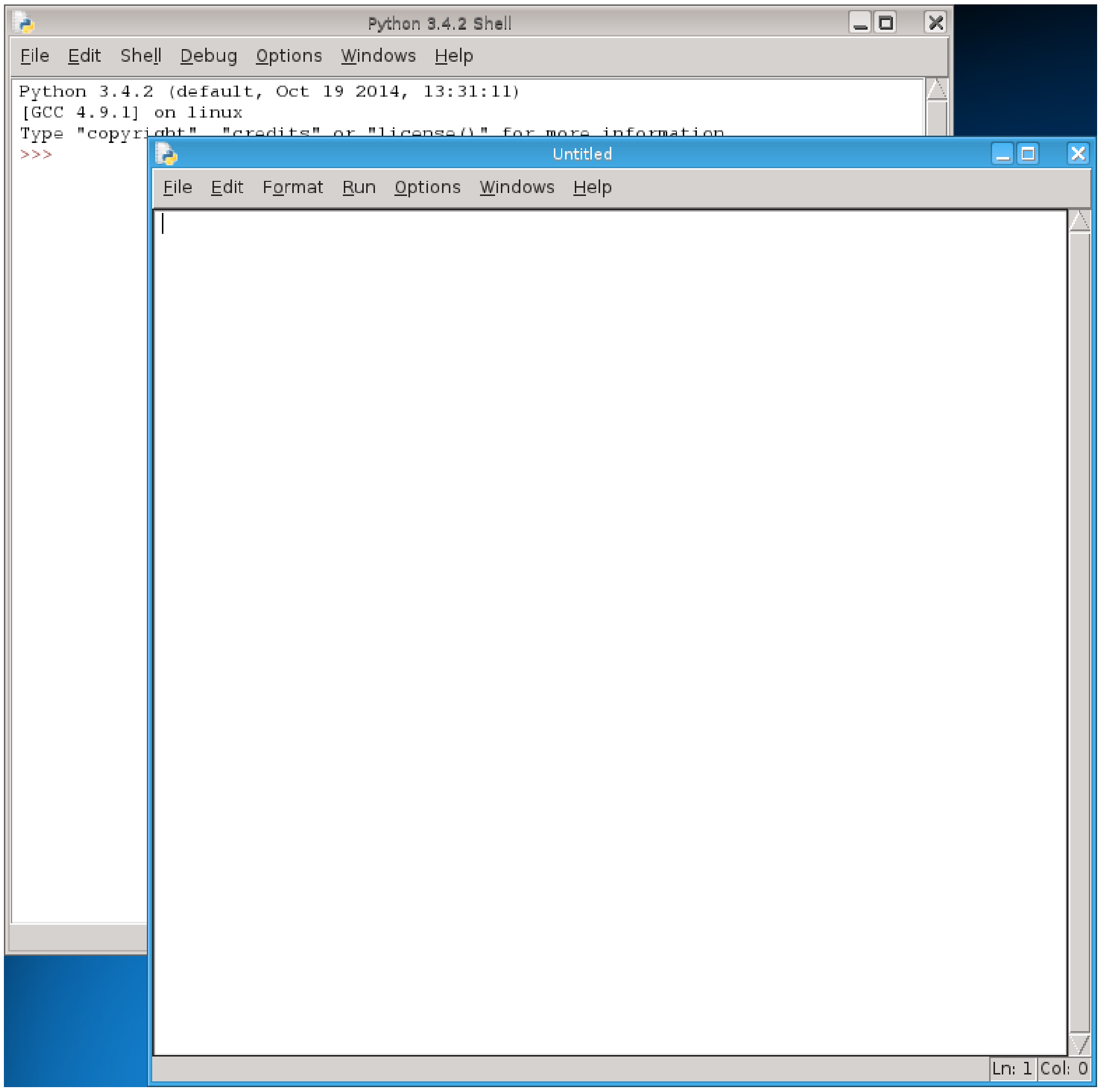
To write a program, and modify it later, we use an *editor*. Thankfully, IDLE not only has the interactive interpreter we have been using, but an editor built-in as well.

To start Python editor, first start IDLE itself (see *chapter 0* for this if you have forgotten). Then, using the mouse pointer, click on the **`File`** menu at the top of the interactive window, and click again on the **`New File`** menu item, like so:



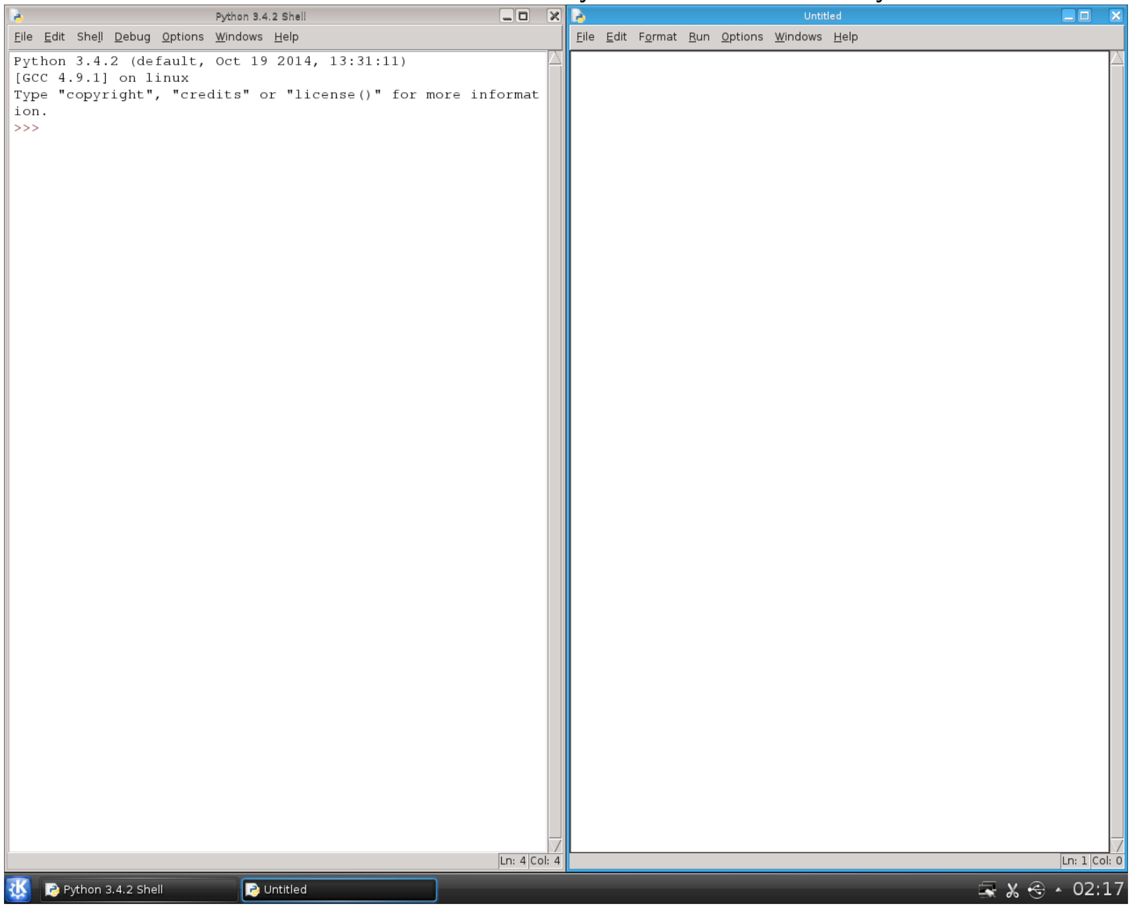
```
Oct 19 2014, 13:31:11)

ts" or "license()" for more information.
```

This will bring up a new window, the editor window. The shortcut for this is to hold down the **Ctrl** key, towards the bottom left of your keyboard, and hit the **N** key (N for New). This blank window is where we type in our new program:


Python 3.4.2 Shell
File Edit Shell Debug Options Windows Help
Untitled
File Edit Format Run Options Windows Help
Ln: 1 Col: 0

```
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>>
```

Tip: The best arrangement is to have your interactive interpreter on the left and the editor window on the right. You can do this on the Raspberry Pi by using the mouse, clicking on the interactive interpreter title bar using the left mouse button (keeping it pressed down) and try to pull the top of the interactive window past the left hand side of the screen - it should then snap into a shape that takes up the left 50% of your screen - and let go of the mouse button. Do a similar procedure with the editor window, but try to drag it past the right side of the screen. You will then have the two windows you need to see, side by side:


Python 3.4.2 Shell
File Edit Shell Debug Options Windows Help
Untitled
File Edit Format Run Options Windows Help
Ln: 4 Col: 4
Ln: 1 Col: 0
Python 3.4.2 Shell
Untitled
02:17

```
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more informat
ion.
>>>
```

The scene is set.

6.1 Our first program

Our first complete program will be the same as the code we typed in for *chapter 4* on drawing turtles. Therefore, in this new editor window, type in the following:

```
import turtle

turtle.forward(100)
turtle.left(90)
turtle.forward(100)
```

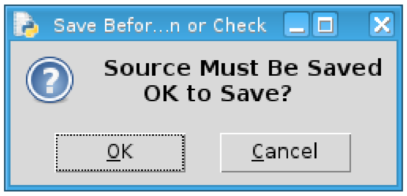
Cont...

```
turtle.left(90)
turtle.forward(100)
turtle.left(90)
turtle.forward(100)
turtle.left(90)

turtle.begin_fill()
turtle.circle(40)
turtle.end_fill()
```

Type in everything, including the blank lines! You can use copy (**Ctrl-C**) and paste (**Ctrl-V**) to repeat the lines in the middle to save time.

6.2 Running our program

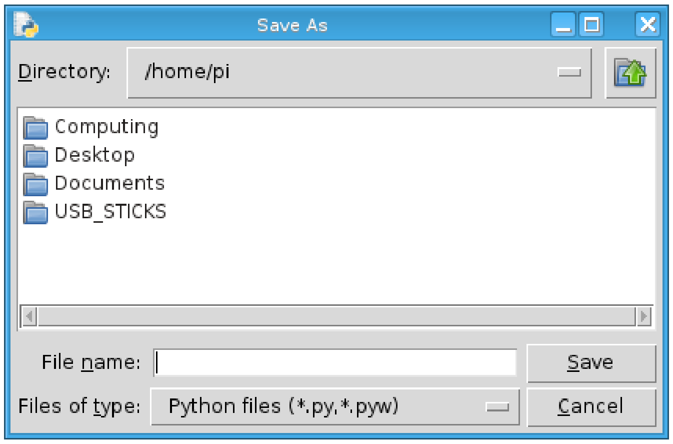
Once you have typed it all in, we are ready to run our program. To run, press the **F5** key (or if you really want to do it the hard way, then click on the **Run** menu and select **Run Module** menu item). This will bring up a dialog box like this one:



Source Must Be Saved
OK to Save?

Simply agree to this by pressing the **Return** or **Enter** key on your keyboard. Again, if you like doing things the hard way, then use your mouse to click on the **OK** button.

On this, the first time of writing your program, you will need to give it a name. If you have already done this, then IDLE should proceed onto attempting to run your program. The **Save As** dialog is shown, asking you where to save your program and what to call it:



Directory: /home/pi

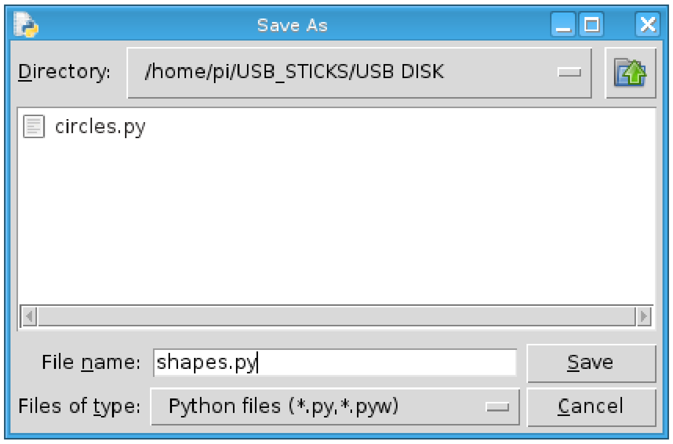
Computing
Desktop
Documents
USB_STICKS

File name:
Files of type: Python files (*.py,*.pyw)

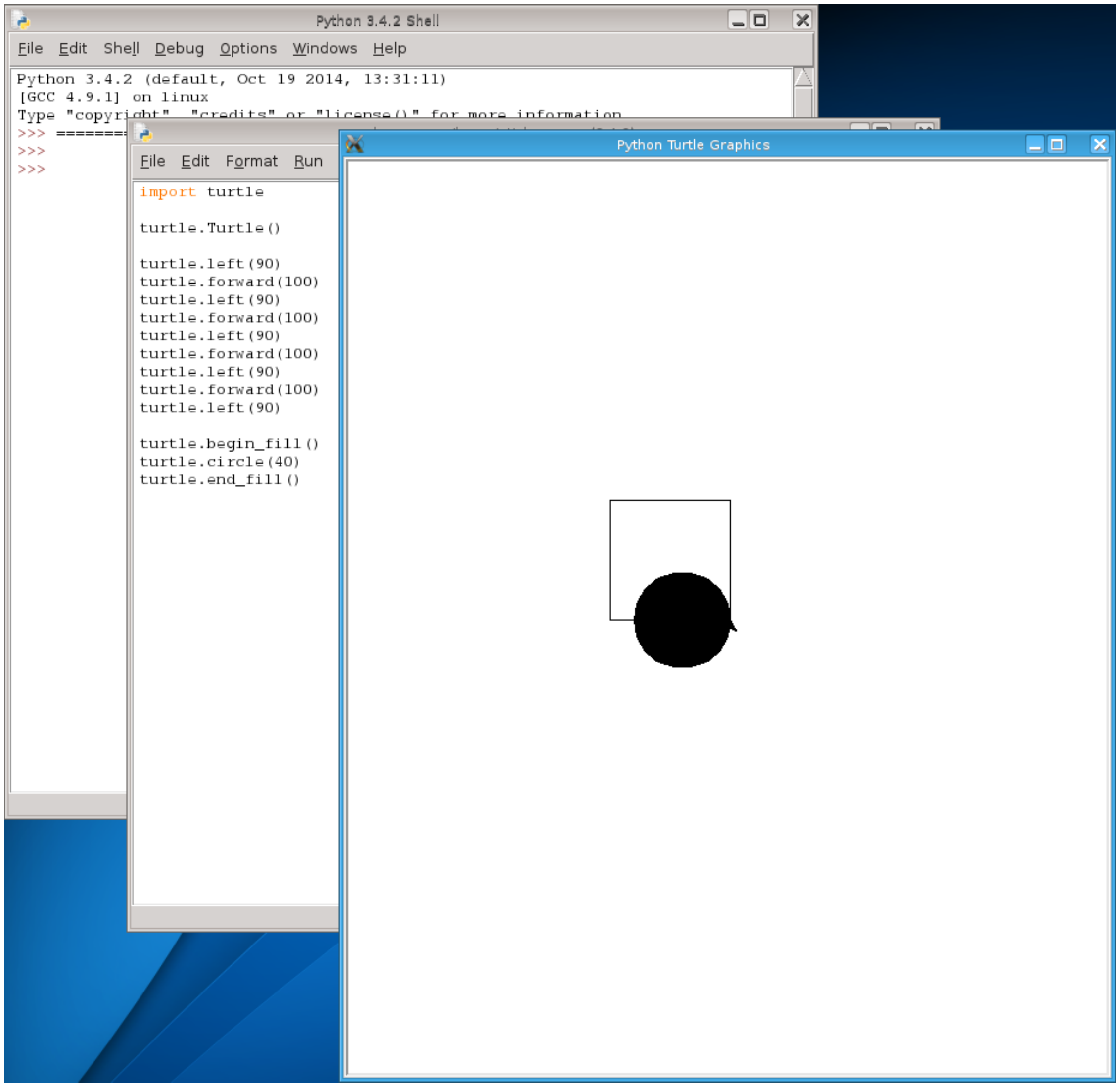
Most importantly, change the location of where the program is being saved to either your USB stick or the computer directory with your name. This is so your own programs are saved in the same place, and you will be responsible for looking after it.

Tip: To save on your USB stick when the **Save As** dialog box appears, first click on the directory `USB_STICKS`, and then click on the name of your USB stick, which is `USB Disk` by default.

Once you are in the right place, give your program a name (such as `shapes.py` in this instance) and click the **Save** button (or hold down the **Alt** key and hit the **S** key), like so:



Your program will now run! It should look roughly like this:



```
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> ========
>>>
>>>
```

```
import turtle

turtle.Turtle()

turtle.left(90)
turtle.forward(100)
turtle.left(90)
turtle.forward(100)
turtle.left(90)
turtle.forward(100)
turtle.left(90)
turtle.forward(100)
turtle.left(90)

turtle.begin_fill()
turtle.circle(40)
turtle.end_fill()
```

However, the alternative is that it doesn't work - there is a problem or error in your program. To correct this, return to your edit window, and change it so that it reflects the above program exactly. Python, and all computer languages, are very fussy - if there is an error, the computer is not allowed to guess what you meant like a person could do!

6.3 Comments

That is not all - we can also insert *comments* to help other people understand what is going on (including ourselves when we examine our own work in the future). Modify your program so it looks like this - in other words, add the lines beginning with the `#` symbol:

```
import turtle

# Draw a square - move forward and turn for each side
turtle.forward(100)                                           Cont...
turtle.left(90)
turtle.forward(100)
turtle.left(90)
turtle.forward(100)
turtle.left(90)
turtle.forward(100)
turtle.left(90)

# Draw a filled circle
turtle.begin_fill()
turtle.circle(40)
turtle.end_fill()
```

Here we have dedicated each line containing the `#` symbol as a comment, a piece of annotation. It is simply a note to describe what is going on - but do not over do it. If the line is obvious, then there is nothing to say. The best thing is to describe sections of code, or if the code is hard to understand. It is best to describe the thinking behind why you wrote the code in the way you did, rather than simply repeat what the code is saying already.

It is also possible to put comments at the end of lines, like so:

```
turtle.end_fill()  # fill in the drawn shape
```

Effectively, Python ignores everything after the `#` symbol - up until the end of the line. The next line starts afresh.

6.4 Things to remember

1. To start a new program, click on the **File** menu and click on **New File**. This opens up a new window ready for your program to be typed in. The shortcut for this is to hold the **Ctrl** key down and hit the **N** key.
2. The best arrangement is to move your interactive interpreter window to the window so it takes up the left half of the screen, and the editor window to the right so it takes up the right half of the screen. See the tip above to see how to do this.
3. Once the program has been typed in, or at least a little of it, then to run it you either click on the **Run** menu and select the **Run Module** item, or press the **F5** key as a shortcut. You will be asked to save your program, so simply press the **Return** or **Enter** key. If you are running your program for the first time, you will need to give a name. Save all your programs into the directory with your name, or onto your USB stick. If you are using a USB stick, it will appear in the `USB_STICKS` folder in your home directory once you have plugged it in.
4. Name your Python programs with a *.py* postfix (e.g. shapes.py). Avoid including spaces in your program name, and include only one period `.` character.

5. *Comments*, or annotations intended for other people to read, are denoted with hash `#` symbol.

CHAPTER

SEVEN

PRINT THAT OUT!

What you see is all you get

— Brian W. Kernighan

7.0 Printing numbers

Now that we have started writing proper programs that can be saved and run over and over, we will soon discover a little problem.

To see this, open up a new file (click on the **File** menu and select **New File**), and type in the following into the new window that appears:

```
a = 10
b = 20
c = a + b
c
```

This is just the same code as we did in Chapter 2 when working with variables for the first time. However, back then, we typed in each individual line in the interactive interpreter, and it gave an answer every time if there was one.

What happens now, given that it is living in its own program? Try it and see - press the **F5** key, click **OK** to save it and give it the name `sum.py`. Remember, the `.py` postfix stands for Python. Moreover, save all your programs in the appropriate place (the directory with your name, or onto your USB stick).

When the program runs in the run window (the same as the interactive interpreter), what does it display onto the screen? Does it give 30 as expected? No, it does not!

The reason for this is that just typing a value (e.g. `10`) or an expression to be evaluated (e.g. `10 + 20`) or a variable name (`c` in the example above) will do nothing in a real program. The interactive interpreter is just that - interactive. But in a real program, you have to tell Python what to do with the item you are dealing with. Otherwise, the value will simply be discarded as it is not being used for anything.

So what do we want to do with the variable `c` above? We want to display or print it out onto the screen. In Python, the way to do this is to use the `print` function. You simply wrap your value inside the parentheses in the `print` call, and it will then print it out as we originally intended. Therefore, modify your program so it looks like this:

```
a = 10
b = 20
c = a + b
print(c)
```

Run your program again, using the F5 key (you may have to click the OK button, or just press the Return or Enter key to confirm). This should now print out the number 30 in the run window.

You can print out more than one item by using a comma `,` between the items to separate them. Modify the last line in your program again:

```
a = 10
b = 20
c = a + b
print(a, b, c)
```

and run, again by using the F5 key and pressing the Return or Enter key to confirm. This will print out 10, followed by 20 and finally 30, all on the same line.

You can even get the `print` function to perform the calculation for you when passing in the values to print. All arguments like this are evaluated (processed or simplified) before the values are presented to the function to use. Again, modify your program like so:

```
a = 10
b = 20
c = a + b
print(a, b, c, a + b + c)
```

This will print out the three numbers from before (10, 20 and 30), and then 60 (all the variables added together), without the need of a fourth variable to hold this extra number.

7.1 Printing messages

Dealing with numbers all the time is very useful, but it is bit limiting. Computers do not just compute numbers! They also deal with textual messages, not to mention pictures, music and videos! Dealing with messages containing text is very easy, but subtly different.

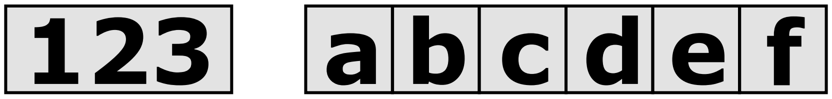
To deal with text, we need to enclose the words with quotation marks, just like how a piece of speech in a book is surrounded by quotation marks. In Python, it is very similar.

Open up a new file (click on the File menu and select New File), and type the following:

```
print("Hello World!")
print('How are you?')
print("I love Python")
print('This is fun!')
```

Save this file as `lines.py` (press the **F5** button, press the **Return** or **Enter** key to confirm and save it in the usual place), and see the text being printed out onto the screen. Notice how we use either double quotation marks `"` or single quotation marks `'` - Python does not mind which one you use, as long as you are consistent. This means that if you start with a double quote, then you must close with a double quote.

These pieces of text in Python are called *strings*. Think of them as strings of characters, made up of either letters from the alphabet, numerical digits or symbols. This includes almost any key from your keyboard, and more besides. Similar to a string or chain of pearls, a string in programming is a sequence of characters. To illustrate, whilst with an integer number (shown below on the left) is stored as a whole number (with the 1s, 10s, 100s, etc., all in their correct places), strings (shown on the right) are simply a sequence of characters, where digits are treated the same as letters and symbols:



We now know three types of data - integers, floats and strings.

Modify your `lines.py` program to include strings and integers together, below the lines you added previously:

```
print("1 plus 2 equals:", 1 + 2)
print("I am", 18, "years old")
print("That bag of apples cost", 75, "pence")
```

Again note how we separate a number of items being printed together - using a comma between each.

We can even have a bit of fun, and join and replicate strings like so:

```
print('One piece of text' + 'joined to another')
print('How about this ' * 10)
```

Here we are using the addition operation `+` to join two items of text together, and then print out the result. Notice, the result does not have a space between each item, as the other examples previously do. This is called *string concatenation*. The multiplication operation `*` is used to repeat the string however many times you specify - 10 times in this example. This is called *repetition*.

Variables can also be assigned to strings. Add the following onto your program, `lines.py`:

```
name = 'Fred'
occupation = 'Farmer'
age = 25
print('Here are my details:', name, occupation, age)
```

Finally, you can use special characters in strings to denote certain things. Here I will introduce just three of them, so add these lines to finish:

```
print('Here is a newline character\nThis is now on a separate line!')
print('This introduces a horizontal tab \t to space out my text')
print('I love St. Michael\'s - notice the quote inside the quote!')
```

The first one breaks the line with a newline character (`\n`), the second spaces out the text using a tab, and the third is a way of using quotes inside quotes, otherwise Python will get confused between an apostrophe (e.g. St. Michael's School) and the closing quotation mark.

7.2 Keeping things in line

If you wish to call `print` a number of times, with each item being appended onto the end of the line rather than starting a new line, then you need to use a special argument called `end`. If you set `end` to a particular string value, it will use that instead of a new line. For example, add these lines to the end of your `lines.py` program and run it once again:

```
print('This is the first line')
print('This will appear on another line')
print('But these two lines', end='')
print(' will appear on the same line!')
```

You could, of course, specify the `end` argument to be anything you want, but it is most common to either not use it at all, or set it to an empty string (two single quotes, one after the other). This kind of argument is called a *keyword argument*, which will be described further in chapter 19 on functions.

Tip: When you are writing a program later on and cannot see why it is not working as you would expect, try inserting some `print` statements in the code with the variables your program is using. That way, you will see what is going on, whilst it is running. This should then show you what needs changing to make it work better. This is called *debugging* your program.

7.3 Exercises

1. Assign 5 variables to various numbers, and print them out, along with the sum.
2. Print out the year of your birth, your age, and your age in 10 years time.
3. Print out the same as number 2, but with strings of text indicating what number is what, e.g. I was born in: 1999 my age: 16 in 10 years: 26
4. Print out your name a hundred times, with a tab (using the code `\t`) to introduce space in-between each item.

5. You can use strings to describe colors when drawing with `turtle`, for example, `'black'`, `'white'`, `'red'`, `'green'`, `'blue'`, `'cyan'`, `'magenta'`, `'yellow'`, etc. Two turtle functions in particular can be called; the first called `pencolor` (note the American spelling of color), and the second called `fillcolor`. So for example, you can call `turtle.pencolor('red')` before you start drawing, or `turtle.fillcolor('yellow')` before drawing a shape. Write a program that draws a number of circles with different pen and fill colors.

7.4 Things to remember

1. Use the `print` function to display or output any *expression* onto the screen.
2. Separate the items to print using a comma.
3. A piece of text can be used by enclosing it in quotation marks, whether using single (e.g. `'fred'`) or double quotes (e.g. `"fred"`).
4. The only arithmetic operators that can be used with text are addition (i.e. joining strings together, known as *concatenation*) and multiplication (*repetition*).
5. Use the `\n` for newline, `\t` for tab and `\'` or `\"` (quotation marks) inside strings.
6. Use the optional `end` argument when calling the `print` function to keep subsequent calls to `print` on the same line, rather than starting a new line.
7. We have now covered three types of data: *integers*, *floats* and *strings*.

CHAPTER

EIGHT

ASKING QUESTIONS

Where is the **any** *key?"*

— Homer Simpson

In response to the message, "Press any key"

8.0 String input

It is now time to make our programs more interactive, allowing the user to get involved whilst the program is running. Up until now, Python has carried out our instructions, one line at a time, with the program doing exactly the same thing every time.

In this chapter, we will allow the user to affect what happens in the program with the opportunity to enter data of their own. The way Python allows the user to enter data is by using the `input` function. This reads in what the user is typing on the keyboard, and returns the value to the program as a string so that it can be stored or used elsewhere.

Begin by creating a new program (**File** $\longrightarrow$ **New File**), and type the following:

```
name = input('What is your name? ')
print('Hello there', name)
```

Press **F5** to run your program, and give it the name `hello.py`. You will notice the program pauses at the first line - it is waiting for your input. Use the keyboard to type in your name, and press the **Return** or **Enter** key. You may have to click on the interactive window used to run your program with your mouse to make it active. Whatever you typed in will be stored in the variable `name`. This variable is then used in the second line to print out a message along with the value referred to by the variable `name`.

Note that we are using a message (a *string* value) when calling the `input` function - this is the message that is presented to the user when you are asking for input. This is not essential - you could miss it out, but then the user may not know they are expected to type something. It is best to present the message, so they know what to do next.

8.1 Using numbers

The type of data given to us by the `input` function (i.e. its return value) is always a string.

Remember, when you *add* two strings, you are really joining them together or *concatenating* them (e.g. `"12" + "34"` would equal `"1234"`). When you *multiply* a string by a number, you are *repeating* the contents of the string (e.g. `"123" * 3` would equal `"123123123"`).

Therefore, if you need to use the input as an actual number, you need to convert it from a string type to an integer or float type. This means you can then use the result in a normal calculation like any other number. It is effectively converts a sequence of digits into a proper number where the right most digit is the 1s, the second column is the 10s, the third column is the 100s, etc., with all the numbers combined together to form a complete number.

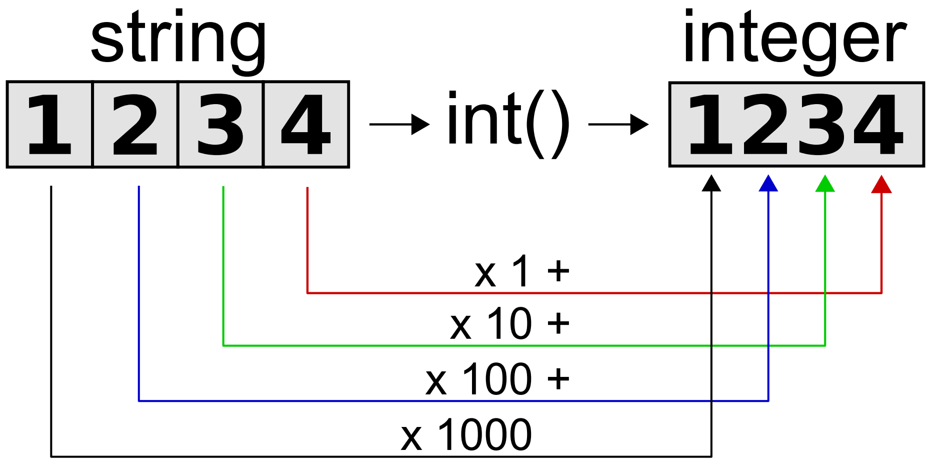
Note: Each character of a string is actually encoded according to a table of numbers (a character set). One common character set is *ASCII* (American Standard Code for Information Interchange), which includes all the Latin characters, digits and common symbols. Another is *Unicode*, which includes characters from many other languages. These character sets encode the letter 'A' as the number 65, the digit '0' as 48, and the symbol "!" as 33. This demonstrates how text is stored inside computer systems.

To convert from a string to an integer, you use the `int` function. So for example, `int("123")` would return the actual number `123`.

To convert from a string to a floating point number, you use the `float` function. So for example, `float("123")` would return the actual number `123.0`.

To convert back from a number (whether integer or float) into a string, you use the `str` function. So for example, `str(123)` would return `"123"`, and `str(123.0)` would return `"123.0"`.

To appreciate what is involved in converting a string of digits into an integer using the `int` function, look at the following diagram to see how Python multiplies the place values, and adds them all together:



To practice, start a new program called `sumup.py`, and type in the following into your new window:

```python
first_num_str = input('First number: ')
second_num_str = input('Second number: ')

first_num = int(first_num_str)
second_num = int(second_num_str)

total = first_num + second_num

print('The sum of those two numbers is', total)
```

Press F5 to run it, confirm to save, and name your program `sumup.py`. Careful when entering those numbers - they must be integers, otherwise converting from a string to an integer in the program will not work!

To explain what is going on, we first input what the user has typed in, and save it in a variable called `first_num_str`. We do the same again for `second_num_str`. We then convert this input from a string value to an integer value so that we can perform a proper numerical addition operation. We do this addition with the line where we assign a new variable called `total` to `first_num` added onto `second_num`. We finish by printing out a message, printing the value of total alongside.

Run your program again with different numbers to check it works. Now run it, and instead of entering integer numbers (e.g. `10`, `20`, `-50`, `123`, etc.), type in a fractional number (e.g. `10.75`). This should result in an error, as the period `.` is not part of an integer number. Python will not like this!

To correct this, let us finally modify your program to change the calls to the `int` function, so that they call the `float` function instead, as follows:

```python
first_num_str = input('First number: ')
second_num_str = input('Second number: ')

first_num = float(first_num_str)
second_num = float(second_num_str)

total = first_num + second_num

print('The sum of those two numbers is', total)
```

Now try entering fractional numbers, and it should handle them quite happily.

8.2 Exercises

1. Ask the user's name. Print it out a 100 times.
2. Ask the user's name and a number. Print out the name that number of times. You will need to convert the number string to an integer using the `int` function before repeating

the name string.

3. Ask for a day between 10 and 20. Print it out with the letters "th" appended onto the day, as with a date. So if the user entered 10, print out `10th`; if the user entered 18, print out `18th`. There is no need to convert the number to an integer - use the addition operator `+` to simply join the data input and the letters "th" together.
4. Expand on the `sumup.py` program we did in the chapter so that it also prints out the second number subtracted from the first (i.e. the first minus the second), the first divided by the second, and both numbers multiplied together.

8.3 Things to remember

1. Use the `input` function to read input from the keyboard. You can store this in a variable by putting the variable name and the assignment operator to the left hand side of the call to input. Moreover, you can optionally include a message (a string) to present to the user when the program pauses for input.
2. To convert to an *integer*, use the `int` function.
3. To convert to a *float*, use the `float` function.
4. To convert to a *string*, use the `str` function.

CHAPTER

NINE

PERFORMING SELECTION

If Baggins loses, we eats it whole.

— Gollum / Sméagol

9.0 Taking a different route

With what we have learned up until now, we can do arithmetic, store values against variable names, call functions, import other people's modules, and even get input from the user. We have also dealt in different types of data, whether numbers (integers or floating point) or strings (sequences of letters, digits and symbols). Our programs have started running (technically known as *executing*) from the top, and finished at the bottom, performing each line (a code statement) at a time, without any deviation whatsoever.

If a program simply performs the same instructions, line by line, every single time, then effectively they do the same thing every time they are run. This is not very interesting! Programs become more useful when they can make decisions on what to do, depending on the circumstances. Usually, this involves testing the value of a variable, and then performing some instructions over others. In programming, this is known as *selection* - the program is selecting some code statements for running, and not others - it is making a decision.

For example, if the program is working out the price of a cinema ticket, and VIP tickets cost 20% extra, the program needs to take a decision as to whether to add on this 20% or not. It cannot add on 20% for all the tickets!

To achieve this in Python, we use the `if` statement. We start with the word `if`, and then give it a test to evaluate. If the test evaluates to true (i.e. it is successful), then the statements underneath are performed (executed). If the test evaluates to false (i.e. it was unsuccessful), then the statements underneath are skipped.

Note: Statements that belong to an `if` statement must be pushed to the right in order to show what code belongs to what line. In programming, this is called *indentation*. In Python, we indent by four spaces. When the code block is finished, we unindent back to the column of the initial line.

Indentation is like drawing up a shopping list, like so:

```
clothes:
    shirt
    socks
    hat
food:
    apples
    milk
    cheese
    bread
others:
    cup
    clothes pegs
```

Each section is defined by a header, followed by the `:` colon symbol to indicate what follows belongs to the header. The items beneath each header are indented, or pushed in to form a group.

Let us start a new program called `vip.py` - click on the **File ⟶ New File** menu item, and type in the following:

```python
print('Welcome to our VIP program for calculating cinema ticket prices')
print('Ticket prices are £5.00 for ordinary tickets, £6.00 for VIPs')

vip = input('Do you want a VIP ticket, yes or no? ')

price = 5.0
```

So far, so good! Now we need to take a decision, so add the following to your program:

```python
if vip == 'yes':
    price = price * 1.2
```

You will notice we have used the `if` statement to perform a test. The test is whether the variable `vip`, created when we saved the answer from asking the user the question over VIP tickets, is the same as the string "yes". To perform the test, we have introduced a new operator, called the equals (comparison) operator. It is looks similar to the assignment operator, which creates variable names, but the equals operator has two equals signs, not one. It is testing whether what is on the left is equal to what is on the right. If the equals test is successful, then the expression is `True`, and the code drops into the code below the `if` statement, indicated by the code being indented to the right. If the test is unsuccessful, then the expression is `False`, and the code is skipped over.

Also note the use of the colon `:` symbol. This is used at the end of every line that has other lines that are attached to it. We will be seeing that on many more occasions in the future with other statements we will be introducing. It effectively tells Python to run the following lines if the test just evaluated was true. Please don't miss these off!

We can now finish our program, so add the last line on the end so that your complete programs looks like the following:

```
print('Welcome to our VIP program for calculating cinema ticket prices')
print('Ticket prices are £5.00 for ordinary tickets, £6.00 for VIPs')

vip = input('Do you require a VIP ticket, yes or no? ')

price = 5.0

if vip == 'yes':
    price = price * 1.2

print('Your total price is:', price)
```

See how the program carries on past the test, whether the `if` statement test was true or not - you simply have to move back 4 spaces to the left. This means our *code block* attached to the `if` statement is over, and we now carry on as usual.

Now save and run your program using the **F5** key, pressing the **Return** or **Enter** key, and using the program name of `vip.py`. Remember, to save your program in the usual location (the directory with your name, or your USB stick).

You will have to run your program twice. Initially, input *yes* as the answer to the question, and secondly, input *no* (or vice versa). You should get different results depending on what you enter on the keyboard - either a price of £6 or £5. Be careful what you type - if you do not type *yes* exactly, then the test will fail, and the indented statement will not be executed.

In a code block, you can have any number of statements, just like the program as a whole. Add onto your code block, belonging to the `if` statement, so it looks like this:

```
if vip == 'yes':
    price = price * 1.2
    print('You have chosen a VIP seat - enjoy your film!')
```

Run your program again, and notice how either both of these statement will be performed, or neither of them. They belong together in the same code block, and are attached to the `if` statement above them. You can even have blocks inside blocks. Change your `if` statement block to do the following:

```
if vip == 'yes':
    price = price * 1.2
    print('You have chosen a VIP seat - enjoy your film!')
    weekend = input('Is your viewing at the weekend, yes or no? ')
    if weekend == 'yes':
        price = price * 1.5
        print('Weekend viewing adds on another 50%, sorry!')
```

Watching films at the weekend is very expensive, 50% more expensive! You will notice, though, that this is only added on for VIP seats, as the question and the test, along with the increase in price, all live inside the test for VIP seats only. Regardless, this demonstrates that one block of code (with a certain level of indentation) can reside inside another. There is no limit to how many blocks can be inside other blocks, although if we overdo this, it will make the code harder to understand.

You can add another block of code that is selectively executed after the `if` statement above, just by starting it in the same column as the first. Add these extra three lines onto your program:

```
if vip == 'yes':
    price = price * 1.2
    print('You have chosen a VIP seat - enjoy your film!')
    weekend = input('Is your viewing at the weekend, yes or no? ')
    if weekend == 'yes':
        price = price * 1.5
        print('Weekend viewing adds on another 50%, sorry!')

popcorn = input('Would you like popcorn, yes or no? ')
if popcorn == 'yes':
    price = price + 1.25
```

You finish with the `print` statement as usual. Now run your program again - there are now six routes or paths through your `vip.py` program - firstly, whether the seat chosen is VIP or not, and within this, whether it is weekend or not, and finally whether popcorn was purchased. See if you can run the `vip.py` program, with all of these possibilities tried out.

9.1 Exercises

1. Modify your `vip.py` program so that the code to increase the price is done for both ordinary and VIP seats (i.e. move this part of the code outside the VIP block, adjusting the indentation accordingly).
2. Write a program called `kiosk.py` which prints out a menu of snacks to buy, e.g. Mars bar for 50p, Kitkat for 40p, Galaxy for 55p, Haribo for 30p (even better: make up your own items). Ask the user to type in a choice (it could even be "1", "2", "3", etc.), and print out the correct price for the item chosen.
3. Write a program called `weather.py` which asks whether it is sunny or rainy or cloudy. If the user types in "sunny", then tell the user to bring some sun cream. If the user typed in "rainy", then tell the user to bring his umbrella. If the user typed in "cloudy", then tell the user to bring his jumper. Otherwise, just ignore the response.

9.2 Things to remember

1. Use the `if` statement to perform *selection*. It is given an *expression* to *evaluate*, and if true, it will then execute the statements below the `if` statement.

2. Put a colon `:` at the end of the line containing the `if`.
3. Statements grouped together belonging to an `if` statement is called a *block of code*. It should be *indented* by 4 spaces, thus showing Python what code belongs to which test.
4. To resume the program regardless of whether the test for the `if` statement was successful or not, then you should unindent your code (push it back to the left by 4 spaces).
5. Use the *equals* operator `==` to test whether the left-hand side of the expression is equal to the right-hand side. Do not use the *assignment* operator `=` for this!

CHAPTER

TEN

DECISIONS, DECISIONS

To be, or not to be, that is the question.

— William Shakespeare

10.0 Differing ways of comparing

In the previous chapter, we introduced the `if` statement, which evaluates a test, and if true, it runs the statements which have been indented to the right following the colon `:` symbol. We call this group of statements indented to the right a *block of code*.

We also introduced our first comparison operator, the equals sign `==`. There are many more, most of which you will recognise from maths. We list the possibilities at the end of this chapter, but here are some highlights.

We will use the interactive interpreter to experiment for a while. Type the following for practice:

```
>>> a = 10
>>> b = 20
>>> a == b
False
>>> a != b
True
>>> a < b
True
>>> a <= b
True
>>> a > b
False
>>> a >= b
False
```

Then try changing the values of `a` and `b` (make them the same, for instance), and repeat the comparisons.

From the above snippet, the first two lines create our variables, `a` and `b`, using the *assignment operator* (not the *equals operator*). The next six lines demonstrate the different comparison operators, one by one. The only one that really requires explanation is the second one, the *not equal to operator* `!=`. In maths, you would use a different symbol, such as this one: $\neq$. Since this symbol is not readily available on a computer keyboard, we use the exclamation mark in front of the equals sign to make the not equals operator `!=`. The other symbols are also separated, so in maths you could use the $\geq$ symbol, in programming, we expand it into `>=`, thus taking up two symbols to mean the same. Similarly with $\leq$ is turned into `<=`.

Note: We call these *expressions*, as they express a value once properly evaluated by the computer. Normally, one value is on the left, and another on the right, with the operator in the middle. Each of these operators can be used with other types of data than just integer numbers - floats and strings could also be used, e.g. `2.5 > 1.2` and `"fred" != "harry"`.

You will also notice that the result of each expression is either `True` or `False`. This is our fourth type of data, and they are called *boolean* values. They are the simplest type of data imaginable - they are either `True` or `False`. They cannot be any other value. This is the equivalent of on or off, 1 or 0, up or down. There is no in-between value, it is one or the other. For example, above we tested whether `10` was equal to `20` (`a == b`) - this is either `True` or it is `False` (obviously the latter). It cannot be something else, or both!

Booleans are very useful, and you can set variables to boolean values as well. We will see more of this in *chapter 13* on escaping out of a loop.

10.1 More than one possibility

Testing just with the `if` statement is very useful, but Python does offer more flexibility than just this. For example, if we want to write a program that inputs two numbers, and prints out the largest number, we could do the following. So open up a new file window, and type the following:

```
num1_str = input('Enter your first number: ')
num2_str = input('Enter your second number: ')

num1 = int(num1_str)
num2 = int(num2_str)

if num1 > num2:
    print('The largest number is:', num1)
if num2 > num1:
    print('The largest number is:', num2)
```

Run it with the F5 key, and save it onto your USB stick with the name `nums.py`.

It should run fine, printing out the largest number of the two input by the user. Note how we have to convert the inputs, entered as a sequence of digits, into integer numbers before we can use them in the comparisons. Otherwise, the comparisons will not work correctly.

The problem is that we are doing the test twice, which is wasteful. It is more efficient to do the test once, and do one block of code if the test was successful (i.e. it was true), and a different block of code if unsuccessful (i.e. the result was false). This is where the `else` statement comes into play. What is does it run a block of code if the `if` statement above it failed in all its tests. To see this in action, change your last-but-one line to read as follows:

```
if num1 > num2:
    print('The largest number is:', num1)
else:
    print('The largest number is:', num2)
```

Don't forget those colon `:` symbols at the end of the lines above each new code block! Remember, a block of code can be as little as a single statement, or hundreds of lines long. It depends on what you want to do.

Now run your program again - it should do exactly the same, but more efficiently (i.e. faster) this time.

Again, the `else` presents a block of code to be run if all of the tests in the `if` statement above it have failed. It is like a safety net at the bottom of the `if` statement which takes care of everything if none of the tests are true. Read the word *else* like the word *otherwise* if that helps - test this and do this if true, otherwise do this.

10.2 Many, many possibilities!

What if we had more than two possibilities - doing one thing for the main test, and another thing for everything else? Python has you covered for this eventuality as well - a combination of the `if` and the `else` put together to form the `elif` - short for *else if*.

The `elif` statement is placed after the `if` (so there must always be an `if` statement first), and there can be as many `elif` statements as you need. Each one has an expression to evaluate, and if true, then the `elif` code block is run.

For example, modify your `if` and `else` statement in your `nums.py` program above, so that it now states the following:

```
if num1 > num2:
    print('The largest number is:', num1)
elif num1 == num2:
    print('The numbers are the same!')
else:
    print('The largest number is:', num2)
```

We introduced the middle two lines, the `elif` followed by the call to the `print` function. What this does is test the `elif` condition only if the test for the `if` fails. If the `elif` test succeeds (it is true), then the line - or lines - under the `elif` statement are run.

10.3 A bit more practice

Combining what we have learnt in this chapter, let us write another program called `noises.py`. Start it in the usual method of clicking on the **File** menu and selecting **New File**. Once the new blank window has appeared, type in the following:

```
animal = input('What animal do you have there with you? ')
if animal == 'cow':
    print('Moo!')
elif animal == 'sheep':
    print('Baa!')
elif animal == 'pig':
    print('Oink!')
elif animal == 'horse':
    print('Neigh!')
elif animal == 'chicken':
    print('Cluck!')
elif animal == 'duck':
    print('Quack!')
elif animal == 'dog':
    print('Woof')
elif animal == 'cat':
    print('Meow!')
elif animal == 'dinosaur':
    print('Roar!')
else:
    print('Sorry, I don\'t recognise that animal!')
```

Obviously, we could go on and on! Save it using the **F5** key, name it `noises.py` making sure you save it in the usual location, and run it. You need to run it several times in order to test all the possibilities (i.e. see all the different noise words being printed out).

As you can see, the `if` statement is tested first. If the test evaluates to true, then the first optional block of code is run (printing out 'Moo!'), and it will then jump to the end (past the `else`). Otherwise, it will test each test in turn, only running the code blocks if the test is true. Otherwise, it will eventually drop down to the `else` statement, and run the last block of code, but only if all the other tests have failed.

10.4 Exercises

1. Modify your `kiosk.py` program you wrote for the previous chapter so that instead of using lots of `if` statements, you use one `if` statement, followed by a number of `elif` statement. The `else` statement should be used to print out a message telling the user that he has not entered a valid choice.

2. Write a program called `move.py`, and ask the user the form of transport, either a plane, car, bicycle or walking. Depending on what they have entered, print out ‘fast’, ‘quick’, ‘steady’ or ‘slow’.
3. Write a program that uses the `turtle` module called `shapes.py`. Ask the user what shape to draw, e.g. circle, square or star. Depending on what the user has entered, draw the appropriate shape. If the user didn’t type in anything sensible, then print out an error message.

10.5 Things to remember

1. Remember your comparison operators, using the variable `a` (an integer) as an example:

Name	Operator	Example	Maths equivalent
Equals	`==`	`a == 10`	$a = 10$
Not equal to	`!=`	`a != 10`	$a \neq 10$
Greater than	`>`	`a > 10`	$a > 10$
Greater than or equal to	`>=`	`a >= 10`	$a \geq 10$
Less than	`<`	`a < 10`	$a < 10$
Less than or equal to	`<=`	`a <= 10`	$a \leq 10$

2. We now know four types of data - integers, floats, strings and booleans. *Boolean values* are either `True` or `False`.
3. Each selection statement must contain an `if` statement, along with a test to evaluate and at least one line of code to run, indented to the right. If the test is evaluated as true, then even if there are `elif` or `else` statements below, the program will skip them.
4. You can optionally include one or more `elif` statements, each with their own tests to evaluate and their own blocks of code. If more than one of these is evaluated as true, then the first one is run, and the others are skipped.
5. Finally, you can also optionally include an `else` statement, without any test, but with its own block of code to run. This block of code is only run if the `if` and `elif` tests all fail (i.e. are all false).

CHAPTER

ELEVEN

COMBINING DECISIONS TOGETHER

The best thing about a boolean is even if you are wrong,
you are only off by a bit.

— Anonymous

11.0 Juggling tests

We have learned how to use the `if` statement to evaluate a test, and if true it then executes a block of code you provide immediately underneath. It makes our programs cleverer so that they can take different decisions depending on the circumstances.

This chapter introduces *logical operators* which make combining tests together easier. Although this part of programming is not essential, it does make our programs shorter and easier to write.

For example, take a program that wants to tell the user whether he can go the beach or not. This decision depends on the weather (preferably warm), and whether it is during the holidays or not.

With what we know now, we could code this as follows, so type this into a new file window:

```
weather = input('How is the weather, warm or cold? ')
holidays = input('Are we on holiday at the moment, yes or no? ')

if weather == 'warm':
    if holidays == 'yes':
        print('Yes! Let\'s go to the beach!')
    else:
        print('Sorry, let\'s try again later!')
else:
    print('Sorry, let\'s try again later!')
```

Save this program and call it `beach.py`. Run it and test it out - it should all work fine, regardless of the combination of values you type in.

However, this took a lot of typing to get working, and programmers like to avoid typing when they can! One problem is that the 'Sorry' message is done twice, so we are duplicating a line unnecessarily. We also have two `else` statements to go with the two `if` statements. It would be much better if we could combine the two tests together on one line, with one `if` statement and one `else` statement, with each call to the `print` function done once each as well.

So far, we have only learned how to attach one test to either an `if` statement or an `elif` statement. With *logical operators*, we can combine a number of tests together to form a larger single test. There are three logical operators:

- `and` which tests whether the left-hand side and the right-hand side are both true, giving an overall result of true. For example: `a > 10 and b > 10` tests whether `a` and `b` are both greater than 10, and if so, the whole expression is true.
- `or` which tests whether either the left-hand side or the right-hand side are true, thus giving a overall result of true. For example, `a < 0 or a > 100` tests whether `a` is either less than zero (i.e. negative) or greater than 100 (but obviously not both at the same time), and if so the whole expression is true. Both sides can also be true, which is also fine.
- `not` which takes a single boolean value and inverts its value, so `True` becomes `False` and `False` becomes `True`. For example, `not weather == 'warm'` which tests whether the variable weather is equal to the value `'warm'`, and then flips the result.

Note: Just like with arithmetic operators, the `and` and the `or` logical operators need two values (formally called operands), one to the left and another to the right. For example, you could write `(a > 10) and (b < 20)` or `(a == 10) or (b == 10)`. And similar to the negative operator `-`, the `not` operator only needs one, the value it is inverting, such as `not (a == 10)`.

So, how do we apply this to our code in the `beach.py` program? To see this, you need to change the big `if` statement from this:

```
if weather == 'warm':
    if holidays == 'yes':
        print('Yes! Let\'s go to the beach!')
    else:
        print('Sorry, let\'s try again later!')
else:
    print('Sorry, let\'s try again later!')
```

to this:

```
if weather == 'warm' and holidays == 'yes':
    print('Yes! Let\'s go to the beach!')
else:
    print('Sorry, let\'s try again later!')
```

Save and run this version, and make sure it does the same thing.

Notice how we have taken the two separate `if` statements in the previous version of the program, and combined them together - since one was inside the other - with the logical `and` operator. This will then only perform the first call to the `print` function if both the weather is 'warm' *and* and holidays is 'yes'. Otherwise, we do what comes after the **`else`** statement.

The logical **`or`** operator is useful when a number of separate tests all do the same thing, so their blocks of code are all the same.

For example, create a number program called `numbers.py`, and type in the following:

```
ticket1 = input('Enter ticket number 1: ')
ticket2 = input('Enter ticket number 2: ')
ticket3 = input('Enter ticket number 3: ')

prize = input('What is the prize number? ')

if ticket1 == prize or ticket2 == prize or ticket3 == prize:
    print('We won the prize')
else:
    print('Nevermind, maybe next time!')
```

Without the use of the **`or`** operator, we would have to have written 3 **`if`** statements, all doing the same thing. The **`or`** operator has allowed us to combine these 3 tests into one, thus saving on lots of typing and duplication in our code, which is never a good thing.

11.1 Exercises

1. Write a program called `largest.py` to input three numbers, convert the inputs from strings to integers, and print out the largest. Use the **`if`** and **`elif`** statements and the **`and`** operator to perform your tests.
2. Write another program called `car.py` to ask the user the attributes of a car, such as color (e.g. `'red'`, `'green'` or `'blue'`), type (e.g. `'van'`, `'sports'`, `'estate'`) and price. The program should print out `'I want that car'` if the color is `'red'`, the type is `'sports'` and the price is less than 10,000.

11.2 Things to remember

1. There are three *logical operators*: **`and`** for testing whether the left and right-hand side tests are both true; **`or`** for testing whether either the left or right-hand side tests are true (or both); **`not`** for inverting a boolean value.
2. Put the **`and`** and **`or`** operators in-between boolean expressions. Put the **`not`** operator in front of a boolean expression.

CHAPTER

TWELVE

GOING LOOPY

Q. How did the programmer die in the shower?

A. He read the shampoo bottle instructions: Lather. Rinse. Repeat.

— Anonymous

Computers are very good at doing a number of steps repetitively, relieving us of many mundane tasks in our lives. In programming, this makes our programs much more flexible and involves a lot less typing which is always a good thing!

Imagine drawing a hexagon (a 6 sided polygon) using turtle. There is the hard way, and the easy way.

12.0 Doing it the hard way

With what we have learned up until now, we would do the following. So open a new file window, and this in:

```
import turtle

turtle.forward(100)
turtle.left(60)
turtle.forward(100)
turtle.left(60)
turtle.forward(100)
turtle.left(60)
turtle.forward(100)
turtle.left(60)
turtle.forward(100)
turtle.left(60)
turtle.forward(100)
turtle.left(60)
```

Run it - calling the program `hexagon.py` - and check that it draws a hexagon in the turtle window.

This program is 14 lines long (not including the blank lines), with 12 of those lines just to draw the hexagon. Not a great return!

Clearly, this is too much typing for what it does. Imagine the typing involved in drawing a 100 sided polygon! There are also cases where this kind of sequential programming (i.e. programming that executes - or performs - each line, one after another) simply cannot do what we want. If you do not know in advance how many times to do a particular task, then you cannot do it sequentially. For example, if the program is adding up a series of numbers, one after the other, and is relying on the user to press the `=` equals sign at the end (or perhaps typing "stop" when using a program), then the program cannot calculate in advance how times to repeat the code to input the next number. We have to find a different way.

12.1 Doing it the easy way

To overcome this hurdle, Python - and pretty much every other programming language - has the ability to repeat a block of code a number of times. This is called looping or *iteration*.

Roughly speaking, looping is similar to the way we construct `if` statements. In an `if` statement, we use the `if` keyword (a *keyword* is a word reserved by Python and given a special meaning), following by a test which results in a boolean value - either `True` or `False`. It will then conditionally execute a given code block, which follows the colon `:` symbol. If the test fails (results in a value of `False`), then the code block is skipped. Either way, the program continues on its way after the `if`, along with any `elif` and `else` statements are done.

A loop uses a different keyword, but still has a condition that is tested, and also a code block that belongs to it. The main difference is that a loop will execute the code block not just once, but potentially many times. To be precise, the code block can execute zero or more times, depending on whether the test at the top of the loop is `True` in the first place! The loop will repeat whilst the condition remains `True`, so clearly we need a way of changing the condition as we go along or otherwise it will repeat forever. In the next chapter, we will do this on purpose, but with a way of escaping!

Enough talking - let us go about changing our program above to get the computer to do more of the work! Add the following lines onto your program (do not bother modifying your existing code, as it will be a good point of comparison):

```
side = 0
while side < 6:
    turtle.forward(100)
    turtle.left(60)
    side = side + 1
```

Run your program, again saving it in the usual location, and see what it does. It should now draw the same hexagon twice!

The new thing to learn here is the use of the new keyword `while`. With this, the program performs the indented block of code, following the `:` symbol, *while* the condition is true.

Initially, the condition is `True`, as we have set an integer variable to zero, and zero is less than 6. Each time we draw a side of the hexagon, we add (*increment*) one onto the side variable. So the variable side goes from 0 to 1 to 2, 3, 4, 5 and finally 6. We then test whether the value 6 is less than 6, which it is not - it is equal to 6. Therefore, the condition is then `False`, and the loop stops.

You can see we have reduced 12 lines to draw a hexagon, down to 5. The great thing is that if we change the condition from 6 to 100 (and changing the angle of turning left would be good too, as well as the length of each side), then we would then draw a 100 sided shape without any further changes. In the hard way above, this would involve another 188 lines of code!

12.2 Exercises

1. Write a program called `hundred.py` that prints out the numbers from 0 to 100, inclusive.
2. Modify your `hundred.py` program so that after counting up to a 100, it then counts down from 100 to 0, printing as it goes.
3. Modify your `hundred.py` program so that the loops count up or down in steps of 5, not 1. Run it again to check that it works as expected.
4. Write a program using the `turtle` module that asks the user how many sides to draw, converts it into an integer, and then uses turtle to draw a shape with that number of sides. You will have to work out the angle by dividing 360 degrees by the number of sides when turning the turtle to the left or right.

12.3 Things to remember

1. Use the `while` keyword to repeat a block of code.
2. The condition used after the `while` keyword is just like when using an `if` statement - a boolean expression. It can use any of the comparison and logical operators.
3. The code block to be repeated can contain 1 or many lines of code. It all depends on what you want to do. It can even contain other loops nested within the outer loop.

CHAPTER

THIRTEEN

ESCAPING THE CYCLE

Have you heard about the new Cray super computer? It's so fast, it executes an infinite loop in 6 seconds.

— Anonymous

In the previous chapter we posed a problem that a sequential way of programming cannot solve - needing to repeat a block or set of instructions a number of times that is unknown in advance. In other words, you have to repeat a set of statements, but you only know when to stop at the very end. It is not possible to code this in a top-down or sequential way. However, we can achieve this by using loops in our programs.

13.0 Breaking out

Although we have introduced looping, we have only used it by counting from a starting number to an end number, and then stopping. We really just want to repeat a block of code a certain number of times. But what, like our problem above, you don't know when to stop until the end?

Remember that the test after the `while` keyword is just a boolean expression. If it evaluates to a `True` value, then the loop should perform another cycle of its block of code. It will then test the expression again to see if it has changed in the meantime. So if we don't know when to stop, a good start is to make the loop go round and round indefinitely. It is a called an *infinite loop*, and - in theory at least - it goes round forever! And to make an infinite loop, we simply make the boolean expression `True` by using the value `True`.

To see this in action, start a new file window and type in the following:

```
while True:
    print('Help, I\'m stuck in a loop!')
```

Save it as `adder.py`, and see what happens. It should keep printing out the *Help* message. To stop the program, you need to press the `Ctrl` and `C` (same shortcut as copy) keys together - this breaks out of the loop. Alternatively, you can select the "Restart Shell" item from the "Shell" menu at the top of the IDLE program.

Clearly, we need a better way of breaking out of the loop than relying on the user to do it for us. This is where the `break` keyword comes in, combined with what we know already about the `while` loop and the `if` statement to make a selection.

Therefore, modify your `adder.py` program like so:

```
while True:
    name = input('What is your name, or type stop to quit: ')
    if name == 'stop':
        break
    print('Hello there', name)
```

Save and run it again and see what happens. Notice how the `while` statement is the same, but inside the loop it is very different. The first line of the loop code block simply asks for the user's name, using the `input` function, and stores it in a variable called `name`. Then we do something new - we test whether the contents of the variable `name` is equal to the value 'stop' (which we have told the user to type in to quit the loop), and if so, we use the new keyword `break` to break out of the loop. It simply jumps passed the end of the code block, attached to the `while` loop, to carry on with the rest of the program (if there is any). If we didn't break out of the loop, then we print a message to the user, using the contents of the variable `name` to do so.

We could also do this by using a boolean variable in a slightly different way:

```
keep_going = True
while keep_going:
    name = input('What is your name, or type stop to quit: ')
    if name == 'stop':
        keep_going = False
    else:
        print('Hello there', name)
```

It is slightly longer, but is a useful technique to use in certain situations.

13.1 Adding up

Let us change our program a third time to finish with a program that will ask the user for one number at a time, adding them onto the total as it goes along. The user can gain type 'stop' to break out of the loop, but this time, it will print out the total at the end.

Therefore, modify your `while` loop to reflect the following:

```
total = 0
while True:
    num = input('Enter a number, or type stop to quit: ')
    if num == 'stop':
        break
    total = total + int(num)
print('The grand total is:', total)
```

Notice how creating the variable of `total` with a value of 0 is outside of the `while` loop code block, as is the call to the `print` function at the end. This is determined by those lines of code being vertically aligned with the `while` statement, and not with the code block beneath the header of the loop. The four middle lines form the code block which is repeated, potentially forever. What stops the loop is the user typing the word 'stop' into the variable `num`. This then means the test belonging to the `if` statement is True, so the break is then run. If the user does not type 'stop', then the number is converted into an integer using the `int` function, and added onto the running total. The loop then repeats until the user does type 'stop', and then print function finally does its bit.

Note: The `break` keyword will break out of your present loop. There is another keyword called `continue` which will stop executing the code block and continue the loop from the beginning again. This is a way of skipping any remaining lines in the loop and starting the next loop early.

13.2 Exercises

1. Change your `adder.py` so that the user types 'quit' instead of 'stop' to break out of the loop.
2. Write a program called `words.py` which inputs a word at a time, appends it onto a string (e.g. `sentence = sentence + word`), and prints it out at the end.
3. Modify your `polygon.py` program from the previous chapter so that it keeps drawing polygons, one on top of the other, until the user types 'stop'. Each time it will ask the user how many sides to draw, just as before.

13.3 Things to remember

1. Use the value of `True` as the expression - or test - for the `while` loop to make it go on indefinitely, i.e. an *infinite loop*.
2. Use the keyword `break` to break out of the present loop. If one loop is nested inside another and the break resides in the inner loop, it only breaks out of the inner loop, not the outer one as well.
3. The `while` loop is best used for this kind of looping - when you do not know when to stop until you have reached the end. In *chapter 17*, we will introduce another kind of loop which is better for looping a set number of times.

CHAPTER

FOURTEEN

GOING RANDOM

The generation of random numbers is too important to be left to chance.

— Robert R. Coveyou

14.0 A bit of variation

To add variation to our programs, we can ask the user for some input, and then behave differently depending on what the user has typed in. But what if we wanted to play a game, with the game needing to change its behaviour from one run to the next? If a game we played behaved exactly the same way every time, which computers tend to do, games soon become rather predictable! A game of chess where the computer always started in the same way would never sell very well!

Computer programs which need a bit of variation are programs like flight simulators (where the weather differs from time to time), board games (so the moves vary from game to game), racing games (where your competitors do different things from race to race), and so on. They do this by using a piece of software to provide them with some *random* data, such as a simple number which varies between a range of numbers. This simple piece of data is then used to vary the decisions made within the program, thus making the program more interesting to use. In a way, it becomes a little more like the real world.

14.1 That's random

Python does this by using the `random` module. We will get to know this by first using the interactive interpreter - bring this onto your screen, and you can then import the `random` module in a similar fashion to importing the `turtle` module:

```
>>> import random
```

We can list what functions the random module offers by using the `dir` function:

```
>>> dir(random)
```

We can then experiment with some of its functions. Try calling the `random` module's `random` function (random module, random function), a few times:

```
>>> random.random()
>>> random.random()
>>> random.random()
```

Note: You can repeat a command in the interactive interpreter by using the up arrow key on your keyboard, and pressing the `Return` or `Enter` key. This brings the statement down onto your current line, allowing you change it. Execute that statement by pressing the `Return` or `Enter` key again.

Run this line a few time with the note above. See how it always returns a number that is between 0.0 and 1.0, but hardly ever the same exact number. See if you can get it to repeat a number - it is not easy!

In fact, let's use our new found knowledge on looping to see how this random number changes every time you ask for it. So type the following into the interactive interpreter:

```
while True:
    random.random()
```

Just like in the previous chapter on infinite loops, you will have to press the `Ctrl` and `C` keys together on your keyboard to break out of the loop manually. It will probably go so fast, that only by breaking out of the loop will you be able to look at the numbers properly. If this doesn't work, then make sure you have imported the `random` module first.

Let's try another function - `randrange` - this time, instead of returning a number between 0.0 and 1.0 (which could be scaled up, if needs be), it will return an integer up to (but not including) the number you give it. For example, try the following a few times:

```
>>> random.randrange(100)
>>> random.randrange(100)
>>> random.randrange(100)
```

And try different numbers too:

```
>>> random.randrange(10)
>>> random.randrange(50)
>>> random.randrange(25)
```

Try any end number you like, although the number must be above zero. You can also give it a start number as a first parameter, so try these or other numbers as you wish:

```
>>> random.randrange(10, 20)
>>> random.randrange(50, 100)
>>> random.randrange(1000, 2000)
```

The number returned is always between the numbers you give, including the start number, but excluding the end number.

14.2 A guessing game

Now we can use this knowledge to construct a simple game, where the program comes up with a random number, and the user has to guess it. We'll give the user 6 tries until we give the answer. So call your program `guess.py`, and type in the following for starters:

```
import random

number_to_guess = random.randrange(1, 101)
```

We have imported the random module, in order to use it within our `guess.py` program, and asked for a random number between 1 and 101 (1 and 100, inclusive, not including 101) and stored it against a variable name `number_to_guess`. Now we add the loop to give the user 6 tries at guessing, so add the following:

```
num_tries = 0
while num_tries < 6:
    num_tries = num_tries + 1
```

We define a variable `num_tries`, and initially set it to zero. We then loop while this value is less than six (so it should loop over the values 0, 1, 2, 3, 4 and 5 - six numbers in total), adding 1 onto the `num_tries` variable each time.

Inside the loop, we can add these lines (only add the new lines!):

```
num_tries = 0
while num_tries < 6:
    user_guess = int(input('Guess the number: '))
    if user_guess == number_to_guess:
        print('Well done - you guessed right!')
        break
    num_tries = num_tries + 1
```

We ask the user a question, input what they have typed, and convert it into an integer storing it against a variable name called `user_guess`. If this variable is equal to the value the computer stored initially, then we print a message and then break out of the loop. Otherwise we carry on by adding one onto the `num_tries` variable, and go back up to the top of the loop to repeat.

You can then finalise your program by giving the answer at the end, after the loop has finished. So, in total, your program should look like this:

```
import random

number_to_guess = random.randrange(1, 101)

num_tries = 0
while num_tries < 6:
    user_guess = int(input('Guess the number: '))
```

Cont...

```
    if user_guess == number_to_guess:
        print('Well done - you guessed right!')
        break
    num_tries = num_tries + 1

print('The answer was:', number_to_guess)
```

You could enclose the `print` at the end with a test to only display it if the `num_tries` is 6, as if the user did guess the number they don't really need to be told what it was.

14.3 Exercises

1. Modify your `guess.py` program so that after testing whether the `user_guess` variable is equal to the computer's number `number_to_guess`, the program will then test whether the user's number is less than the computer's number and print an appropriate message (e.g. 'Too low!'), and also if the user's number is larger than the computer's number, then print out another message (e.g. 'Too high!'). This will give the user a hint as to which direction to head in!
2. Write a program called `poly.py` to randomly choose how many sides a polygon should have (e.g. between 3 and 12), and then draw the appropriate polygon. So if the `randrange` function returns 3, then a triangle is drawn, or if it returns 8, an octagon is drawn.

14.4 Things to remember

1. To add variation, or a bit of *randomness*, into your program, then `import` the `random` module, and make use of what it offers.
2. Two functions we used in this chapter are the `random` function (note: it has the same name as the module), which returns a floating point number between 0.0 and 1.0, and `randrange` which returns an integer number between 0 (or the starting point you provide) and up to (but not including) the end point.

CHAPTER

FIFTEEN

GROUPING DATA TOGETHER

A computer is a stupid machine with the ability to do incredibly smart things, while computer programmers are smart people with the ability to do incredibly stupid things. They are, in short, a perfect match.

— Bill Bryson

15.0 Scattered data

We have already covered variables, which simply attaches a name (like a label) to a piece of data. This is very handy, and allows us to give data a memorable name we can refer to and modify later. Technically, variable names (identifiers) are memory addresses that point to the data we have stored - it is like a signpost pointing at the data.

However, it can become messy when there are many variables in our programs, especially if they are closely related. For example, if we had a bunch of people's names, we could define each name individually. Use the interactive interpreter to type in the following:

```
>>> name1 = 'fred'
>>> name2 = 'bob'
>>> name3 = 'harry'
>>> name4 = 'tom'
```

All the variables here describe the same kind of data - a group of names, one after the other. It would be good if we could simply group these items together under a single name. This is usually very good practice in programming - those things (whether code or data) that belong together should be kept together.

15.1 Boxing up

The way we do this in Python is by using *lists*. As its name suggests, a list is simply a sequence of other pieces of data, whether integers, floats, strings or even other lists.

Remember, if we have more than one item to print out, we simply use a comma `,` in-between each item. This is easy to forget! For example, if we wish to print out our names above, you can type the following:

```
>>> print(name1, name2, name3, name4)
fred bob harry tom
```

We do the same in lists to separate each item. To define a list, to group a sequence of items together, we simply use brackets - square ones. It is like a box grouping the values together. So, for example, let's group together the names we defined earlier:

```
>>> names = ['fred', 'bob', 'harry', 'tom']
```

Notice how we have done away with the individual variables, e.g. `name1`, `name2`, etc., and now only have one name, `names`. This means all four values are referred to by the same variable name - the entire list is given a single name. You can print out the entire list in one go as well:

```
>>> print(names)
['fred', 'bob', 'harry', 'tom']
```

This list is a sequence containing four strings - the names `'fred'`, `'bob'`, `'harry'` and `'tom'`. As mentioned above, lists can contain almost anything, so let's try a list containing different data:

```
>>> my_ints = [1, 2, 3, 4, 5]
>>> my_floats = [2.5, 17.2, -1.7, 123.9]
>>> my_lists = [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
```

Notice how the contents of the list can be in any order you wish, although it can be sorted later. The last one is slightly complicated - it is a list containing four sub-lists, each of which contain three integers. A list of lists, or sometimes referred to as a *matrix*. The data can even be mixed, although this does not always make sense. Try this:

```
>>> my_stuff = ['bacon', 123, 99.5, [1, 44.2, 'fred']]
```

Again, you can print these list out using the `print` function. Try it with each list in turn.

15.2 Playing with lists

We can do all sorts of things with lists in Python that can turn out to be very useful. To get Python to sum up a group of numbers, we can just put them in a list and use the `sum` function:

```
>>> numbers = [1, 5, 50, 23, 77]
>>> sum(numbers)
156
```

And other functions, `min` and `max`, will give the minimum and maximum value out of a list:

```
>>> numbers = [10, 123, 40, 89, 65]
>>> min(numbers)
10                                                            Cont...
```

```
>>> max(numbers)
123
```

The `len` function gives us the number of items in the list:

```
>>> len(numbers)
5
```

You can even get Python to sort a list in any order you wish. Try this out:

```
>>> numbers = [15, 5, 35, 10, 25, 20, 30]
>>> print(sorted(numbers))
[5, 10, 15, 20, 25, 30, 35]
>>> print(sorted(numbers, reverse=True))
[35, 30, 25, 20, 15, 10, 5]
```

The first print statement prints out the numbers in ascending order. The second prints out the numbers in descending (i.e. reverse) order. Notice how we call `sorted` function initially with one argument (i.e. the list to sort), and secondly, we call the same function but with an extra argument we refer to by name - `reverse`, which we set to `True`. We will cover this kind of argument passing in *chapter 19* when we are creating our own functions.

Note: We have described how to use functions built into Python itself by typing its name, followed by parentheses with any parameters enclosed between the parentheses. For example, `print()`, `len('Yo man')`, `min(10, 20, 30)`. We have also described how to use functions that have been defined in other modules by typing the module name followed by a period `.` For example, `random.random()`, `math.sqrt(81)`, `turtle.circle(100, 180)`. Below, for the first time, we will also call functions that belong to a particular type. These operate on the data referred to by the variable name or the value itself. It effectively wraps the value and the functions that work on that value together, so the `int` type contains functions that work on integers, the `str` type contains functions that work on strings, the `list` type contains functions that work on lists, etc. For example, if using a string `name` we could type `name.upper()` to change it to uppercase (i.e. capitals), `name.find('fred')` to find a string within the name, `name.split()` to split the string into a list of words. If you intend to change the value itself, the function will either return a new value or will change the value in-place, depending on the data type. To find out what type functions (also known as *class methods*) are available, you just type `dir(str)` or `dir(list)` depending on the data type you are working with. See the *chapter 5* for more on getting help, and *Appendix B.3* for more on functions.

Lists can also be modified after they have been defined by using the `append` and `insert` functions on the list variable itself. The `append` function adds a new item onto the end of the list; `insert` adds an item into the list (so you need to give a position as well); `remove` removes the named item from the list (the first one if more than one exists). To see how this works, try out the following:

```
>>> names = ['Bilbo', 'Frodo', 'Sam']
>>> print(names)
```

Cont...

```
['Bilbo', 'Frodo', 'Sam']
>>> names.append('Pippin')
>>> print(names)
['Bilbo', 'Frodo', 'Sam', 'Pippin']
>>> names.insert(0, 'Merry')
>>> print(names)
['Merry', 'Bilbo', 'Frodo', 'Sam', 'Pippin']
>>> names.remove('Bilbo')
>>> print(names)
['Merry', 'Frodo', 'Sam', 'Pippin']
```

As described by the note above, we are using type functions for the first time. These functions are called by using the variable name (or even just the value itself), followed by a period `.` character, and then the function name, similar to calling a function inside a module that has been imported. This means that the code `names.append('Pippin')` is simply shorthand for `list.append(names, 'Pippin')`, and the code `names.insert(0, 'Merry')` is shorthand for `list.insert(names, 0, 'Merry')`, and finally the code `names.remove('Bilbo')` is shorthand for `list.remove(names, 'Bilbo')`.

We can split a sentence into a list of words using the `split` function:

```
>>> sentence = 'Mary had a little lamb'
>>> sentence.split()
['Mary', 'had', 'a', 'little', 'lamb']
```

Here we are using a function belonging to the `str` (string) type, so the code `sentence.split()` is shorthand for `str.split(sentence)`.

We can also find out whether a value is a member of a list (i.e. is contained within the list) by using the `in` operator. Try this out:

```
>>> numbers = [1, 2, 3, 4, 5]
>>> 3 in numbers
True
>>> 10 in numbers
False
>>>
>>> sentence = 'jack and jill ran up the hill'.split()
>>> 'jack' in sentence
True
>>> 'jill' in sentence
True
>>> 'bob' in sentence
False
>>>
>>> breakfast = ['porrige', 'toast', 'coffee', 'juice', 'bacon', 'egg']
>>> 'waffle' in breakfast
False
>>> 'pancake' in breakfast                                        Cont...
```

```
False
>>> 'egg' in breakfast
True
```

All of these things would have been much harder to do by ourselves - Python is great at helping out in this way. There is so much to what Python offers, but here we have at least given a brief overview.

We will learn how to dip into a list to fetch individual items (or a section of the list) in the next chapter. We learn how to step over a list, one item at a time, in the chapter after next. All this will make our programs easier to write.

15.3 Rock, Paper, Scissors

Let's write a rock, paper, scissors game, where you play versus the computer. The computer will use the random module we learnt in the previous chapter to choose one option out of either rock, paper or scissors, and we will make our own choice. The program will compare the choices, and declare a winner (or possibly, a draw).

Start by opening up a new file window, and type in the following:

```
import random

choices = ['rock', 'paper', 'scissors']

while True:
    user_choice = input('Enter your choice, rock, paper'
                        ' or scissors (or stop to quit): ')
    if user_choice == 'stop':
        break
```

This should be fairly familiar to you now. We are importing the `random` module, and we have a loop which repeats until the user breaks out by entering the word 'stop'. You can keep the question inside the `input` function on the same line.

The new bit is the second line, where we define a list with the possible choices contained (boxed up) in a list, which we call `choices`. It is a list of strings. We can then add onto our program (watch the indentation, these lines are *within* the while loop):

```
computer_choice = random.choice(choices)
```

This is also new - we are using a different function from the random module called `choice`. It takes a list, and returns back one item selected in a random fashion from that list. We now have both choices needed in order to declare a winner - remember what the rules of the game are:

- if they are the same, it is a draw

- rock beats scissors
- scissors beats paper
- paper beats rock.

So we can now finish:

```
if user_choice == computer_choice:
    print('Draw!')
elif ((user_choice == 'rock' and computer_choice == 'scissors') or
      (user_choice == 'scissors' and computer_choice == 'paper') or
      (user_choice == 'paper' and computer_choice == 'rock')):
    print('You won!')
else:
    print('Computer won!')
```

And that's it! Now save and run your program calling it `rockpaperscissors.py` (or `rps.py` if you like).

Notice how we can split a large test (the one attached to the `elif` testing whether the user has won or not) over 3 lines to make it more readable by putting parentheses around the entire expression. Otherwise Python will complain.

There is one problem with our program - if the user does not type in exactly either 'rock or 'paper' or 'scissors', then the computer always wins. Look at the tests - it fails the first test (the two choices cannot be equal), and it also fails the second test (as the `user_choice` is none of the possible values provided). This is where we can use the `in` operator we described above.

Therefore, modify the long `if` statement in your `rps.py` program so it includes the new lines below:

```
if user_choice not in choices:
    print('Not a valid choice, please try again.')
elif user_choice == computer_choice:
    print('Draw!')
elif ((user_choice == 'rock' and computer_choice == 'scissors') or
      (user_choice == 'scissors' and computer_choice == 'paper') or
      (user_choice == 'paper' and computer_choice == 'rock')):
    print('You won!')
else:
    print('Computer won!')
```

Careful - you need to enter the first two new lines, and also change the existing `if` to an `elif` - otherwise, there would be two independent `if` statements instead of a series of tests following on from one another.

Now try our your program again. It should behave itself whatever the user types in.

15.4 Exercises

1. Modify your `rps.py` program so that it prints out what the choices were, particularly the computer choice. It is nice for the user to know how they won or lost a game!
2. Write a program called `sizes.py` to use the `turtle` module to draw a shape (e.g. a circle) with a fill color randomly selected. You could define your colors such as `colors = ['red', 'green', 'blue', 'magenta', 'cyan', 'yellow']`, and use the `random.choice` function to choose between them, passing the result into `turtle.fillcolor` function. Don't forget to call `turtle.begin_fill` and `turtle.end_fill` before and after drawing your shape, respectively!

15.5 Things to remember

1. To group a number of items together we box them together using square brackets, with an opening bracket `[` at the beginning, and a closing bracket `]` at the end. This creates a *list*.
2. Separate each item within the list using a comma.
3. Use the `len` function to find out how long a sequence is.
4. Use the `in` operator to test whether a value is contained by the list.
5. Use the `choice` function from the `random` module to select one item, chosen in a random fashion, from a list of possible items.
6. Every value or variable in Python belongs to a type (e.g. str, int, list), and every type has a number of functions that operate on the data it contains. Use the `variable.function_name` notation to invoke a type function, just like invoking a function inside a module.
7. We now know five types of data - integers, floats, strings, booleans and lists. Lists can contain any of the other types of data, including sub-lists!

CHAPTER

SIXTEEN

SLICING SEQUENCES

A programmer is a device for turning caffeine into code.

— Many people

16.0 Dipping in

In the previous chapter we learnt how to group a sequence of items together under a single name as a list. These are all sequences in Python:

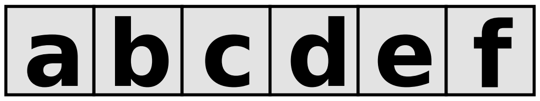
```
>>> escape_tunnels = ['tom', 'dick', 'harry']
>>> numbers = [0, 1, 2, 3, 4, 5]
>>> matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> jumble = [10, 'fred', 17.25, True, ['username', 123456789]]
```

However, there is another type of sequence that we are already familiar with - strings. As we know from *chapter 7* on printing, strings are sequences of characters, whether letters, digits or symbols, and can be treated as sequences in terms of slicing as we shall see later in this chapter. The main difference is that you cannot modify the contents of a string, but you can form new strings from old ones.

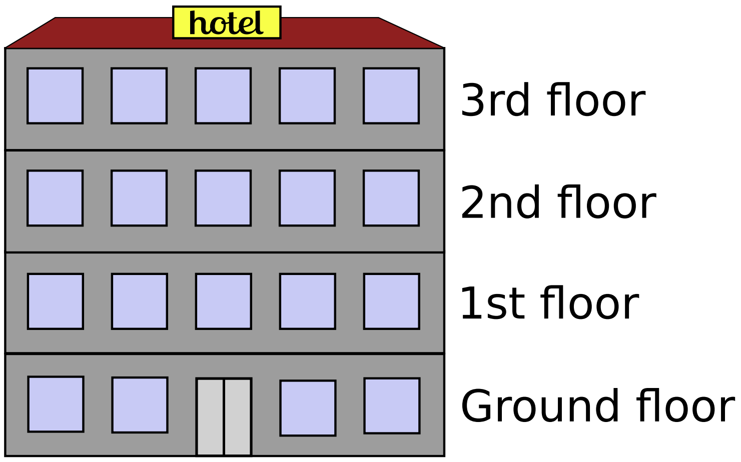
We have been able to refer to the group as a whole, but what if we need to select individual items out of the group - how do we dip into the sequence and reference a single item? Firstly, type this in so we start off with a simple string as our sequence to use:

```
>>> letters = 'abcdef'
```

We can refer to the whole sequence of letters, such as `print(letters)`, and even randomly select an item as we did the previous chapter. But how would we get at just one of those letters? Firstly, just like we illustrated in *chapter 7* on printing, think of this string as a sequence of boxes, each of which contains a single letter:

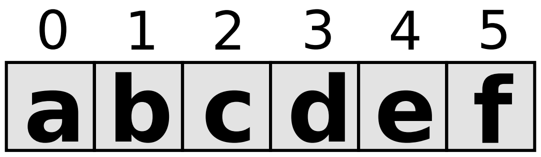


To "dip in" and fetch a single letter, we need to *index* the item as an offset from the start of the sequence. To picture this, think of a hotel, like so:



In the UK at least, we do not number the floors from the 1st floor and count up - the floors are numbered as how far up they are. Effectively, the floor number is an *offset* from the ground, i.e. 1st floor up, 2nd floor up, etc. The ground floor is really floor 0.

In Python, sequences are very similar - the number of each item is how far from the beginning it is - the offset from the first item:

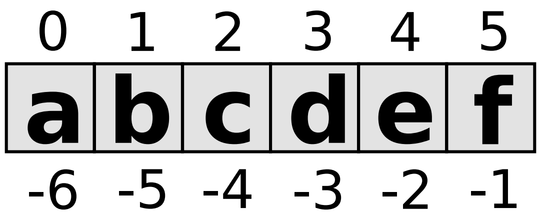


To use the offset of a particular item inside a sequence, you need to use square brackets in the form `[index]` or `[offset]`, just like how have leant how to box up a list of items. This is placed directly after the name of the sequence itself. We can now practice this in the interactive interpreter:

```
>>> print(letters[0])
>>> print(letters[1])
>>> print(letters[2])
>>> print(letters[3])
>>> print(letters[4])
>>> print(letters[5])
```

This should print off each letter in turn. Remember, the offset refers to how many places from the beginning, or the left, the item is to be found.

If we want to refer to an item not from the beginning, but instead from the end, we simply use negative numbers:



Practice again in the interactive interpreter:

```
>>> print(letters[-1])
>>> print(letters[-2])
>>> print(letters[-3])
>>> print(letters[-4])
>>> print(letters[-5])
>>> print(letters[-6])
```

This should print off each letter in turn, this time from the end or the right of the sequence. Notice when we used positive numbers, we start from 0 and ended up at an item offset by 5 (one less than the length of the sequence itself). When we use negative numbers, we start from -1 (as 0 is the first item), and end up at -6. This may not sound like a big improvement, but when your sequence is very long it is useful to state the offset from the right rather than from the left.

All of this is relevant for any type of list, whether they contain characters, strings, numbers, sub-lists or anything else. Let us use one from the previous chapter:

```
>>> names = ['fred', 'bob', 'harry', 'tom']
>>> print(names[0])
>>> print(names[3])
>>> print(names[-1])
>>> print(names[-4])
```

Be careful that you open and close the brackets correctly, if you are having problems! Moreover, if you use an offset that is past the end of the sequence, Python will complain - `print(names[4])` in this example.

16.1 Outside In

What if the sequence contains more than one level, like a matrix we mentioned in the previous chapter? For example, type the following to define a matrix of numbers:

```
>>> matrix = [[0,1,2], [3,4,5], [6,7,8]]
```

How do we get at the individual items inside on of the inner lists, such as the number 3? We use the same notation, and go from the outside sequence inwards. In this example, to get at the number 3, we first index the second item of the overall sequence, which gives us `[3,4,5]`. With this item, we can then index the actual number we wish, which being the first item has an index of 0. Although it is conceptually two steps, we can do it in one line by first indexing the correct item in the overall list, and then indexing the individual item in this inner list:

```
>>> matrix[1][0]
3
```

So the indexing goes from outside in, left to right, with the name of the overall sequence (or matrix) on the left-hand side.

The same occurs if the list is even deeper, such as a list of lists, each containing a string:

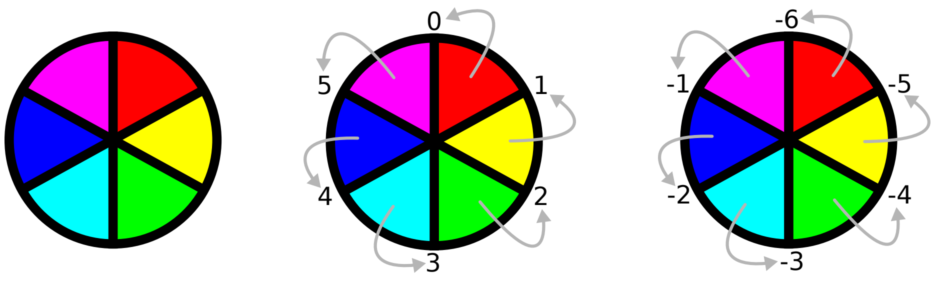
```
>>> names = [['tom', 'jones'], ['john', 'smith'], ['zippy', 'james']]
```

If we wish to pull out the `'y'` in `'zippy'` then - from the outside in - we index `2` to get at `['zippy', 'james']` then index `0` to get at `'zippy'` and finally `4` to index the `'y'`:

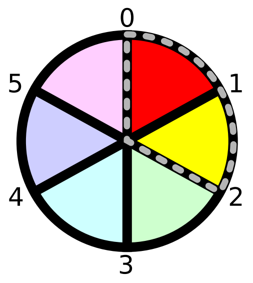
```
>>> names[2][0][4]
>>> 'y'
```

16.2 Unidentified Food Object

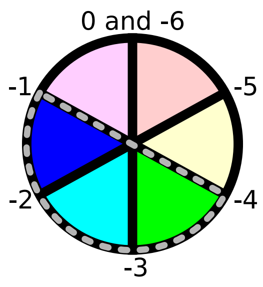
The aliens have landed on Earth, and they have brought pizza! Thankfully, their numbering system is the same as Python's, so here is a summary of how they refer to each slice:



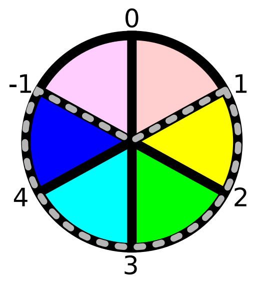
However, aliens are not satisfied with one slice, they are greedy. They are also lazy, and cannot be bothered to say every single number. So they say a range. For example, if an alien wants the red and yellow slice, he can say he wants all the slices between cuts 0 and 2. The serving alien takes piece 0, and adds one, taking piece 1. If he adds 1 again, he gets 2, so he has got all the pieces, and gives pieces 0 and 1 to the alien:



Aliens also do negative slices. An alien wants -4 to -1, which is the same as 2 to 5 so adding 1 gives the slices 2, 3 and 4.:

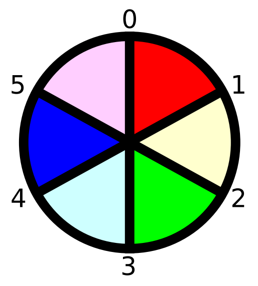


The opposite does not work, as you cannot add ones to 5 to get 2. 1 to -1 is the same as 1 to 5, so the slices are 1, 2, 3, and 4:

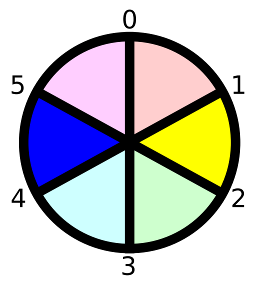


16.3 I'll pass

What if an alien only likes red, green and blue? Well, he can ask for every second piece from 0 to 5. The serving alien takes 0, adds two, so takes 2, and adds 2 again and takes 4. Adding two again will mean that he takes slice 6, but 6 is greater than 5, so he stops:



Every third slice from 1 to -1? That's the same as every third slice from 1 to 5, which is 1 and 4:



16.4 Python likes pizza

OK, now we know how to ask aliens for pizza, but how does Python do this? Let us represent the pizza as a list of the colors, one for each slice:

```
>>> pizza = ['red', 'yellow', 'green', 'cyan', 'blue', 'magenta']
```

Our `pizza` has all the colors of the alien pizza, in clockwise direction. As with indexing our letters string at the beginning of the chapter, we can pull out whole words from the list in a similar way (remember, each item is a word, not an individual letter):

```
>>> pizza[1]
'yellow'
>>> pizza[4]
'blue'
>>> pizza[-1]
'magenta'
>>> pizza[-3]
'cyan'
```

Now we can accommodate our hungry aliens. If we want all the slices from 0 to 2 we first we type the first index, the start, like before: `pizza[0`. Then we type a colon, `:`, followed by our second index, `2` which is the stop, followed by the closing bracket, `]`:

```
>>> pizza[0:2]
['red', 'yellow']
```

See how Python has given us a list of our slices! The other aliens would be happy:

```
>>> pizza[-4:-1]
['green', 'cyan', 'blue']
>>> pizza[1:-1]
['yellow', 'green', 'cyan', 'blue']
```

But what if our red-green-blue loving alien turned up? We first type the start and stop index: `pizza[0:5`. Then we type another colon, `:`, followed by the step we wish to take each time. To ask for every second slice, the step will be `2`, followed by the closing bracket, `]`:

```
>>> pizza[0:5:2]
['red', 'green', 'blue']
```

To obtain a list from the color sequence with every third slice, use a step of 3 (this time starting from index 1, all the way to the end indicated by a stop value of -1):

```
>>> pizza[1:-1:3]
['yellow', 'blue']
```

16.5 Slicing and dicing

When we use a single number to reference a single item it is called *indexing*; when we use more than one number to reference a range of items it is called *slicing*. The general form for slicing is `sequence[start:stop:step]`.

Indexing and slicing can happen on sequences containing data of any type. Define this list of the numbers from 0 to 20:

```
>>> nums = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
            10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

As before, we use an index of 0 to grab the first number in the list:

```
>>> nums[0]
0
```

And an index of -1 to obtain the last number:

```
>>> nums[-1]
19
```

We can grab the first 3 numbers by using a stop value in addition to the start:

```
>>> nums[0:3]
[0, 1, 2]
```

If we want to slice from the start you can miss the zero out:

```
>>> nums[:3]
[0, 1, 2]
```

Similarly, we can miss off the stop index if we want to slice to the end. For example, to get the last 5 numbers type the following:

```
>>> nums[-5:]
[15, 16, 17, 18, 19]
```

To get all the even numbers, we can use the step value all by itself:

```
>>> nums[::2]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

All the multiples of 3:

```
>>> nums[::3]
[0, 3, 6, 9, 12, 15, 18]
```

All the multiples of 3, offset by 1:

```
>>> nums[1::3]
[1, 4, 7, 10, 13, 16, 19]
```

16.6 Reverse gear

If you wish to slice a sequence in reverse (backwards), then you simply need to use a negative step. However, in this case, you must ensure the end index lower than the start index, otherwise it will return an empty sequence. Type in these examples:

```
>>> letters = 'abcde'
>>> letters[4:0:-1]
'edcb'
>>> letters[4::-1]
'edcba'
>>> letters[::-1]
'edcba'
```

The first slice goes from the 4th element (the letter 'e') to the beginning (up to, but not including, the letter 'a'), with a step of -1 every time.

If we wish to include the beginning as well, we can miss out the number for the end position - it will then stop when the sequence stops. This is the approach we take with the second example. Since we wish to go from the end all the way back to the beginning, we don't really need the start position either - let Python fill in those numbers for us. To copy the whole sequence, you would simply type `letters[:]` as it encompasses both the beginning and the end, inclusive, so adding a step of `-1` will slice from the end all the way back to the beginning, including both ends as it does so.

Tip: If you simply want to reverse a sequence of items, then use the built-in function `reversed`. For example, `''.join(reversed('abcde'))`, will print out `edcba` - the call to the `join` function is to join the list back together again, each separated by an empty string!

16.7 Cut the string

As strings are sequences as well as lists, this means we can slice them too. As before, if we want the first letter of someone's name, we can index it as follows:

```
>>> name = "Isaac Newton"
>>> name[0]
'I'
```

First three letters:

```
>>> name[:3]
'Isa'
```

First name:

```
>>> name[:5]
'Isaac'
>>> name[:-7]
'Isaac'
```

Surname:

```
>>> name[6:]
'Newton'
>>> name[-6:]
'Newton'
```

Initials:

```
>>> name[::6]
'IN'
```

Note: The above three examples are better done by:

```
>>> name.split()
['Isaac', 'Newton']
>>> name.split()[0]
'Isaac'
>>> name.split()[1]
'Newton'
>>> name.split()[0][0]
'I'
>>> name.split()[1][0]
'N'
>>> name.split()[0][0] + name.split()[1][0]
'IN'
```

This will work regardless of the length of the first name and surname.

Given the alphabet:

```
>>> alphabet = "abcdefghijklmnopqrstuvwxyz"
>>> len(alphabet)
26
```

We can find various things:

```
>>> alphabet[:3]
'abc'
>>> alphabet[::2]
'acegikmoqsuwy'
>>> alphabet[1::2]
'bdfhjlnprtvxz'
>>> alphabet[-3:]
'xyz'
>>> alphabet[5:8]
```

Cont...

```
'fgh'
```

16.8 Exercises

1. Write a program called `sentence.py` that inputs a sentence, and then prints out every other letter (i.e. prints even letters, but misses out odd ones), and also in reverse. Use both a `while` loop and slicing to achieve this, so that each print occurs twice.
2. Write a program called `daysofweek.py` which defines a list containing the days of the week (assume that Sunday is the first day). Ask the user for a number between 1 and 7, and print out the appropriate day of the week. For example, if the user types in `1`, then print out `Sunday`. If the user types in `7`, then print out `Saturday`. Note, you will have to take 1 off what the user has typed in before you use it as an index into your days of the week list.
3. Write a program called `planets.py` which defines a list with the 8 major planets of our solar system: Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus and Neptune (each one will be a string). Ask the user whether he wants either the rocky or gaseous planets. For the former, print out the first four planets; for the later, print out the last four planets - use slicing to do this.
4. Write a program called `colors.py` which defines the colors of the rainbow as red, orange, yellow, green, blue, indigo and violet. Your program should print out the primary colors of red, green and blue as a slice of your color list.
5. Write a program called `seasons.py`, which defines a list containing three sub-lists, for example:

   ```
   seasons = [['December', 'January', 'February'],
              ['March', 'April', 'May'],
              ['June', 'July', 'August'],
              ['September', 'October', 'November']]
   ```

   Ask the user which season, for example, "winter", "spring", "summer" or "autumn". If the user has entered "spring", then print out the first item in the seasons list, if "summer", then print out the second item, and so on. Bonus: use `', '.join(seasons[index])` to print out the month names nicely, with a comma between each and missing out the brackets.

16.9 Things to remember

1. Lists and strings are *sequences*, and so can be indexed and sliced.

2. The first item in a sequence has the *index* `0`, the second `1`, the third `2`, and so on. Think of the index as the offset from the beginning.
3. Negative indices can be used, counting from the end of the sequence. The last item is `-1`, the second from last -2, and so on.
4. If a sequence has more than one level (i.e. is multi-dimensional like a matrix), then you index from the outside in, with each index using the `[]` notation.
5. *Slicing* is done by `sequence[start:stop:step]`.

CHAPTER

SEVENTEEN

WALKING ALONG DATA

The Road goes ever on and on...

— Bilbo Baggins

17.0 Round and round

Python actually has two ways of repeating a block of code, something we call looping. The first method which we have already covered is by using the `while` keyword. The `while` statement includes a test (a *boolean expression*) that can change over time, thus affecting how many times the following block of code is run. Effectively the block of code is repeated *while* the expression remains `True`. It stops repeating when the expression becomes `False`. You can break out of the loop early by using the `break` statement.

For example, if we want to print out the numbers up to 10, then on the interactive interpreter we could do the following:

```
>>> num = 0
>>> while num < 10:
        print(num)
        num = num + 1

0
1
2
3
4
5
6
7
8
9
```

The test is whether the variable `num` is less than 10. We initially set it to 0, and every time we repeat the block of code, we add 1 onto it, and give it the same name. Eventually, it reaches 10, and the loop stops, as the variable `num` now equals 10 it is not less than 10, so the test evaluates to `False`. Hopefully this is all very straightforward by now.

However, Python has an easier way of repeating a block of code a set number of times (10 in this example). It is the `for` loop, and all it does is to step through a sequence such as a list or a string. We have been working with such sequences in the previous two chapters.

So let's introduce this step by step. First define a list and give it a name using the assignment operator:

```
>>> numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The name of the list is `numbers`, and we have given it the same values as we printed out using the `while` loop above.

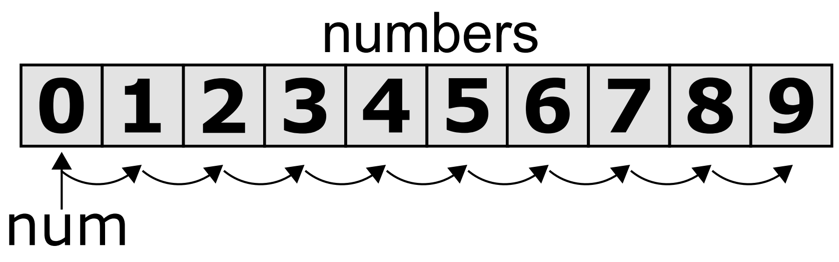
Now we use our new keyword `for`. We combine it with the `in` operator we first saw in *chapter 15* on working out whether a value is a member of a list, but this time it is used to step along each item *in* the list. To see it in action, type this in and make sure you get the same result as when we used the `while` loop earlier:

```
>>> for num in numbers:
        print(num)
```

And that is all we need to print out the numbers from the list we created. We could combine those three lines into two like this, thus avoiding the need for defining the list variable:

```
>>> for num in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]:
        print(num)
```

To explain what is going on - the `for` keyword is telling Python that *for each* item in the list (stored in the variable `num`), then execute this block of code. It is simply a way of stepping or *iterating* over a list, one item at a time. The variable `num` is defined for you - you do not have to set it up in advance, and it changes value as you walk along the list. It is referred to as the *loop variable*, and acts like a baton in a relay race - it is passed from one list item to the next in turn, like so:



To put it another way, you are asking for each item in the list, which Python passes to your program in the variable name provided, and repeats the code until the list runs out.

17.1 Ranges

However, it is a bit of a chore to have to type out the contents of our number lists all the time. Typing in ten numbers is one thing. What if we wanted to have a list with a 1,000 numbers? Or a million? Not only would this be very time consuming (and also make our programs very long), it would also be rather error prone. Think about typing in a few thousand numbers, and making mistake somewhere in the middle!

It is useful then that Python provides a function to give us a list of numbers in just the way we want. It is called `range`, and as the name suggests, it simply provides a range of numbers as a list of integers. Let's do the above program again, this time using the `range` function:

```
>>> for num in range(10):
        print(num)
```

Even shorter than before, now that we are using the `range` function to do the work for us. Usually, it just takes one argument - the number of integers that you want. If you pass in `5`, you get back `[0, 1, 2, 3, 4]`, that is, 5 numbers starting from 0. If you pass in `1000`, you get a list of a 1000 numbers, from 0 to 999.

Note: In programming generally, we like to start from 0 and not from 1, just like when indexing lists, or using `random.randrange`. We count up to an end point, but since we usually count from 0, we do not include the end point in our range.

If you don't want to start from 0, then you need to pass in two numbers - a start point and an end point. For example, if you want the numbers between 100 and 200, you simply ask for `range(100, 200)`. For numbers between -100 and 50, you ask for `range(-100, 50)`.

The third thing you can do is to ask for a step in the numbers, so instead of each number going up by 1, you can go up by a different amount. This is achieved by using a third argument, the `step`. For example, type this in the interactive interpreter to print out all the even numbers from 100 to 200:

```
>>> for num in range(100, 200, 2):
        print(num)
```

By this stage, you should realise that the range function is very similar to how we slice up sequences - specify an end point, and also optionally a start point along with a step to perform for each number. Therefore, there are three ways of calling the range function, depending on what kind of number range you want to produce:

```
range(end)
range(start, end)
range(start, end, step)
```

By using the *step*, you can also obtain a list that counts down rather than counting up. All the number ranges we have done so far have counted up. To count down, you need to use a negative step. However, you must also remember to make the end point lower than the start point! Again, this is similar to slicing a sequence backwards. Try this to count down from 10 to 1 (down to, but not including 0), inclusive:

```
>>> for num in range(10, 0, -1):
        print(num)
```

The variable used to step through the list (the loop variable) can be called anything you like.

It is a little different to how we have defined variables up until now which is by using the assignment operator:

```
>>> num = 10
>>> number = 20
>>> my_int = 123
```

As described above, the `for` loop defines its loop variable as part of the `for` statement, but just like ordinary variables, we can call it what we want to:

```
>>> for item in range(10):
        print(item)

>>> for counter in range(100):
        print(counter)
```

And you don't need to use it all, of course. It is used to just step through the list - what you do with it is up to you:

```
>>> for num in range(10):
        print('Going round and round 10 times!')
        print('Weeeeee!')

>>> for line in range(1000):
        print('I will not draw on the classroom wall again.')
```

17.2 Not just numbers

The list that the for loop uses to step over need not be a list of numbers. It can be a list containing anything you like. Try this:

```
>>> names = ['Bilbo', 'Gandalf', 'Thorin', 'Golum']
>>> for name in names:
        print('Enjoy your adventure', name)

>>> sentence = 'Mary had a litle lamb'
>>> for word in sentence.split():
        print(word)
```

The variable to step along need not be a list - it can be any sequence at all, including strings:

```
>>> word = 'rotavator'
>>> for letter in word:
        print(letter)
```

and items in reverse, using the `reversed` function we mentioned in the previous chapter on slicing:

```
>>> sentence = 'The attack starts at dawn'
>>> for letter in reversed(sentence):
        print(letter, end='')
```

We introduced the `end` argument in chapter 7 on printing - it simply instructs the call to not end the print with a new line, but with nothing (i.e. an empty string) instead. It allows multiple calls to print to add onto the current line, instead of starting a new one.

Finally, you can combine two sequences together, and then step over the result at the same time. To do this, use the built-in `zip` function which will turn the two individual sequences into a single sequence, made up of items from the originals. To step over the combined sequence, you need to use two `for` loop variables which will be set to the individual items from the original sequences. Try this out with these strings below, and notice how it only goes as far as the shortest of the two quotes, as the combined sequence effectively runs out of letters:

```
>>> first_quote = 'Rosebud.'
>>> second_quote = 'My precious.'
>>> for ch1, ch2 in zip(first_quote, second_quote):
        print(ch1, ch2)

R M
o y
s
e p
b r
u e
d c
. i
```

Again, as in *chapter 15* on grouping, whatever you can place in a list variable, you can use the `for` loop to step along it and execute the block of code that follows.

17.3 Drawing

Let's put this knowledge to use to draw a shape using turtle. Open a new file window and type in the following:

```
import turtle

# Set color and start shape
turtle.fillcolor('red')
turtle.begin_fill()

# Draw octagon
for side in range(8):                                          Cont...
```

```
    turtle.forward(50)
    turtle.left(45)

turtle.end_fill()
```

Save it as `redoctagon.py`, and run it. Not surprisingly, it should draw a red octagon.

A bit of explanation: we import the `turtle` module so that we can use it in our program; we then set the fill color as red and start the fill operation; we then loop round 8 times using the `for` keyword by going forward 50 pixels and turning left 45 degrees each time; we end by ending our fill operation so that the shape is filled in.

This is now much easier than before, whether using a sequence of statements, or even when we were using `while` loops.

17.4 Vertigo

Open up another new file window, and type in the following:

```
import turtle
import random

# Define the colors we will use below
colors = ['red', 'green', 'blue', 'magenta', 'cyan', 'yellow']

# Set the pen size, color and drawing speed
turtle.pensize(2)
turtle.speed('fastest')

# Start with a length of 5, and increase as we draw
length = 5

# Draw 400 lines, changing the color and length as we progress
for i in range(400):
    new_color = random.choice(colors)
    turtle.pencolor(new_color)
    turtle.forward(length)
    turtle.right(91)
    length = length + 2
```

Run and save it as `spirals.py`, and see what happens. If there any problems, then check your code carefully!

Some explanation: we import the modules we need, `turtle` for drawing, `random` to introduce a bit of variation. We then define the colors (note, not the English spelling - Python requires the American spelling) we are going to use. We then change the pen size and the drawing speed (so it doesn't take so long). We start with a line length of 5, which is increased for each line so the shape moves outwards. We then use a `for` loop to step along the range of numbers, from 0 to 399 (400 times in total). Inside the block of code that we are repeating (the loop), we change the pen color, move forward, change the angle (a little more than 90 degrees) and increase the length. We then repeat. The lines are drawn longer and longer, at an increasingly skewed angle.

Try changing the numbers to see what happens to the final result.

17.5 Loops inside loops

As we noted in *chapter 9* on selection, you can have blocks of code inside other blocks of code. These blocks of code could be repeated, with other blocks also repeated - in other words, we can have loops inside loops - one section of code repeated inside another.

Let us practice this concept by using the interactive interpreter:

```
>>> for outer_number in range(1, 10):
        print('outer loop', outer_number)
        for inner_number in range(1, 10):
            print('inner', inner_number)
outer loop 1
inner 1
inner 2
inner 3
inner 4
inner 5
inner 6
inner 7
inner 8
inner 9
outer loop 2
inner 1
inner 2
inner 3
inner 4
inner 5
```

and so on. You will notice the outer loop starts, and before it repeats the inner loop takes over. This then repeats until it runs out of items to step over (numbers in this example), and then the outer loop resumes.

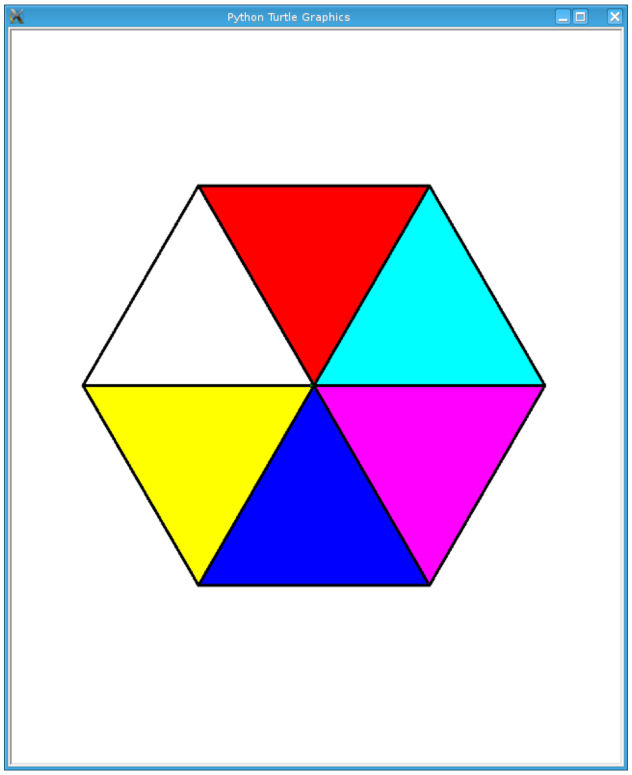
Now to do something longer and more colorful, start a new program and type in the following:

```
import turtle
import math

# Define our colors to use lower down                           Cont...
```

```
colors = ['red', 'cyan', 'green', 'magenta', 'blue', 'yellow', 'white']

# Set pen size and speed
turtle.pensize(5)
turtle.speed(0)

# Set the size of each triangle
length = 400

# Move a bit up and to the left so the shape is centred
turtle.up()
triangle_height = length / 2 * math.sqrt(3)
turtle.goto(-length/2, triangle_height)
turtle.down()

# Use colors from the start, and move along each time
color_index = 0

while True:
    # Draw six triangles, centred on a point
    for triangle in range(6):
        # Select a color from the color list
        color = colors[color_index % len(colors)]
        turtle.fillcolor(color)
        turtle.begin_fill()

        # Draw each of the triangle's 3 sides
        for side in range(3):
            turtle.forward(length)
            turtle.right(120)
        turtle.end_fill()
        turtle.forward(length)
        turtle.right(60)

        # Increment our index, so the colors are rotated
        color_index = color_index + 1
```

Run it, naming it `triangles.py`, and see what happens. Much of what we have typed in is similar to the `spirals.py` program, but this time we have loops inside other loops. The first loop simply repeats the main part of the program forever, an infinite loop. We know it is an infinite loop as the condition is `True`, which never changes to `False`. Inside this loop, we draw 6 triangles, centred on a point so they form a hexagon shape. We do this centring by turning 60 degrees towards the end of this loop. Inside this loop drawing all the triangles is another `for` loop which draws the 3 sides of each triangle, turning 120 degrees each time.

Your turtle window should show something like this being drawn:



One extra note - we use an index to rotate our colors, so it steps along the colors in sequence. We do this by incrementing (adding onto) the index after drawing every triangle. When selecting a color, we use the modulus operator `%` (the remainder) so that it repeatedly go from 0 to 6, inclusive. In other words, once it reaches 6, it returns to 0 and climbs back up again. Notice how we have one more color (7 in total) than we do triangles (6), so that every time we repeat the `while` loop, the colors shift by one from one triangle to the next. This allows us to see the outer loop working in action, moving the colors along as it runs.

17.6 Exercises

1. In the interactive interpreter, write a `for` loop that counts from 1000 to 2000 in steps of 50.
2. In the interactive interpreter, write a `for` loop that counts from 100 to 0 in steps of -5.
3. Write a program called `sides.py` which uses the `turtle` module to draw a polygon having the number of sides the user has input. Use a `for` loop to draw the sides of the polygon. This is similar to the exercise in *chapter 12*, but this time the looping is different.
4. Write a program called `brekkie.py` which creates an empty list called breakfast (using the notation `breakfast = []` to create an empty list). Ask the user what they had for breakfast, one item at a time, and call `append` for each item to append it onto the breakfast list. Use a `while` loop to accomplish this, allowing the user to type 'stop' to

break out of the loop. Then use a `for` loop to print out each item in the breakfast list, printing out how yummy each item is.

5. Write a program called `bullseye.py` which draws a series of red and white circles, ever smaller, to form a bullseye shape. The program should draw 11 in total, starting with a large red circle, and finishing with a small red circle, with alternate white and red circles in the middle. Try and centre your shape in the middle of the turtle window.

17.7 Things to remember

1. Use the `for` loop to repeat a block of code a set number of times. Use the `while` loop to repeat a block of code an unknown number of times (e.g. depending on whatever the user types in). The `for` keyword can be read as *for each* item in the sequence, then do this code block.
2. Use the `range` function to provide a sequence of numbers to step through. You can use it with just one argument, the end point, or with two, the start and end point, or three arguments, start, end and step.
3. You can use the `break` keyword inside a `for` loop as well as the `while` loop we learnt in *chapter 13* on escaping the cycle. This breaks out of the loop before the loop has finished stepping along the sequence of data.

CHAPTER

EIGHTEEN

NAMING CODE

There are 2 hard problems in computer science: cache invalidation, naming things, and off-by-1 errors.

— Leon Bambrick

18.0 Data and code

Variables are a way of naming data. A piece of data, like the number `123`, can be given the name `number` like this in Python (try it in the interactive interpreter):

```
>>> number = 123
```

In this case, the data is an integer, a whole number. We can name other types of data as well:

```
>>> cups_of_flour = 2.5
>>> name_of_recipe = 'wholemeal bread'
>>> pages_from_book = [19, 22, 23]
>>> is_recipe_tasty = True
```

This is very good for people reading the code, as it describes the meaning behind the data much clearer when it is given a name like this. We can also use the same name, but modify its value over the course of the execution of the program, thus allowing variables to really *vary*:

```
>>> people_going_to_mordor = ['Frodo', 'Sam']
>>> people_going_to_mordor.append('Merry')
>>> people_going_to_mordor.append('Pippin')
```

or more simply such as:

```
>>> number = 100
>>> number = number + 1
```

Programs are made up of data, which we have been discussing here, and code. We have the ability to name our data, but it would be helpful if we also had the ability to name our code. This would prevent repeating our code, using the name instead of the code itself, and also allow our programs to become more organised.

18.1 Functions

We have already used named sections of code - they are called *functions*. Python programs are generally organised into modules (the programs themselves) and functions (the named sections of code contained within each module). A function is given a name, and we invoke it (call it) by placing parentheses after its name. Try the following, again in the interactive interpreter:

```
>>> print()

>>> print('Hello, World!')
Hello, World!
>>> abs(-15)
15
>>> new_num = round(10.75)
>>> print(len('The quick brown fox jumps over the lazy dog'))
43
```

As we can see above, functions can take in values (which we have learned are called *arguments*), or not. They can return values, or not. Even if they do return a value, it is up to us whether we use it or not. It just depends on what they do, and what we want to use them for.

In this chapter, we will begin learning how to define our own functions, and not just use the functions already present in the language (built-in ones), or from other programs (imported modules).

But before we do, why should we bother? Can't we just use loops instead to avoid repeating code?

Loops are very useful, but only when the code to be repeated is in the same place. Functions are useful when they are being used from a number of places, especially if they are not in related parts of your program. They are more flexible, and offer a way of being called from anywhere - even outside your program, just like we do to other modules (e.g. `turtle`). Moreover, giving pieces of your code a name makes it more understandable to those reading it, or even to you when you come back to your program in the future.

Here is an example of a bit of code that needs sections to be separated into functions and given a name. You don't need to type this in, it is just provided for information:

```
print('Hello there')
print()
print('Hello there')
num = 10
print('Hello there')
num = num + 1
print('Hello there')
```

Lots of repetitive code that could be parcelled up, given a name and put in one place, but called from where it is needed. Moreover, when code only needs to be written once, we tend to make less mistakes than when we have to write it many times.

18.2 Rolling our own

When naming data we use the assignment operator, when naming code we use the `def` keyword. This means we are *defining a function* - not calling it, but creating it. Just because you define a function, doesn't mean it gets called - it is simply there ready to be called upon when needed.

Open up a new file window, and type in the following:

```
def say_hello():
    print('Hello there')
```

Save your program as `functions.py`, and run it. It should do nothing - as described just above, it is available to your program, but it is not being run yet. To do that, insert the following line lower down your program:

```
say_hello()
```

Now save and run your program again - it should now print out something. In fact, it should say hello! Just like calling other functions, if you miss off the parentheses, then the function does not get called, it simply returns where it is located in memory - probably not what you intended!

So to define a function, we use the `def` keyword. To call or invoke a function, we use the name of the function followed by parentheses `()`.

18.3 Passing in data

However, a function that always does the same thing is very limited. It is more useful to have the ability to pass information into the function, so the function can use this information on the inside, as it were. For example, let's take the example we did above, but vary it slightly. So type this in beneath your definition of the `say_hello` function in your `functions.py` program (i.e. not at the bottom, so as to keep your functions together, and your calls together):

```
def say_hello_times(times):
    print('Hello there' * times)
```

Now insert this new line below your `say_hello()` call at the bottom of your program:

```
say_hello_times(5)
```

Run it and ensure it prints out the same message, but this time 5 times. That will be 6 times in total, with the initial call to the `say_hello` function as well. Try a couple more combinations:

```
say_hello_times(10)
say_hello_times(50)
```

It should print out the message the number of times you are requesting. If not, then something is very wrong!

Now, of course, if you can pass in one item, you should be able to pass in more than one item. So our next function (again, insert it between where the functions are defined and where they are being called), will take a message and a number, so the caller can specify what he wants printing:

```
def say_message_times(msg, times):
    print(msg * times)
```

and insert these lines below the last call to `say_hello_times`:

```
say_message_times('Yo! ', 25)
say_message_times('I will stop talking in class ', 100)
```

Note: An argument is the value or variable being passed *into* a function. A *parameter* is the variable as received inside a function. Or in other words, it has parameters, but takes in arguments. Perhaps it is easier to remember it like this: for the sake of argument, if we pass in `'Yo!'` and `25`, the function uses them as parameters - arguments on the outside, parameters on the inside. If you cannot remember the difference, don't worry - just call them all parameters!

These parameters can be anything you like, it just depends on what the function does, and what type of information it needs. Let's add onto our list of functions one that takes in two numbers, and prints out the sum:

```
def add_two_nums(num1, num2):
    print(num1 + num2)
```

and again, lower down, we can call this function:

```
add_two_nums(10, 20)
add_two_nums(-50, 25)
add_two_nums(100, 200)
```

If you call this function with three arguments (e.g. `add_two_nums(10, 20 30)`), then Python will complain - three arguments into two parameters does not go!

These functions are very short, so may not appear very useful yet, but imagine writing a function that is 10 or 20 lines long, and is used in a number of places in your program. Then your program will be much shorter and be more readable as you have given part of your code a name that describes what it does.

In the next chapter, we will discuss not only passing data into a function, but also getting data back out again.

18.4 Keeping your data local

Usually, data used inside a function should be passed in, so if a function adds two numbers together, both numbers should be passed in as parameters. It should never have to rely on variables outside of its own definition - if it needs the data, pass it in. These variables are called *local* variables, as they are defined locally, or within, the function itself.

However, sometimes this is impractical, so functions always have the ability to use variables defined in the module itself. Variables that have been defined in the module (i.e. the program), and not part of a function, are called *global* variables, as they are defined for use throughout the program and not just a part of it. Type this into your `functions.py` program:

```
def add_by_5():
    print(num + 5)
```

and at the end of your program, add this:

```
num = 10
add_by_5()
```

This should print out 15, with 5 being added onto 10. However, if you want to change the variable `num`, or any global variable, then you will hit trouble. Change your function to read like this:

```
def add_by_5():
    num = num + 5
    print(num)
```

This should print out an error, as Python assumes you are using a local variable called `num` before defining it (in using it on the right-hand side of the assignment statement). If you really want to change a global variable, then you must state this in advance by using the `global` keyword alongside the variable name itself. Change the function in your program as follows:

```
def add_by_5():
    global num
    num = num + 5
    print(num)
```

Run your program again, and it should now be happy, finding the global variable of `num` as you intended.

The general rule, though, is to pass in all the data the function needs, unless the data never changes such as a list of month names or the value of :math‘pi‘ from the `math` module, for example.

18.5 Exercises

1. Write another function called `calc` which accepts two numbers and also a string value which you can call `operator`. The operator parameter can be either "add", "subtract", "multiply" or "divide". Depending on this value, you should perform the appropriate calculation, and print out the result, For example, if the values 4, 5, "add" are passed in, then it should print out the result 9. If the values 100, 8, "divide" are passed in, then it should print out 12.5. You can place this function inside the same `functions.py` program.
2. Write a function called `timestable` which receives a number and prints out a times table with the specified number of rows and columns. For example, if the number 5 is passed in, then the 5 times table is printed. If the number 12 is passed in, then the 12 times table is printed. It is best to use two `for` loops - one for the rows, and inside this, another for the numbers themselves, both counting along the same range of numbers (multiplying them to produce the result to display). Again, put it in the same `functions.py` program.
3. Start a new program called `shapes.py`. It should use the `turtle` module and repeatedly ask the user what shape to draw - for example, box, circle, polygon, star. Depending on what the user types in, the program should draw that shape. The code for drawing each shape should reside in its own function, e.g. a function each for `box`, `circle`, `polygon` and `star`. Each function will have to ask the information it needs itself, e.g. a box will need its length and width, a circle will need its radius, etc.

18.6 Things to remember

1. To define a new function, use the `def` keyword, followed by the name of the function, and then parentheses.
2. Inside the parentheses, place any *parameters* you are expecting. Separate each one using a comma. This is the way of passing in data to affect how the function behaves - pass in different data, it should do different things.
3. The *function definition* is completed with a colon `:` symbol, followed by the code that is inside the function. This code, like any block of code, is indented to the right.
4. Defining a function does not mean it is used - it is simply available to be used, like a tool in a toolbox. To *call* or invoke a function, you must use its name, followed by parentheses, but without the `def` keyword.

CHAPTER

NINETEEN

MORE ON FUNCTIONS

Once more unto the breach, dear friends, once more!

— William Shakespeare

Henry V

In the previous chapter we learned how to define functions of our own, and how to pass in data that the functions can then use. To follow on from this, we will now learn how to define functions that not only allow data to be passed in, but also return data that can be used by the calling code. We are already used to this in the way we use functions built into Python itself. Try these in the interactive interpreter:

```
>>> round(1.75)
2
>>> abs(-10)
10
>>> max(10, 30, 20)
30
>>> min(50, 100, 25)
25
>>> sum(range(1000))
499500
```

Hopefully that is all very straightforward to you now. The last example gets a list of numbers from the `range` function, and passes it into the `sum` function, which sums all the numbers together, returning the total which is then shown in the interactive interpreter. It has effectively added up the first 1000 numbers, from 0 to 999.

How do we do this in our own functions?

19.0 Please talk to me

Open up your `functions.py` program, and add the following function between your functions and the code calling them:

```
def add_5(num):
    return num + 5
```

We have used the new keyword `return` - this takes an expression, and returns it to the caller of the function. This means it is used (or discarded if it is not needed), by whatever code that has called the function in the first place.

Now add these lines onto the bottom of your program, so that the function defined above is called:

```
print(add_5(10))

my_num = 20
print(add_5(my_num))

new_num = add_5(my_num)
print(new_num)
```

Now run the program, and see what it does. It should call our new function `add_5` a number of times. The first ones simply passed in the integer value 10. Inside the function, the parameter `num` will refer to this value of 10. The value is incremented by 5, and the result is *returned* or sent back to the code that called the function. In the first call of `add_5`, this happens to be a `print` function, which naturally prints out the result it has been given (the number returned back from the function call).

The second use of the `add_5` function is similar, but instead of passing in a value, it passes in a variable which is referring to an integer value. It then proceeds as before.

The third use of the `add_5` function is similar to the second use, but instead of printing the value returned back from the function call straight away, it first assigns the returned value to a new variable called `new_num`. This is then printed out on its own.

19.1 Forming a chain

With the use of the `return` keyword to send data back, you can effectively form a chain of functions just like we have done with the built-in ones at the beginning of the chapter. Type this into your `functions.py` program, under your other function definitions:

```
def sum_up(num1, num2):
    return num1 + num2
```

Then below, with the other calling code, add the following lines:

```
print(sum_up(10, 20))

total = sum_up(100, -50)
print(total)
```

This is very similar to what we have done already. Now let's chain our functions together:

```
print(sum_up(sum_up(1, 2), sum_up(3, 4)))
```

This could go on and on! You are effectively forming an expression in the shape of a tree - the inner calls to `sum_up` are called first, the one on the left, and then the one on the right. With these two values, 3 and 7, respectively, the outer `sum_up` is called, thus producing the final printed result of 10. This expression is equivalent to `(1 + 2) + (3 + 4)`.

19.2 Naming parameters

So far we pass *arguments* into functions, used inside the function as *parameters*, to feed data into the function. We generally do the following, which you should now type into your ever increasing `functions.py` program:

```
def box_volume(length, height, width):
    return length * height * width

print(box_volume(10, 20, 30))
```

You should place the calling of the function, the line containing the function name `print`, along with the other code towards the bottom of your program.

It is quite clear that the integer value `10` is passed into parameter `length`, `20` is passed into the parameter `height`, and `30` is passed into the parameter `width`. In Python, these are called *positional arguments* - the position of each argument determines which parameter it is passed into. The first argument is passed into the first parameter, the second argument is passed into the second parameter, and so on. If you get the order of your arguments wrong, then then the wrong data will be fed into the wrong parameters. Bad things will happen.

An alternative is to explicitly state what parameters you want to use for each argument (remember, arguments on the outside are being passed into parameters on the inside). Use the same function definition, but call it in this way. You should place this line beneath the last statement from above:

```
print(box_volume(length=10, height=20, width=30))
```

Run your program again, and make sure it now prints out the same volume twice. However, the line calling the function `box_volume` makes more sense with the parameter names being assigned to the argument values explicitly. Yes, it is more typing, but reads better.

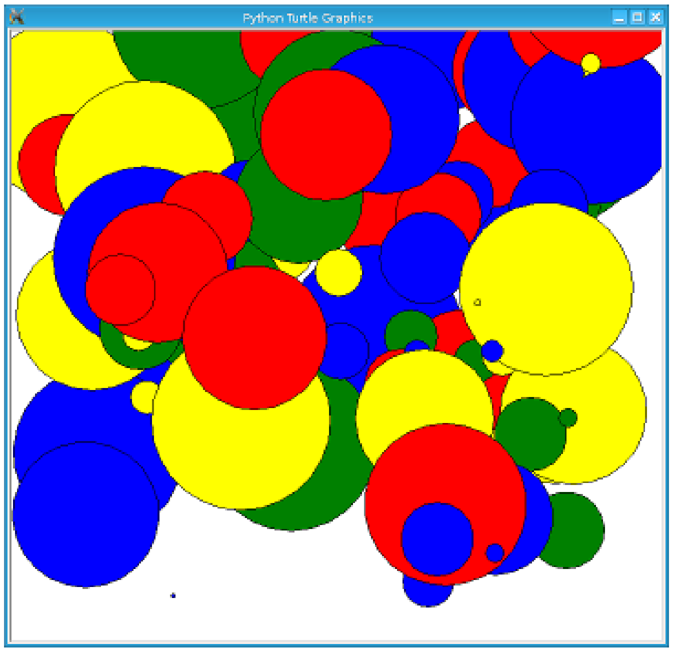
This is called *keyword arguments* - you are referring to each parameter by name or keyword, and supplying the data you want to be associated with each. This may not look very useful in this example, but when function definition and function invocation (i.e. calling the function) are in different modules, then it allows you to immediately see what value is being passed into what parameter. The function call contains more information, and allows you to see what is going on.

19.3 A little more practice

We will write a little turtle based program to demonstrate some of the concepts we have been learning here. Open up a new file, and type in the following:

```python
import turtle
import random


def draw_circle(radius, red, green, blue):
    turtle.pencolor(red, green, blue)
    turtle.fillcolor(red, green, blue)
    turtle.begin_fill()
    turtle.circle(radius)
    turtle.end_fill()

# Set the speed and save width and height
turtle.speed('fastest')
win_width, win_height = turtle.window_width(), turtle.window_height()

while True:
    # Define the radius of the circle, between 20 and 200 pixels each
    size = random.randrange(20, 200)

    # Move to a random position in the window
    # Remember to pick up the pen first
    x = random.randrange(-win_width // 2, win_width // 2)
    y = random.randrange(-win_height // 2, win_height // 2)
    turtle.up()
    turtle.goto(x, y)
    turtle.down()

    # Draw circle
    draw_circle(radius=size,
                red=random.random(), green=random.random(),
                blue=random.random())
```

Run your program, saving it as `circles.py`, and make sure it runs without errors. You should be getting lots of randomly colored and sized circles on the screen, similar to the following:



A little explanation:

- We import the modules we need, `turtle` for drawing, `random` for producing a bit of variation.
- We then define a function called `draw_circle` which take four arguments - the radius, followed by red, green and blue to define the color.
- Inside the `draw_circle` function, we set the pen and fill color, tell turtle we are starting the shape so it can be filled in later, and then draw a circle. We then end the shape, so the circle is filled in.
- In the main part of the program, we set the speed to hurry things up, and then save the window width and height so we can use them later.
- We then enter a loop which continues forever.
- Inside the loop, we first define the size of the circle by using the `randrange` function in the `random` module. We ask for a radius somewhere between 20 and 200.
- We then pick up the pen, and move it to a random place in the drawing window, and then put the pen down again.
- We then call our `draw_circle` function using the data we have at hand.

19.4 Exercises

1. Write a function called `add_list` in your `functions.py` program, which accepts a list comprising of a list of integers. The function will step through the list, and return the

sum. The sum should then be printed out.

2. Write a function called product in your `functions.py` program, which accepts two numbers. The function returns the product of these numbers (i.e. the numbers multiplied together). Then call this function, `product`, along with the function `sum_up` we wrote earlier, to form a tree-like expression. Print out the result. For example, use your functions to imitate this arithmetic expression: `(4 * 5) + (6 * 7)`.

3. Write a function called prime in your `functions.py` program, which accepts a single number and returns `True` (a boolean value) if it is a prime number or `False` if not. Remember, 0 and 1 are not prime, 2 is prime, and for the other numbers, a prime number is one that is only divisible by itself and 1.

19.5 Things to remember

1. Functions can both receive and return data. Data is received via the use of *parameters*. Data is returned via the use of the `return` keyword. You combine the `return` keyword with an optional expression to form the *return statement*.

2. Even functions without the `return` statement return a value - the value `None`. It is like a non or null value, similar to zero but not actually an integer number.

3. When a program comes across the `return` keyword, control returns immediately to the calling code. This is the case even if there is more code after the return statement - this code is effectively out of reach by the program. This is why it is called *unreachable* code.

4. There are two ways of passing in arguments with functions. Firstly, by *position* (*positional arguments*), so the order of arguments is matched up with the order of parameters. Secondly, by *keyword* (*keyword arguments*), so you can specify the name of the parameter, followed by the equals sign, and then the expression (e.g. a value or variable name) that parameter should set to.

CHAPTER

TWENTY

READING FILES

Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?

— Brian W. Kernighan

20.0 Opening the file

We have used Python to store values in memory by using variables. Type the following in the interactive interpreter:

```
>>> message = 'The attack starts at dawn'
>>> troops = 85
>>> print('Your message:', message, 'and your troops:', troops)
```

However, storing values in memory by using variables is only transitory - just like some maths calculator. If you store a number in memory using the **MS** (memory store) button, turn your calculator off and on again, and try and recall the number you stored by using **MR**, it will be gone. Variables in computer memory are similar - when your program stops (or the computer is switched off), the memory is gone. To demonstrate this, having typed in the code above, select the **Restart Shell** menu item from the **Shell** menu. Then redo the print command (you can use the up arrow and press the **Return** or **Enter** key twice), and see what happens - Python will complain it cannot find the variables stating you have not defined them. In other words, they have gone from memory!

So how do we retain information from one run of our program to the next? How do we, as a computer scientist would say, make our data *persistent* - i.e. recall the data when the program is run again? Think of a game with a highest scores table - we need a way to store these numbers (and names) so that they can be read and changed every time the game is run. To do this, we need to place our data in a *file* that is stored on disk - whether a hard drive or flash storage such as a USB stick. This data, when the electricity is turned off, retains its state - the data does not drain away with the current! To work with files, we need to learn how to read from them (in this chapter) and write to them (in the next).

The first step in reading a file is opening it. Think if a file like a folder or a book - before you can start reading its contents, you need to open its cover to reveal the pages within. Firstly, we need to create our book, so click on **File → New File**, and copy in the following text (remember to use **Ctrl-C** keys to copy and **Ctrl-V** to paste):

```
Three Rings for the Elven-kings under the sky,
Seven for the Dwarf-lords in their halls of stone,
Nine for Mortal Men doomed to die,
One for the Dark Lord on his dark throne,
In the Land of Mordor where the Shadows lie,
One ring to rule them all, one ring to find them,
One ring to bring them all and in the darkness bind them
In the Land of Mordor where the Shadows lie.

- The Lord of the Rings, Epigraph
```

Save it as `mission.txt`, making sure you save it in your home directory (`/home/pi` directory), not on your USB stick. That way we can experiment with it in the interactive interpreter. We now have a file to open and read.

In Python, we use the `open` function to open files, so type the following:

```
>>> f = open('mission.txt')
>>> f
<_io.TextIOWrapper name='mission.txt' mode='r' encoding='UTF-8'>
```

This shows that the file has been opened. The `mode` is `'r'`, which means the it is open for reading. The other mode, `'w'`, is for writing and it is covered in the next chapter. Note you have to enclose the name of the file - the filename - in quotation marks as it is a string. If this does not work, make sure the `mission.txt` file is in the correct location as indicated above and named correctly.

We have called the variable that refers to our open file `f`, but it could be called anything just like other variables, such as `my_file`, `saurons_dark_secret`, `input_file`, `my_todo_list`, or the like.

Note: If you get the filename wrong (or the file is located in a different directory), you will get an error like this:

```
>>> open('missing.txt')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'missing.txt'
```

20.1 Reading words of wisdom

Once we have an open file, in this case `f`, we can read its contents. The open file has a function, `read`, to give the contents of the file as a string:

```
>>> print(f.read())
Three Rings for the Elven-kings under the sky,
Seven for the Dwarf-lords in their halls of stone,
Nine for Mortal Men doomed to die,
One for the Dark Lord on his dark throne,
In the Land of Mordor where the Shadows lie,
One ring to rule them all, one ring to find them,
One ring to bring them all and in the darkness bind them
In the Land of Mordor where the Shadows lie.

- The Lord of the Rings, Epigraph
```

This allows us to call all the functions that belong to the string type. To convert the file into a list of words, type the following:

```
>>> f = open('mission.txt')
>>> f.read().split()
['Three', 'Rings', 'for', 'the', 'Elven-kings', 'under', 'the', 'sky,',
'Seven', 'for', 'the', 'Dwarf-lords', 'in', 'their', 'halls', 'of',
'stone,', 'Nine', 'for', 'Mortal', 'Men', 'doomed', 'to', 'die,',
'One', 'for', 'the', 'Dark', 'Lord', 'on', 'his', 'dark', 'throne,',
'In', 'the', 'Land', 'of', 'Mordor', 'where', 'the', 'Shadows', 'lie,',
'One', 'ring', 'to', 'rule', 'them', 'all,', 'one', 'ring', 'to',
'find', 'them,', 'One', 'ring', 'to', 'bring', 'them', 'all', 'and',
'in', 'the', 'darkness', 'bind', 'them', 'In', 'the', 'Land', 'of',
'Mordor', 'where', 'the', 'Shadows', 'lie.', '-', 'The', 'Lord', 'of',
'the', 'Rings,', 'Epigraph']
```

And to count the number of words in the file we can do:

```
>>> words = open('mission.txt').read().split()
>>> len(words)
81
```

However, if you play with files, you will some interesting behaviour, such as:

```
>>> f = open('mission.txt')
>>> f.read()
'Three Rings for the Elven-kings under the sky,\nSeven for the
Dwarf-lords in their halls of stone,\nNine for Mortal Men doomed
to die,\nOne for the Dark Lord on his dark throne,\nIn the Land
of Mordor where the Shadows lie,\nOne ring to rule them all, one
ring to find them,\nOne ring to bring them all and in the
darkness bind them\nIn the Land of Mordor where the Shadows lie.
\n\n - The Lord of the Rings, Epigraph \n'
>>> f.read()
''
```

These special escape sequences (such as `\n` for newline and `\t` for tab) were covered briefly in *chapter 7* on printing. This is the text file as it really is, not formatted nicely for reading.

If you read a file completely, the open file points to the end of the file. This is like having a book open at the end of the last page. If you want to re-read the file, you can re-open the file (similar to closing a book and reopening it at the beginning again), or use the function `seek` to move back to the start (similar to flicking through the pages back to the beginning, but much quicker):

```
>>> f.seek(0)
0
>>> f.read()
'Three Rings for the Elven-kings under the sky,\nSeven for the
Dwarf-lords in their halls of stone,\nNine for Mortal Men doomed
to die,\nOne for the Dark Lord on his dark throne,\nIn the Land
of Mordor where the Shadows lie,\nOne ring to rule them all, one
ring to find them,\nOne ring to bring them all and in the
darkness bind them\nIn the Land of Mordor where the Shadows lie.
\n\n - The Lord of the Rings, Epigraph \n'
```

20.2 Line by line

To get the entire file as a string, we use `read`. If we want it line by line, however, we can use a `for` loop, and *iterate* over the file:

```
>>> f = open('mission.txt')
>>> for line in f:
        print(line)

Three Rings for the Elven-kings under the sky,

Seven for the Dwarf-lords in their halls of stone,

Nine for Mortal Men doomed to die,

One for the Dark Lord on his dark throne,

In the Land of Mordor where the Shadows lie,

One ring to rule them all, one ring to find them,

One ring to bring them all and in the darkness bind them

In the Land of Mordor where the Shadows lie.


                                                      Cont...
```

```
- The Lord of the Rings, Epigraph
```

For most purposes, this is the best way to read a file and fits in well with what we have learnt elsewhere in the book. Notice how the print inserts an extra blank line in-between each line from the file - since the line from the file contains a newline character already, this is added onto the newline that the print function adds itself.

This also works for the `list` function:

```
>>> f = open('mission.txt')
>>> list(f)
['Three Rings for the Elven-kings under the sky,\n',
 'Seven for the Dwarf-lords in their halls of stone,\n',
 'Nine for Mortal Men doomed to die,\n',
 'One for the Dark Lord on his dark throne,\n',
 'In the Land of Mordor where the Shadows lie,\n',
 'One ring to rule them all, one ring to find them,\n',
 'One ring to bring them all and in the darkness bind them\n',
 'In the Land of Mordor where the Shadows lie.\n',
 '\n', ' - The Lord of the Rings, Epigraph \n']
```

To read directly as a list, we can use `readlines`:

```
>>> f = open('mission.txt')
>>> f.readlines()
['Three Rings for the Elven-kings under the sky,\n',
 'Seven for the Dwarf-lords in their halls of stone,\n',
 'Nine for Mortal Men doomed to die,\n',
 'One for the Dark Lord on his dark throne,\n',
 'In the Land of Mordor where the Shadows lie,\n',
 'One ring to rule them all, one ring to find them,\n',
 'One ring to bring them all and in the darkness bind them\n',
 'In the Land of Mordor where the Shadows lie.\n',
 '\n', ' - The Lord of the Rings, Epigraph \n']
```

Notice that in both cases, the newline characters (`'\n'`) are still in the string.

20.3 Put down the book

When we have finished with a file, we should always call `close` on the file variable:

```
>>> f.close()
```

This just like closing the covers of a book, or the flap of a real folder from a filing cabinet. It is polite way to finish working on a file - in the next chapter, this becomes more essential.

20.4 Exercises

For these exercises, you will need to copy the text file `mission.txt` from the home directory on the Raspberry Pi (`/home/pi/mission.txt`) to the same location as your programs - either on your USB stick (e.g. `/home/pi/USB_STICKS/USB Disk`) or in your named directory (e.g. `/home/pi/fredbloggs`).

1. Open and print out the contents of the `mission.txt` file, converting each line to uppercase (tip: use the `upper` function on the line string to achieve this).
2. Open and print out the length of each line in the `mission.txt` file.
3. Open and step through each line of the `mission.txt` file. Once done, print out the average word length - you will need to keep a running total of all the word lengths, plus how many words there were in order to print out the average.

20.5 Things to remember

1. Use the built-in `open` function to open files.
2. Use `read` function on the file variable to get the contents of the file.
3. Use a `for` loop to iterate over the file, getting each line in turn. This is the best way to step over the contents of a file, one line at a time.
4. Use the `readlines` function on the file variable to get a list of lines.
5. When you `read` a file, you need to move back to the start by re-opening the file, or using `seek`.

CHAPTER

TWENTY ONE

WRITING FILES

Complexity has nothing to do with intelligence, simplicity does.

— Larry Bossidy

21.0 Prepare your ink

The opposite to reading a file is writing to it. Whilst in the previous chapter on reading we used IDLE's editor to create a file in order to read from it, this chapter we will do the writing from within the program itself.

To write to a file, we first have to open it in writing mode. To do this, we pass a mode of `'w'` into the `open` function:

```
>>> f = open('todo.txt', 'w')
```

This will open the file, creating it if it does not exist, and assigns the data containing the open file to the variable `f`. The file is then emptied or truncated, "cleaning the slate" for any data you will write. In the previous chapter, we could have passed in `'r'` for reading, but this is not necessary as it is the default (normal) behaviour when opening a file.

21.1 Learning to write

To write a string to the file, we use the `write` function:

```
>>> f.write('Do homework\n')
```

Subsequent calls to write will append data on the end, instead of overwriting:

```
>>> f.write('Make death star fully armed and operational\n')
```

The file will now look like:

```
Do homework
Make death star fully armed and operational
```

Note: If you forget the newlines `'\n'`, then the file will look like this:

```
Do homeworkMake death star fully armed and operational
```

After writing, always remember to `close` the file, or the data may not be written fully. This is why we always have to 'Safely Remove' our USB sticks before we physically remove them from the Raspberry Pi computer - data may still be in the process of being written to one or more files.

21.2 Writing lists

To write a list of lines, like that produced by `readlines`, we use `writelines`:

```
>>> f = open('todo.txt', 'w')
>>> f.writelines(['Do homework\n',
                  'Make death star fully armed and operational\n'])
>>> f.close()
```

21.3 Exercises

1. Write a program called `notes.py` which repeatedly asks the user for a sentence, stores it in a variable (a string), and writes it out to file each time (a file called `notes.txt` for instance). The loop should stop (break out) when the user types 'stop'. Use a `while` loop to keep on asking the user for input, and the file `write` function to write the sentence to the file. At the end of the program, the file should contain each sentence, one after the other, each taking up a line by itself.
2. Modify the program in exercise 1 so that each line is prefixed by the line number, so the first sentence is output with `"1.  "` added onto the beginning, with the second having `"2.  "` added on at the beginning, and so on.
3. Write a program called `cipher.py` so that it reads in a sentence, but instead of writing the sentence to a file, it writes the ordinal value of each character instead. To find out the ordinal value of a character, use the built-in `ord` function, passing in a single character at a time. You only need to read in one sentence, but you will need to step over the sentence using a `for` loop. To check your result, an input of `"abcdef"` should be stored in the file as `"97 98 99 100 101 102"` (i.e. the numbers from 97 for 'a', to 102 for 'f'), and an input of `"ABCDEF"` should be stored as "65 66 67 68 69 70" (i.e. the numbers from 65 to 70). Separate each number by a space as demonstrated above. Remember, the `str` function can take an integer and return a string in order for you to output it to file using the `write` function.

21.4 Things to remember

1. Use the built-in `open` function with the `'w'` mode to open files for writing.
2. Use the `write` function on the file variable to write a string to a file.
3. Use the `writelines` function on the file variable to write a list of lines to a file.
4. Don't forget to call the `close` function on the file when you have finished.

CHAPTER TWENTY TWO

CATCHING ERRORS

At the source of every error which is blamed on the computer you will find at least two human errors, including the error of blaming it on the computer.

— Unknown

22.0 It's broke

Computer users generally get annoyed when the programs they are using break easily. These programs are fragile, or brittle, and basically fall over or "crash" when something out of the ordinary happens. However, mistakes are made, unexpected things occur, programs are even used for purposes they were not intended. These things happen in real life, and computer programs should be resilient enough to keep on going, and not stop dead in their tracks. If the user has typed in the 13th month by accident, the program should tell the user so rather than falling apart. If the user has asked for a file to be opened that no longer exists, the program should inform the user instead of disappearing down a hole. If the user has asked for a number to be divided by zero, then the program should report the error rather than losing all the user's data up to that point. Unhappy users will stop using your program and switch to another that is not so delicate.

So the idea is for your program not to fail, but to handle errors gracefully and inform the user politely what has happened. This is a much more pleasant experience for the user of your program.

In Python, we encounter many errors, also known in Python as *exceptions*. We have arithmetic errors, where we cannot divide a number by zero:

```
>>> 200 / 0
Traceback (most recent call last):
  File "<stdin>‘"‘, line 1, in <module>
ZeroDivisionError: division by zero
```

Syntax errors, where the program is breaking the rules of the language:

```
>>> if broken = True:
  File "<stdin>", line 1
    if broken = True:
              ^
                                                                 Cont...
```

```
SyntaxError: invalid syntax
```

Errors with functions, such as passing in two arguments where only one is expected:

```
>>> input('I\'m', 'not working')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: input expected at most 1 arguments, got 2
```

And all sorts of other errors:

```
>>> 1 + 'two'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> error += 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'error' is not defined
>>> lst = ['more', 'bad', 'bugs']
>>> lst[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Some common Python exceptions:

Name	Common reason
`ZeroDivisionError`	You divided something by zero
`FileNotFoundError`	You tried to open a non-existent file
`ImportError`	You tried to import a module that does not exist
`NameError`	You have forgotten to define a variable
`SyntaxError`	Your syntax is wrong
`TypeError`	You tried to add a string and an integer
`ValueError`	You tried to convert a non-number string into an integer

Exceptions are helpful - they give the program information on what has happened so that it can do something about it. They tell us that something is broken, which we should fix. However, it is also nice to be able say "if there is an error, do this". In Python, this is called a `try`-`except` block.

22.1 try not to crash

Say we have some code that could produce an error:

```
your_age = int(input('Your age: '))
```

If the user types an integer number, it works fine. But if the user types something else - even a floating point number - we get an exception:

```
>>> your_age = int(input('Your age: '))
Your age: blah blah
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    your_age = int(input('Your age: '))
ValueError: invalid literal for int() with base 10: 'blah blah'
```

We could test the string to see if it is a number (`str.isdigit`), but instead we will try to handle the problem when it is occurs. This means that our code will be shorter and neater - otherwise every time you use a value that is unknown, it will need to be checked to see whether its contents are correct. This approach often results in a program where the error checking code takes up more space than the code that actually does the work! It also allows another part of your program to handle the errors instead of having to deal with them immediately - for example, a worker may not know what to do when a problem occurs, but his boss will! This last point is more obvious in larger programs rather than the small ones we will be writing here.

To handle an error when it occurs, we type `try:`, followed by our code, which should be indented, just like an `if` statement. Then we type `except:`, followed by the code we want run when there is an error:

```
try:
    your_age = int(input('Your age: '))
except:
    print('Err... No.')
```

Think of this as if the code will *try* to run a block of code - the call to `int` and `input` in this example - *except* if an error occurs then jump straight into this extra block of code. The extra block of code can be run at any time when an error occurs. This means if the initial block may or may not finish.

If the user types in a number as expected, all is well:

```
>>> try:
        your_age = int(input('Your age: '))
    except:
        print('Err... No.')

Your age: 99
>>> your_age                                                   Cont...
```

```
99
```

If the user types in something that is not expected, a message will be displayed instead:

```
>>> try:
        your_age = int(input('Your age: '))
    except:
        print('Err... No.')

Your age: blah blah blah
Err... No.
```

This works with any code:

```
>>> try:
        a = 2 / 0
    except:
        print('Maths says no!')

Maths says no!
```

22.2 Let's be specific about the problem

Using the `try`-`except` block as above works fine, but what if we only want to catch one type of exception? For instance, the following code contains an invalid variable, but we will never know, because the `except` is catching every exception, including the exception due to the invalid variable:

```
>>> i_do_exist = '123'
>>> try:
        a = int(i_do_not_exist)
    except:
        print('That was not a number!')

That was not a number!
```

The exception that we want to catch is a `ValueError`:

```
>>> int('abc')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'abc'
```

So, instead of typing `except:`, we can type `except ValueError:`. This will catch errors due to converting non-numbers, but not other errors:

```
>>> i_do_exist = '123'
>>> try:
```
Cont...

```
        a = int(i_do_not_exist)
    except ValueError:
        print('That was not a number!')

Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: name 'i_do_not_exist' is not defined
```

In general, you should name the type of exception you wish to handle. If you do not name the exception in order to catch all possible errors, then you may miss problems you should be handling differently. It is simply a case of best practice.

To find a full list of error types, you can type the *directory function* `dir` in the interactive interpreter:

```
>>> dir(__builtins__)
```

The errors that you can use to filter your errors are listed at the start - generally they have the word error at the end: `ArithmeticError`, `AssertionError`, `AttributeError`, `BaseException`, all the to `ZeroDivisionError` at the end.

22.3 Exercises

1. Add `try`-`except` blocks to your programs from *chapter 8*, printing out "That was not a number" when the user types in a non-number.
2. Write a program that takes two numbers, and divides one by the other. Print out a message when it tries to divide by zero using a `try`-`except` (the exception type is `ZeroDivisionError`).
3. Write a program called `openfile.py` which asks the user for a filename, opens the file and prints out its contents, just like in *chapter 20* on reading files. However, this time, if the file does not exist you should print out a suitable message such as "Sorry, that file does not exist". The error you need to trap is called `FileNotFoundError`.

22.4 Things to remember

1. Use `try`-`except` blocks to catch *exceptions*. Exceptions are errors or problems raised for the program to handle in a proper manner.
2. The code inside a `try`-`except` block is indented, like an `if` block.
3. It is best to name an exception type, to avoid surprises. You can find out what the exception type is by using the interactive interpreter to provoke the same error.

4. *Syntax errors* are when the program breaks the rules of the language. *Runtime errors* or *logical errors* are problems in the operation of the program itself.

APPENDIX

A

GLOSSARY

Elementary, my dear Watson

— Sherlock Homes

.py

The *file* extension that all python files should use.

argument

A value, or reference to a value, passed into a *function*.

ASCII

The American Standard Code for Information Interchange, a 7-bit character set and character encoding. Based on the Roman alphabet as used in modern English, the code is employed almost universally on computing machinery.

assignment

The process of defining a *variable* with a set value, e.g. `my_name = 'Fred'`.

boolean

A data type that can only have the values `True` and `False`.

boolean operator

logical operator

An operator, such as `and` that always results in a *boolean* result, or boolean-convertible result. For example, `a == 10` or `a < 0 or a > 100`.

built-in function

A *function* that comes with Python, so you do not need to implement it yourself, such as `round`, `print` and `input`.

call

Execute or run a *function* with the necessary *arguments*.

code block

A block is a piece of Python code that is executed as a unit.

comment

A piece of text acting as annotation or a description of the code. It is intended to be read by other people, and is ignored by Python from the `#` character to the end of the line.

comparison operator

An operator that takes two values and compares them, evaluating to a *boolean value*. Python comparison operators include `==`, `!=`, `<`, `>`, `<=` and `>=`. Refer to *appendix C* on Operators for more information.

constant

A *variable* which should not change. Constants are often used to make code more readable, by giving names to otherwise obscure values.

debugging

The process of finding and fixing bugs or defects that prevent the correct operation of a computer program or a system.

decrement

Decrease the value of a variable. For example: `num = num - 1`, or `num -= 1` for short.

dir

A *built-in function* that provides a directory listing of what is contained inside an object such as a module.

editor

A program for creating and making changes to *files*, especially text files.

equality operator

The `==` operator, that compares two objects and evaluates to `True` when they are the same. The opposite of the *inequality operator*.

evaluation

The process of computing a result from an *expression*.

exception

An interruption in normal processing, especially as caused by an error.

expression

A combination of actual values, variables, operators, calls to functions and even sub-expressions, to form a value that is computed or evaluated into its simplest form. For example, the expression `10 + 4 * 3 / 2` is evaluated to form the value `16.0`.

file

A resource for storing information, based on some kind of duration storage. It is usually *persistent*, so retains its state when the computer is turned off.

float

A floating point number is a number that has a fractional part, such as `1.78`, even if the fractional part is zero (e.g. `10.0`).

function

A section of code given a name that implements a task and *returns* a value, even if that value is empty.

function definition

A statement which creates a *function*, such as:

```
def add(a, b):
    return a + b
```

global variable

A variable that has been defined for use throughout a module, not just one function. A global variable can be used inside functions, but if it is to be modified, then it needs to be declared as `global` in advance.

IDLE

IDLE is the Python IDE.

immutable

A data type is immutable if its value cannot change. This means if we change the value of a variable, e.g. `num = num + 1` then the variable is moved to a new location containing the new data value. It is similar to creating a new box for the new data value, and moving the label of the variable to the new box. `int`, `float` and `str` are examples of mutable types, so if we change their value, we need to assign them to a variable name to save the new value, e.g. `number = number + 10`.

import

Including or making available one *module* inside another *module*.

increment

Increase the value of a variable, usually by `1`. For example: `num = num + 1`, or `num += 1` for short.

indentation

Beginning a line with one or more spaces. Used to distinguish *code blocks*.

index

Accessing a single item of a *sequence*, where `0` is the first item.

inequality operator

The `!=` operator, that compares two objects and evaluates to `True` when they are not the same. The opposite of the *equality operator*.

infinite loop

A *loop* which continues indefinitely.

input

Data that is entered by a source outside of the program, such as the user. In Python, this is most often the `input` function or a file.

integer

A whole number such as `18`. These numbers never have a fractional part.

integer division

floor division

An division where the result is rounded down to the nearest whole number (it evaluates to an *integer*).

integrated development environment

An integrated development environment (IDE) is a program that allows you to write, run and *debug* your code. Some IDE programs provide extra tools to allow you to write the code faster such as code highlighting and automatic code completion.

interactive interpreter

A programming environment that takes user input (e.g. a single line of code or a compound statement such as a loop), translating the code (i.e. interpreting it) into a form that the computer can execute directly, returning the result to the user.

iteration

looping

The process where a set of instructions or data are repeated.

keyword

A word with a special meaning. Python has many reserved keywords that it uses for its own purposes, such as `if`, `while`, `for`, `def`, etc., which you cannot use for any other purpose. To see Python's full list of keywords, then import the `keyword` module and type `keyword.kwlist` in the interactive interpreter.

keyword argument

An *argument* identified by a name e.g. `f(x=12, y=24)`.

list

A *sequence* of items, boxed together using the `[]` notation. The items can be of any type, such as integers, strings or even other lists.

local variable

A variable that has been defined within a function for use inside the function alone.

logical error

Where a program behaves in an unexpected or illogical way producing an undesired result, such as an adding program subtracting or a sorting program jumbling up the data.

matrix

A two-dimensional list or list of lists, such as:

```
matrix = [[1, 2, 3],
          [4, 5, 6],
          [7, 8, 9]]
```

Elements can be accessed by *indexing* twice:

```
>>> matrix[2][2]
9
```

module

A module is a file containing Python definitions and statements, e.g. a program that can be used by others.

modulus

The remainder when the first number is divided by the second.

mutable

A data type is mutable if its value can change after having been defined without having to move it in memory. The data is said to change in-place, so that the data in memory is altered rather than the variable name having to move to another memory location. A `list` is an example of a mutable type, so that we can write `my_list.append('fred')`.

operand

A value that is processed or used by an operator. For example, in the expression `10 + 20`, the `+` is an operator, whilst the values `10` and `20` are operands.

operator

A symbol that represents an action, such as addition, assignment or equality. In this book, we have learnt arithmetic, comparison and logical operators.

parameter

Data as received in by a *function*, given a name in-between the function definition's parentheses, which is local to the function itself.

persistent

Data that is saved onto secondary memory, such as an SD card, so that it can be used after the program is restarted.

pixel

A picture element is a single point of color. Many pixels are used to make up an image.

positional argument

An *argument* identified by its position in the list of arguments.

pseudo-code

A description of how your program works in simple English, without any unnecessary details of how it will be written.

random

Unpredictable in value.

return value

The data passed back from a called *function*.

run

execute

To perform the actions represented by the code.

runtime error

An error that occurs during the execution of the program, such as a program crashing with unexpected data (e.g. converting non-numerical value into an integer).

selection

The process of executing one piece (block) of code over another selectively.

sequence

An ordered list of objects. Usually a *list* or *string*.

slice

A portion or section of a *sequence*.

statement

A line of code or a code block that performs an action. A compound statement is a statement that performs a block of code *selectively* or *iteratively* (e.g. belonging to an `if` or `while` statement).

string

A sequence of characters such as letters, digits or symbols stored in encoded form.

string concatenation

The operation of joining multiple strings into one *string*. For example:

```
>>> "string" + " " + "concatenation"
'string concatenation'
```

string repetition

The operation of repeating a string multiple times to form a new *string*. For example:

```
>>> "repetition " * 5
'repetition repetition repetition repetition repetition '
```

syntax error

Where the code is breaking the rules of the language, such as misspelling the word `while`, or missing the colon `:` off the end of a `if` statement.

type

class

A data type defines the range of values and operations that can be performed on a piece of data. For example, integers are whole numbers, whether negative or positive, that can have mathematical operations performed on them, such as addition, subtraction, multiplication and division. Strings have a different set of operations available, such as capitalisation and splitting into words or sentences, although addition (joining together) and multiplication (repetition) are also available. Lists include operations that change, find and sort the values they contain. Other types of data described in this book are floating point numbers, booleans and files.

Unicode

A series of character encoding standards intended to support the characters used by a large number of the world's languages.

variable

A named memory location in which a program can store intermediate results and from which it can read and modify them.

Some definitions taken from https://en.wiktionary.org and https://www.python.org/.

APPENDIX B

PYTHON SUMMARY

I understand everything! Except that wig.

— Jack Sparrow

Pirates of the Caribbean: On Stranger Tides

This chapter summarises what we have learnt about the Python programming language. It is only a subset of the total language, but it is enough for you to do your coursework well.

B.0 Examples

This section lists a number of examples using various parts of the Python languages to achieve a variety of tasks. Refer to this when you need an example of how to do something, from printing out messages or numbers, making decisions, performing loops or catching errors.

Note that comments start with the `#` character.

The `print` *function* to print to the screen:

```
print("Hello, World!")
print('I will meet you at Fred\'s house')  # quote in quote
print('My name is', 'David', 'and my age is', 21)
print('I love Python ' * 1000)
print('two strings' + 'joined together')
```

Arithmetic:

```
# add, multiply, divide, subtract, power,
# remainder (modulo) - but not in that order!
print(3 + 4 * 10 / 2 - 5 ** 2 % 5)

print(3 + 4 * 5)     # will print 23
print((3 + 4) * 5)  # will print 35, brackets go first!
```

Using *variables* to store values in memory:

```
a = 10
b = a + 20
my_name = 'Fred'
```

Cont...

```
my_age = 101
print('hello there', my_name, 'you are', my_age, 'old')
```

Getting help in *IDLE*:

```
dir(__builtins__)  # list all the builtin functions
help(range)  # display help on the range function
help('modules')  # list all the supplied modules
dir(str)  # or dir(int), dir(float), dir(list)
help(str.isdigit)  # or help(str), help(float), help(list)
```

Calling *functions* (pieces of code that you can use easily):

```
# Putting () after function names means you are calling (invoking) it
print(abs(-123))  # prints 123 (makes number positive)
print(len('Hello there'))  # length of a sequence
print(ord('a'))  # print out ordinal number of a character
print(bin(183))  # prints binary 10110111
print(hex(183))  # prints hexadecimal B7
print(int('10110111', 2))  # prints decimal 183
```

Reading *input* from the user (using a function):

```
name = input('What is your name? ')
print('Hello there', name)
```

Converting from one *type* to another:

```
age_str = input('What is your age? ')
age = int(age_str)  # converts from a str to int
print('In 10 years time you will be', age + 10)
num_int = 10
num_float = float(num_int)
num_float2 = num_int * 10.0  # performs float calculation
day = 18
print('The date today is the', str(day) + 'th')  # join strings together
```

Using code from other programs (*modules*):

```
import turtle
turtle.circle(100)
dir(turtle)  # provide a directory listing of module
help(turtle.fillcolor)  # help on a particular item
```

Taking decisions using `if` statement (*operators* you can use are: `==`, `>`, `<`, `<=`, `>=`, `!=`, `in`, `and`, `or`, `not`):

```
a = 10
b = 20
if a > b:
    print('a is larger than b')                              Cont...
```

```
elif b > a:  # means "else if"
    print('b is larger than a')
else:  # catch all when other tests are False
    print('a and b are the same')

if a in range(10, 20):  # check if in range of numbers
    print('a is between 10 and 20!')
```

Looping (repeating the same code whilst a test is `True`):

```
a = 10
b = 20
while a < b:  # print out numbers between a and b
    print(a)
    a = a + 1  # or a += 1

while True:  # loop forever
    input_str = input('what is your name or quit? ')
    if input_str == 'quit':
        break  # escape from loop
    print('Hello there', input_str)
```

To generate *random* numbers, we use the `random` module:

```
import random
print(random.randrange(1, 100))  # random number between 1 - 100
print(random.random())  # random number between 0.0 - 1.0
```

Sequences:

```
import random

# Make a list of items using the square brackets []:
month_names = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',
               'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']

print(random.choice(month_names))  # print random month

print(month_names[0])    # prints Jan, indices start from 0
print(month_names[5])    # prints Jun
print(month_names[-1])   # prints Dec
print(month_names[-2])   # prints Nov
print(month_names[2:5])  # prints Mar, Apr, May

friends = ['Tom', 'Dick', 'Harry']
friends.append('Fred')  # append a new item onto list
print(len(friends))  # will now print out 4
```

`for` loops make stepping through sequences (or looping) very easy:

```
for letter in 'hello there':
    print(letter)

for num in range(100):  # range creates a list of numbers
    print(num)

for name in ['tom', 'dick', 'harry']:
    print('Hello there', name)

breakfast = ['bacon', 'egg', 'tomato', 'mushroom', 'bread']
for item in breakfast:
    print('Yum, I\'m having', item, 'for breakfast.')
```

Defining functions allows us to organise our code better:

```
def say_hello():
    print('hello')


def say_hello_times(times):
    print('hello' * times)


def square_number(number):
    return number * number


def lowest_highest(numbers):  # accepts a list of numbers
    lowest = min(numbers)
    highest = max(numbers)
    return lowest, highest  # returns two values

say_hello()
say_hello_times(10)
print(square_number(5))  # prints 25
low, high = lowest_highest([5, 10, 35, 15, 50, 20])
```

Reading from a *file* on disk:

```
my_file = open('textfile.txt')
for line in my_file:  # step through file line by line
    print(line)
my_file.close()
```

Writing to a file on disk:

```
# Open output file, and then write lines to file and close
output_file = open('test.txt', 'w')  # 'w' for writing
                                                    Cont...
```

```
output_file.write('first line\n')  # note newline symbol

lines = ['second line\n', 'third line\n']
output_file.writelines(lines)

output_file.close()
```

Catching *exceptions*:

```
try:
    number_str = raw_input('Give me a number: ')
    number = int(number_str)  # Try converting it to an int
    print('Another 10 added on is:', number + 10)
except:
    # If not an integer, an error will be thrown
    print('That was not a number!!')
# Now carry on as normal...
```

B.1 Operators

The most commonly used operators in Python.

B.1.0 Arithmetic operators

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Power
//	Floor division (Whole number division)
%	Modulus (remainder)

Examples:

```
>>> 10 + 20 * 2
50
>>> 100 / 4 - 3
22.0
```

Cont...

```
>>> 10 / 4
2.5
>>> 10 // 4
2
>>> 9 % 4
1
>>> 2 ** 8 + 1
257
```

B.1.1 Assignment operators

Operator	Description
`=`	Assign expression to variable
`+=`	Add expression to variable
`-=`	Subtract expression from variable
`*=`	Multiple expression to variable
`/=`	Divide expression into variable
`**=`	Performs power to variable
`//=`	Floor division into variable
`%=`	Modulus into variable

Examples:

```
>>> a = 10
>>> a += 1   # a is 11
>>> a -= 3   # a is 8
>>> a *= 2   # a is 16
>>> a /= 4   # a is 4.0
>>> a **= 3  # a is 64.0
>>> a //= 2  # a is 32.0
>>> a %= 25  # a is 7.0
```

B.1.2 Comparison operators

Operator	Description
`==`	Equal to
`!=`	Not equal to
`<`	Less than
`>`	Greater than
`<=`	Less than or equal to
`>=`	Greater than or equal to

Examples:

```
>>> a, b, c = 10, 15, 5
>>> a == b
False
>>> a != b
True
>>> a < b
True
>>> a >= c
True
```

B.1.3 Bitwise operators

Operator	Description
`<<`	Shift bits to the left
`>>`	Shift bits to the right
`&`	Bitwise and (set to 1 when both are 1) the bits together
`|`	Bitwise or (set to 1 when either are 1) the bits together
`~`	Return compliment - all the 1's and 0's are flipped
`^`	Bitwise exclusive or the bits together, unless both are 1 when the result is 0

Examples:

```
>>> 8 << 1
16
>>> 16 >> 2
4
>>> 127 & 15
15
>>> 10 | 5
15
>>> 10 ^ 15
5
```

B.1.4 Logical operators

Operator	Description
`and`	If both operands are true, then condition is true
`or`	If either of the operands is true, then the condition is true
`not`	Reverses the condition

Examples:

```
>>> a, b, c = 10, 15, 5
>>> a > b and a > c
False
>>> a > b or a > c
True
>>> not a == b
True
```

B.1.5 Membership operators

Operator	Description
`in`	Condition is true if the value or variable is contained in a sequence

Examples:

```
>>> 'a' in 'abc'
True
```

Cont...

```
>>> 'ab' in 'abc'
True
>>> 'abcd' in 'abc'
False
>>> num = 10
>>> num in [5, 10, 15, 20]
True
>>> num in [0, 20, 40, 60]
False
```

B.1.6 Operator precedence

The following table summarises the operator precedences in Python, from lowest precedence (least binding) to highest precedence (most binding). Operators in the same box have the same precedence - so they are evaluated from left to right. If in doubt, use parentheses `(` `)` to force a particular order!

Operator	Description
`or`	Boolean OR
`and`	Boolean AND
`not`	Boolean NOT
`in`, `not in`, `is`, `is not`, `<`, `<=`, `>`, `>=`, `!=`, `==`	Comparisons, including membership tests and identity tests
`|`	Bitwise OR
`^`	Bitwise XOR
`&`	Bitwise AND
`<<`, `>>`	Shifts
`+`, `-`	Addition and subtraction
`*`, `/`, `//`, `%`	Multiplication, division, remainder
`+x`, `-x`, `~x`	Positive, negative, bitwise NOT
`**`	Exponentiation
`x[index]`, `x[index:index]`, `x(arguments...)`, `x.attribute`	Subscription, slicing, call, attribute reference
`(expressions...)`, `[expressions...]`, `{key:  value...}`, `{expressions...}`	Binding or tuple display, list display, dictionary display, set display

B.2 Data types

Python allows more one type of data. Whilst calculators generally deal with only numbers, computers can store all sorts of data that can be interpreted in different ways. In addition to numbers (integers, floating point), programs can also process strings of characters, lists of values, files to store data on a more permanent basis. A bit of interpretation is required (the type itself) as all this data is invariably stored in binary.

Essentially, a data type determines the size of the data being stored, a set of permitted values, and the operations that can be performed on the data. Obviously, we can divide an integer by 2, but it would make no sense to divide a telephone number or someone's address by 2 - that operation is not permitted by strings.

The following table summarises the types of data we have introduced in this book, along with examples of their values and operations you can perform on them:

- **integers (int)** - integers are whole numbers, such as -1, 0, 1, 2, 3, 4, and so on. You convert to an integer data type using the `int` function. However, when doing so, make sure the argument passed in is an actual number, or else an error will be raised. If a floating point number is passed in, then the fractional part is dropped. If a string is passed in, ensure that the string contains a single whole number, without even a fractional part.

  The operations that can be performed on an integer mainly include the arithmetic operators, such as `+`, `-`, `*`, `/` (which results in a floating point answer), `//` (which results in an integer answer), `**` and `%`.

  For example:

  ```
  >>> 10 * 4 + 21 // 2    # results in 50
  >>> 10 * 4 + 21 / 2     # results in 50.5
  >>> 2 ** 8              # results in 256
  >>> 10 % 6              # results in 6 (remainder of 10 divided by 6)
  >>> int("123")          # results in number 123
  >>> int(123.5)          # also results in number 123
  >>> int(input('num?'))  # convert from number
  ```

- **floats (float)** - floating point numbers are fractional numbers that contains a whole number and a fractional part, such as -10.5, 0.0, 1.75, 15.0. You convert to a float data type using the `float` function. Make sure the argument passed in is an actual number, whether just a whole number or one already containing a fraction.

  The operators that can be performed on a floating point number are the same as integers.

  For example:

  ```
  >>> 10.0 * 4 + 21 // 2     # results in 50.5
  >>> 10.0 * 4 + 21 / 2      # also results in 50.5
  ```

Cont...

```
>>> 2.0 ** 8                  # results in 256.0
>>> 10.0 % 6                  # results in 4.0
>>> float("123")              # results in number 123.0
>>> float(123.5)              # results in number 123.5
>>> float(input('num?'))      # convert from input to number
```

- **strings (str)** - strings are a sequence of alphanumeric characters, such as those found on your keyboard. You can convert to a string data type using the `str` function. You can pass almost anything into this function in order to get its string representation.

  Although you can join and repeat strings by using the `+` and `*` operators, respectively, you can also use methods belonging to the string type or class to perform operations contained by the string value or object.

  For example:

```
>>> name = 'General Ike'
>>> capitalised = name.upper()
>>> new_name = name.replace('General', 'President')
>>> repeated = 'Fred' * 100
>>> joined_up = 'Fred' + 'Bloggs' + str(10)
>>> num_to_str = str(123.456)
```

- **booleans** - booleans are either `True` or `False`, and are produced when you compare values or combine comparisons. You can convert to a boolean data type using the `bool` function.

  For example:

```
>>> 10 < 20             # results in True
>>> run = True          # sets run to True
>>> run and 10 < 20     # results in True
>>> not run             # results in False
```

- **lists** - sequences of items of any type. A list is simply a sequence of items, such as a range of numbers, a list of names or months. You can convert to a list data type using the `list` function.

  For example:

```
>>> list('abcdefg')
>>> tunnels = 'tom dick harry'.split()
>>> spring = ['Mar', 'Apr', 'May']
>>> sorted(spring)
>>> list(reversed(spring))
>>> spring.append('gorilla')
>>> spring[0]
```

- **files** - a value or object pointing to an open file. You specify its name when opening the file, including its path if necessary, and optionally the mode in which you wish to open it

- `'r'` for reading, `'w'` for writing.

For example:

```
>>> my_file = open('shopping-list.txt')
>>> my_file.read()
>>> my_file.readlines()
>>> my_file.close()

>>> new_file = open('todo.txt', 'w')
>>> new_file.write('Finish computing coursework\n')
>>> new_file.writelines(['one\n', 'two\n', 'three\n'])
>>> new_file.close()
```

B.3 Different types of functions

In this section we will briefly describe the four general types of functions in Python:

B.3.0 Built-in functions

You can see the list of built-in functions by typing `dir(__builtins__)` in the interactive interpreter. Here is a list of the most useful ones, particularly the ones we have covered in this book with a brief note and example on each:

Name	*Description*	*Example*
`all`	*Returns* `True` *if all values in supplied sequence (iterable) are also* `True`	`all(my_list)`
`any`	*Returns* `True` *if any values in supplied sequence (iterable) are* `True`	`any(my_list)`
`abs`	*Returns the absolute (positive) value of an integer or float*	`abs(-10)`
`bin`	*Returns the binary number equivalent of the supplied integer as a string*	`bin(123)`

Continued on next page

`bool`	*Converts the supplied value into a boolean value*	`bool(1)`
`chr`	*Returns the character equivalent of the supplied ordinal (integer) number*	`chr(65)`
`dir`	*Returns a (directory) listing of the imported module*	`dir(math)`
`divmod`	*Divide one number by another, and returns the quotient and remainder in a sequence*	`divmod(10, 8)`
`enumerate`	*Supply a sequence, return a sequence of items paired with their index from* `0`	`enumerate('abc')`
`exit`	*Exit your program early (same as* `quit`*)*	`exit()`
`float`	*Converts the supplied value into a floating point (fractional) number*	`float('1.5')`
`hex`	*Converts the supplied value into a hexadecimal value as a string*	`hex(127)`
`help`	*Provides help on the supplied item*	`help(input)`
`id`	*Returns the memory address of the supplied name*	`id(my_num)`
`input`	*Waits on the user to type something, and return sequence of characters as a string*	`input('name?  ')`

Continued on next page

`int`	*Convert the supplied value into an integer number*	`int('100')`
`len`	*Returns the length of the supplied sequence (e.g. string or list)*	`len('fred')`
`list`	*Converts the supplied value into a list*	`list('xyz')`
`max`	*Returns the maximum value from the supplied sequence*	`max(1, 2, 3)`
`min`	*Returns the minimum value from the supplied sequence*	`min(5, 1, 3)`
`oct`	*Converts the supplied value into an octal value as a string*	`oct(25)`
`ord`	*Returns the supplied character into an ordinal (integer) value*	`ord('a')`
`open`	*Open the supplied filename and return the opened file*	`open('scores.txt)`
`pow`	*Calculate the power of one number to another and return the result*	`pow(2, 8)`
`print`	*Print out or display the supplied string or list of items*	`print('Hello')`
`range`	*Provide a range of integers, with a set start, stop and step*	`range(10, 20, 2)`
`reversed`	*Reverse the order of a supplied sequence*	`reversed('abcde')`

Continued on next page

`round`	*Round the supplied floating point number to the specified precision*	`round(4.75)`
`sorted`	*Return the supplied sequence in order*	`sorted('azgdbdc')`
`str`	*Convert the supplied value into a string*	`str(100)`
`sum`	*Sum or add up the supplied sequence of numbers returning the result*	`sum([1, 3, 5, 7])`
`type`	*Return the type of the supplied item, e.g.* `int`, `float`, `str`, `bool`	`type('bob')`
`zip`	*Zips up or combines two or more supplied sequences*	`zip('abc', 'def')`

The full list will include other built-in items as well (e.g. exception types). These can be called by any Python program without having to import anything else – they are built-in to the language itself.

B.3.1 Local functions

You can define your own functions in your Python module by using the `def` keyword. These can then be called from within your own program by simply using the name of the function itself, similar to a built-in function. For example, here is a function that accepts a number and returns its square:

```
def square_number(number):
    return number * number
```

Which can then be called as follows:

```
square_number(5)  # returns 25
```

B.3.2 Imported functions

You can use functions in other modules by importing them first. For example, to use functions inside the `math` module, you can do the following:

```
import math
math.sqrt(100)
```

You need to write module name followed by a period `.` before the name of the function when calling it. You can print out a directory listing of what a module contains by performing a 'dir' on its name, for example:

```
>>> dir(math)
```

B.3.3 Functions belonging to a type ("class methods")

A particular set of values is called a type (integers, floating point numbers, strings, files) or a class. These types wrap up the data they contain, and also offer functions that operate on that data. This allows the data and related code to live in one place. For example, once you have defined a string, you can calls its type or class methods (functions) to perform a number of operations on that string value:

```
>>> message = 'the quick brown fox jumps over the lazy dog'
>>> message.upper()  # returns the uppercase version
>>> message.split()  # returns a list of words
>>> message.replace('fox', 'coyote')  # replaces one word with another
>>> message.count('o')  # returns how often one string is in another
>>> message.startswith('the')  # does string start with this?
```

And so on. You need to write the variable name (which refers to a piece of data, or object, belonging to a particular type), followed by a period `.` before the name of the function when calling it. You can list the functions that a type contains by performing a `dir` on its name, for example:

```
>>> dir(int)
>>> dir(float)
>>> dir(str)
```

This will show that some types have functions that are not relevant to other types. For example, floats have a function called `is_integer` which returns `True` if it is a whole number, `False` if not. Strings have functions such as `lower`, `split`, `title`, `upper`, which are relevant to strings of characters, but not numbers and files. These methods are bound up with the data they work on, so only relevant functions are offered with the type of data the variable refers to.

Keep in mind that if you call a function to change its value, some functions will change the data referred to by the variable name, and others will return a new value. This means that some types can change their values (e.g. lists, where you can `append`, `insert` or `remove` values), and others cannot so give you a new value (e.g. integers, floats and strings). Therefore, all the calls on the `message` variable above, being a string, would return a new value if you tried to change it (e.g. changing the string to uppercase or replacing values). You could then assign this new value to a variable name using the assignment `=` operator.

Here are a few more examples to show this difference between types that allow the value to change (*mutable*) and types that do not (*immutable*):

```
>>> hobbits = ['Bilbo', 'Frodo', 'Sam']
>>> hobbits.append('Pippin')  # changes value
>>> hobbits.remove('Bilbo')   # changes value

>>> num = 10
>>> num = num + 1   # new value
>>> num = num * 10  # new value

>>> name = 'The 11th Doctor'
>>> big_name = name.upper()  # new value
>>> new_name = name.replace('11th', '12th')  # new value
```

Functions that change the existing value, rather than returning a new one, are known as in-place operations - they change the data where it lives or is placed in memory, rather than creating a new value with different contents. It is a subtle but vital point!

B.4 Program layout

When a program gets past a few lines, including its own function definitions and the like, you need to structure your program in an orderly fashion to avoid it becoming messy. The general structure of a Python program should follow this pattern:

```
# firstly import any modules you need, e.g.
import math

# define constant variables (variables whose values don't change), e.g.
MULTIPLIER = 2


# function definitions, e.g.
def get_integer():
    number = input('Give me a number: ')
    return int(number)


def double_up(num):
    return num * MULTIPLIER

# main code, e.g.:                                                 Cont...
```

```
number = get_integer()

print('Number doubled is:', double_up(number))
print('and its factorial is:', math.factorial(number))
print('Bye for now.')
```

APPENDIX

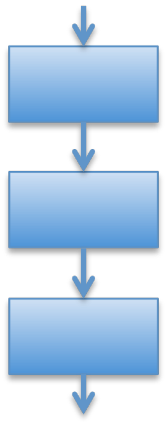
C

PROGRAM FLOW

The river of human nonsense flows on forever.

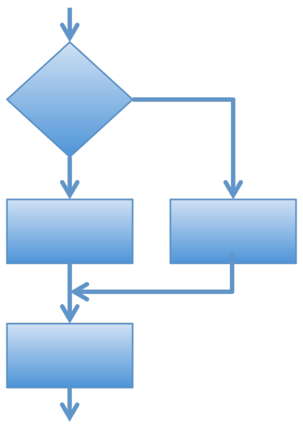
— G.K. Chesterton

C.0 Sequence



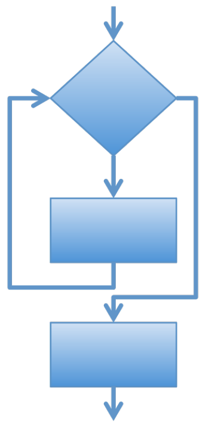
```python
name = input('What is your name? ')

print('Hello there', name)

print('Here is a sum', 10 + 20)
```

C.1 Selection



```python
num = int(input('Number please: '))

if num < 0:

    print('Negative!')

else:

    print('Positive')

print('Off we go again...')
```

C.2 Iteration



```
num = 0

while num < 10:

    print(num)

    num = num + 1

print('Off we go again... ')
```

APPENDIX

D

COMMON MISTAKES

To err is human, but to really foul things up you need a computer.

— Paul R. Ehrlich

D.0 Forgetting to open / close quotation marks

Do not forget to close your quoted strings with a quotation mark at both the beginning and end. Moreover, you must be consistent, so if you start with a single or double quotation mark, then you must finish with the same type of quotation mark.

Bad	Good
`print("Hello, World!)`	`print("Hello, World!")`
`print(Hello!')`	`print('Hello!')`
`print('Your name is, name)`	`print('Your name is', name)`
`print('Name', name, 'age, age)`	`print('Name', name, 'age', age)`

D.1 Using quotation marks insides quotes

If you include a quotation mark inside a string, then you must use the escape sequence `\'` or `\"`, otherwise Python will assume you have closed off the string, thus leaving the remaining text outside the quotation marks.

Bad	Good
`print('St. Michael's School')`	`print('St. Michael\'s School')`
`print("His name was "fred"")`	`print("His name was \"fred\"")`

D.2 Not using commas between items

You must use a `,` comma between each item, whether you are printing, defining a list or defining more than one variable from a sequence of values.

Bad	Good
`print('Hello your name is' name)`	`print('Hello your name is', name)`
`print(10 20 30 40)`	`print(10, 20, 30, 40)`
`months = ['jan' 'feb' 'mar']`	`months = ['jan', 'feb', 'mar']`
`num1 num2 = 10 20`	`num1, num2 = 10, 20`
`num1 num2 = num2 num1`	`num1, num2 = num2, num1`

D.3 Spelling or capitalisation mistakes

You must be consistent in using the names that have been defined, including the way the names are spelt and capitalised.

Bad	Good

`pront(10 + 20)`
`Print(10 + 20)`
	`print(10 + 20)`
	`print(10 + 20)`

`number = 10`
`print(numbre)`
	`number = 10`
	`print(number)`

`turtle.pencolour("red")`
`turtle.beginfill()`
	`turtle.pencolor("red")`
	`turtle.begin_fill()`

`s = 'good morning'`
`print(s.titel())`
	`s = 'good morning'`
	`print(s.title())`

D.4 Using variables before defining them

Before making use of a variable, you must define it to some value. Use the value `None` if you do no know what this value is going to be in advance (or `''` for an empty string, or `[]` for an empty list).

Bad	Good

`num1 = 10`
`print(num1, num2)`
	`num1, num2 = 10, 20`
	`print(num1, num2)`

`while count < 10:`
`    print(count)`
`    count = count + 1`
	`count = 0`
	`while count < 10:`
	`    print(count)`
	`    count = count + 1`

D.5 Using invalid variable names

Variable names must begin with either an upper or lowercase letter, or an underscore `_` character. Do not include symbols in your names.

Bad	Good
`100_num = 100`	`num_100 = 100`
`a$ = 10`	`a = 10`
`first number = 123`	`first_number = 123`

D.6 Setting variables

Variables are defined by placing the variable name on the left, with an assignment operator in the middle, followed by the expression to store on the right. If you are defining two variables at the same time (in the second example below), then you must use the assignment operator between each of them so that they are assigned to the same value.

Bad	Good
`101 = number`	`number = 101`
`num1, num2 = 5`	`num1 = num2 = 5`

D.7 Using modules before importing them

Before you can use anything defined inside another module, you must import it first. This includes even listing its contents by using the `dir` function.

Bad	Good

`num = random.randint(1, 100)`	`import random`
	
	`num = random.randint(1, 100)`

Another common mistake is to save your program with the same name as a module you are importing, so it will import your own program instead. For example, if you are using the `random` module, then do not call your program `random.py`, or if you are using the `turtle` module then do not call your program `turtle.py`.

D.8 Not converting to integers when performing arithmetic

Strings must be converted into numbers, whether integers or floats, before they are used in arithmetic calculations. Use the `int`, `float` and `str` functions to convert between these types of data.

Bad	Good

`age = input('Age? ')`
 	
`print(age + 10)`
	`age = input('Age? ')`
	
	`print(int(age) + 10)`

D.9 Forgetting the colon `:` at the end of compound statements

Any compound statement, whether an `if`, `while`, `for`, `def` or `try`, must have a colon `:` symbol at the end of the line to indicate that the code block that follows belongs to it. For example, if an `if` statement is used, then the code block is only executed if the condition following the `if` keyword evaluates to `True`.

Bad	Good

`if num > 10`
`    print(num)`
	`if num > 10:`
	`    print(num)`

`for num in range(10)`
`    print(num)`
	`for num in range(10):`
	`    print(num)`

`while num < 10`
`    print(num)`
`    num = num + 1`
	`while num < 10:`
	`    print(num)`
	`    num = num + 1`

`def say_hello()`
`    print("Hello!")`
	`def say_hello():`
	`    print("Hello!")`

D.10 Using the assignment operator instead of comparison operator

When comparing values, you must use one of the *comparison operators*. The *assignment operator* `=` is used to define variables. The equals operator `==` is used to compare an expression on the left with an expression on the right, resulting in a boolean `True` or `False` result.

Bad	Good

`if num = 10:`
`    print(num)`
	`if num == 10:`
	`    print(num)`

D.11 Not forming expressions properly

The expressions below on the left (taking the first example) were intended to compare the variable `ch` against either `'A'` or `'B'`, and execute the subsequent code block if this is the case. Unfortunately, it only compares `ch` against `'A'`, and then checks whether the letter `'B'` is not empty (which it isn't). It is equivalent to `(ch == 'A') or ('B')`, with each side of the expression being evaluated separately, and then combined together with the `or` operator. This means that the code block will always execute, as this expression is always `True`. To compare a variable against two separate values, you need to do both comparisons individually, such as on the right of the table.

Bad	Good
`if ch == 'A' or 'B':`	`if ch == 'A' or ch == 'B':`
`if a and b > 10:`	`if (a > 10) and (b > 10):`

D.12 Unnecessarily testing expressions against `0`, `True`, `False`, `None` or `""`

The examples below on the left will work perfectly well, but contain code that is unnecessary. If you wish to compare whether a value is non-zero, is not empty, or is `True`, you simply need to test the variable name alone. You would not type in the expression `a > 0 == True`, but `a > 0` instead. Therefore, do not type in an expression such as `a == True`, but simply test against `a` instead.

Bad	Good
`if a != 0 and b != 0:`	`if a and b:`
`if a == 0 and str == '':`	`if not a and not s:`
`if a == True:`	`if a:`
`if a == False:`	`if not a:`

D.13 Getting the number of brackets wrong in an expression

Always ensure that the same number of left brackets `[` or `(` matches the number of right brackets `]` or `)`, respectively.

Bad	Good
`print(2 + (4 * (8 / (10 / 5))`	`print(2 + (4 * (8 / (10 / 5))))`
`print(((10 + 20) ** 2)`	`print(((10 + 20) ** 2))`
`print((10 + 20) / 8 / 4))`	`print((10 + 20) / (8 / 4))`
`nums = [10, 20, [40, 50]`	`nums = [10, 20, [40, 50]]`

D.14 Indexing past the end of lists

Do not index past the end of a sequence, which ranges from 0 up until the length of the list but one (i.e. 0 to 2, inclusive, in the example below).

Bad	Good

`names = ['fred', 'bob', 'tom']`
 	
`print(names[3])`
	`names = ['fred', 'bob', 'tom']`
	
	`if 3 < len(names):`
	
	`    print(names[3])`

D.15 Forgetting the brackets when calling a function

Always include parentheses when invoking (*calling*) a function. Simply typing the name of the function will provide you with its memory location - it will not actually run it!

Bad	Good
`int`	`int()`
`turtle.begin_fill`	`turtle.begin_fill()`
`print(math.sqrt 144)`	`print(math.sqrt(144))`
`print(int '1010101', 2)`	`print(int('10101010', 2))`

D.16 Using variable names reserved by Python

Do not use reserved keywords as names in your code. To see Python's full list of keywords, then import the `keyword` module and type `keyword.kwlist` in the interactive interpreter.

Bad	Good
`for = 10`	`for_num = 10`
`if = 100`	`if_num = 100`
`while = 'fred'`	`while_str = 'fred'`

D.17 Getting the indentation wrong

Python uses *indentation* (the practice of "pushing in" your code from the left-hand side) to define blocks of code. Ensure that each block of code is exactly indented in the same manner (i.e. they start in the same column as other lines at the same level or indentation). It is recommended practice to use an indentation of 4 spaces for each code block.

Bad	Good

`print('Hello there')`
`    print('How are you?')`
	`print('Hello there')`
	`print('How are you?')`

`for num in range(10):`
`print(num)`
	`for num in range(10):`
	`    print(num)`

`if a == 10:`
`    print('a is 10')`
`  print('where is b?')`
	`if a == 10:`
	`    print('a is 10')`
	`    print('where is b?')`

`if a == 10:`
`    print('a is 10')`
`print('where is b?')`
`else:`
`    print('and what might c be?')`
	`if a == 10:`
	`    print('a is 10')`
	`    print('where is b?')`
	`else:`
	`    print('and what might c be?')`

D.18 Using an `elif` or `else` without an `if`

A selection statement must always include an `if` statement, with the `elif` and `else` statements being optional (i.e. you do not have to include them).

Bad	Good

`a = 10`
`elif a > 10:`
`    print('larger than ten')`
`else:`
`    print('something else')`
	`a = 10`
	`if a == 10:`
	`    print('a is ten')`
	`elif a > 10:`
	`    print('larger than ten')`
	`else:`
	`    print('something else')`

D.19 Placing a condition after an `else`

The `else` line within an `if` statement can be read as *otherwise do this* - or if all of the tests above are `False` then do this instead. It is not meant to include a test of its own.

Bad	Good

`a, b = 10, 20`
`if a > b:`
`    print('a is larger')`
`elif a < b:`
`    print('b is larger')`
`else a == b:`
`    print('a and b are the same')`
	`a, b = 10, 20`
	`if a > b:`
	`    print('a is larger')`
	`elif a < b:`
	`    print('b is larger')`
	`else:`
	`    print('a and b are the same')`

D.20 Getting stuck in a loop

Your loops should always include a way out, whether via the condition at the top eventually changing from `True` to `False`, or having a `break` statement which is performed selectively.

Bad	Good

`num = 0`
 	
`while num < 10:`
 	
`    print(num)`
	`num = 0`
	
	`while num < 10:`
	
	`    print(num)`
	
	`    num = num + 1`

`while True:`
 	
`    name = input('Name? ')`
 	
`    print(name)`
	`while True:`
	
	`    name = input('Name? ')`
	
	`    if name == 'quit':`
	
	`        break`
	
	`    print(name)`

APPENDIX

E

EXTRA EXERCISES

Now is the winter of our discontent

— William Shakespeare

Richard III

These exercises are designed to allow you to practice your programming skills learned during class. Please complete each each task and save in a folder called `practice` on your USB stick to show the teacher when finished. However, you need not complete the tasks in the order written - if you get stuck with one, move onto another and return to the first task later.

This work must be done on your own without help or assistance from others in the class. However, you may consult your previous work, or the books and examples provided by the teacher, as you wish.

E.0 `welcome.py`

Write a program that asks for the user's name and a number, and then prints out their name that number of times.

For example:

```
What is your name? Snoopy
How many times should I print your name: 5
Snoopy Snoopy Snoopy Snoopy Snoopy
```

E.1 `oddoreven.py`

Write a program that inputs a number, and then tells the user whether the number is odd or even.

For example:

```
Please input a number: 17
The number 17 is odd.
```

Cont...

```
Please input a number: 42
The number 42 is even.
```

E.2 `century.py`

Write a program to input the user's name and age, and then print out a greeting including the user's name, and the year in which the user will be 100 years old. Your program can assume the current year is 2015.

For example:

```
What is your name: Fred
How old are you? 99
Hello Fred you will be 100 in 2016

What is your name: Bob
How old are you? 20
Hello Bob you will be 100 in 2095
```

E.3 `circlearea.py`

Write a program that asks for the diameter of a circle, and prints out its area. The area can be calculated as follows:

$$area = \pi * r^2$$

Hint: For π, import the module `math` and use `math.pi`.

For example:

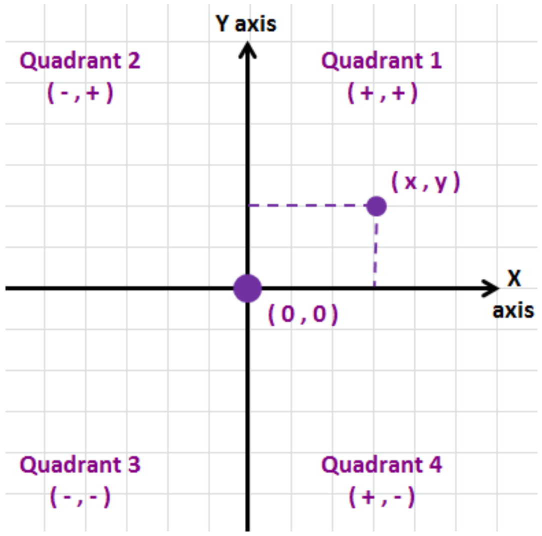
```
Please enter the diameter of the circle: 12
The area is: 113.10
```

E.4 `drawline.py`

Write a program to draw a line on the screen given two x and y coordinates, using the `turtle` module.

Hint: The turtle window has its 0,0 point at the centre, with increasing x and y in the right and up direction, respectively. In other words, x starts at 0 in the centre and goes negative towards the left, and positive towards the right. y starts at 0 in the centre and goes negative towards the bottom and positive towards the top. This is somewhat like graph paper.

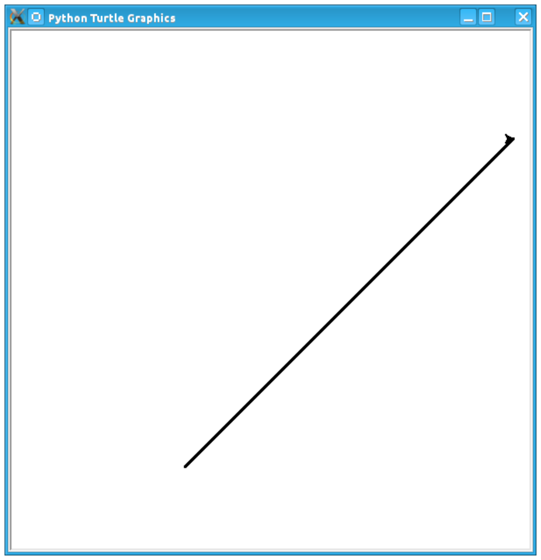
This is the turtle coordinate system illustrated (reference: http://101computing.net):



You will need to create the turtle window using the `Turtle` function, to pick your pen up using the `up` function, and move using the `goto` function.

For example (drawn with a thicker pen using the `pensize` function on a 500x500 window):

```
From where should the line start? -100 -200
And to where should the line end? 275 175
```



E.5 countingdown.py

Write a program to input a number from the user and then print all the numbers from this down to zero. Make sure the number is positive!

For example:

```
Please input a number: 10
10
9
8
7
6
5
4
3
2
1
0
```

E.6 `span.py`

Write a program to ask for two numbers, and print the numbers that span from the first up to the second. Care should be taken in the case that the second number is lower than the first number - you should always count up.

For example:

```
Please input your first number: 10
Please input your second number: 20
The span of numbers are:
10
11
12
13
14
15
16
17
18
19
```

E.7 `squares.py`

Write a program to ask for two numbers, and print out the square of the numbers that span from the first number up to the second. Care should be taken in the case that the second number is lower than the first number - you should always count up.

For example:

```
Please input your first number: 10
Please input your second number: 20
The square numbers between those numbers are:
100
121
144
169
196
225
256
289
324
361
```

E.8 `headstails.py`

Write a program to ask the user how many times the program should flip a coin, and count how many times the coin landed on heads and tails.

Hint: Use `random.choice` function with a parameter of `["heads", "tails"]` to choose between the two options.

For example:

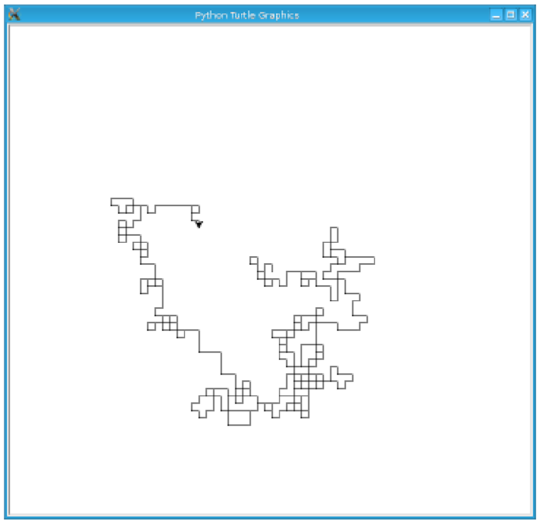
```
How many times should I flip the coin? 1000
The number of heads totalled 459 and the number of tails totalled 541
```

E.9 `randompathtracer.py`

Write a program, using turtle, that asks the user for a number of steps. The program should then loop, and at each step, randomly turn the turtle left by 90 degrees, right by 90 degrees or not turn at all. It should then go forward by 10 pixels.

For example:

```
How many steps should I draw? 500
```



E.10 ascii.py

Write a program to print out a section of the *ASCII* table. It should print out the decimal, binary, hexadecimal and character representation for the values from 32 to 127, inclusive. Use the built-in `bin` function to get the binary value, `hex` for the hexadecimal value and `chr` to get the character representation.

For example:

```
32    0b100000    0x20
33    0b100001    0x21    !
34    0b100010    0x22    "
35    0b100011    0x23    #
36    0b100100    0x24    $
37    0b100101    0x25    %
38    0b100110    0x26    &
39    0b100111    0x27    '
40    0b101000    0x28    (
41    0b101001    0x29    )
42    0b101010    0x2a    *
43    0b101011    0x2b    +
44    0b101100    0x2c    ,
45    0b101101    0x2d    -
46    0b101110    0x2e    .
47    0b101111    0x2f    /
48    0b110000    0x30    0
49    0b110001    0x31    1
50    0b110010    0x32    2
51    0b110011    0x33    3
```

Cont...

```
52    0b110100    0x34    4
53    0b110101    0x35    5
54    0b110110    0x36    6
55    0b110111    0x37    7
56    0b111000    0x38    8
57    0b111001    0x39    9
58    0b111010    0x3a    :
59    0b111011    0x3b    ;
60    0b111100    0x3c    <
61    0b111101    0x3d    =
62    0b111110    0x3e    >
63    0b111111    0x3f    ?
64    0b1000000    0x40    @
65    0b1000001    0x41    A
66    0b1000010    0x42    B
67    0b1000011    0x43    C
68    0b1000100    0x44    D
69    0b1000101    0x45    E
70    0b1000110    0x46    F
71    0b1000111    0x47    G
72    0b1001000    0x48    H
73    0b1001001    0x49    I
74    0b1001010    0x4a    J
75    0b1001011    0x4b    K
76    0b1001100    0x4c    L
77    0b1001101    0x4d    M
78    0b1001110    0x4e    N
79    0b1001111    0x4f    O
80    0b1010000    0x50    P
81    0b1010001    0x51    Q
82    0b1010010    0x52    R
83    0b1010011    0x53    S
84    0b1010100    0x54    T
85    0b1010101    0x55    U
86    0b1010110    0x56    V
87    0b1010111    0x57    W
88    0b1011000    0x58    X
89    0b1011001    0x59    Y
90    0b1011010    0x5a    Z
91    0b1011011    0x5b    [
92    0b1011100    0x5c    \
93    0b1011101    0x5d    ]
94    0b1011110    0x5e    ^
95    0b1011111    0x5f    _
96    0b1100000    0x60    `
97    0b1100001    0x61    a
98    0b1100010    0x62    b
```

Cont...

```
99    0b1100011    0x63    c
100    0b1100100    0x64    d
101    0b1100101    0x65    e
102    0b1100110    0x66    f
103    0b1100111    0x67    g
104    0b1101000    0x68    h
105    0b1101001    0x69    i
106    0b1101010    0x6a    j
107    0b1101011    0x6b    k
108    0b1101100    0x6c    l
109    0b1101101    0x6d    m
110    0b1101110    0x6e    n
111    0b1101111    0x6f    o
112    0b1110000    0x70    p
113    0b1110001    0x71    q
114    0b1110010    0x72    r
115    0b1110011    0x73    s
116    0b1110100    0x74    t
117    0b1110101    0x75    u
118    0b1110110    0x76    v
119    0b1110111    0x77    w
120    0b1111000    0x78    x
121    0b1111001    0x79    y
122    0b1111010    0x7a    z
123    0b1111011    0x7b    {
124    0b1111100    0x7c    |
125    0b1111101    0x7d    }
126    0b1111110    0x7e    ~
127    0b1111111    0x7f
```

E.11 vowel.py

Write a program to input a character, and tell the user whether it is a vowel or not (i.e. one of these characters - a, e, i, o or u). Make sure only a single character has been input.

For example:

```
Please type one character from the alphabet: a
The letter a is a vowel!

Please type one character from the alphabet: z
The letter z is not a vowel!

Please type one character from the alphabet: E
The letter E is a vowel!
```

E.12 prayers.py

Write a program to offer an index of prayers, ask for a choice of one of them - or none at all - and print out that prayer in full. Your choice of prayers is up to you.

For example:

```
The choice of prayers is as follows:

1) Apostles Creed, 2) Our Father, 3) Hail Mary, 4) Glory Be,
5) Hail Holy Queen, 6) Exit

What is your choice? 3
Hail Mary, full of grace, the Lord is with thee; blessed art thou
amongst women, and blessed is the fruit of thy womb, Jesus. Holy Mary,
Mother of God, pray for us sinners, now and at the hour of death. Amen
```

E.13 palindrome.py

Write a program that will input a word, and then inform the user whether the word is a palindrome or not (i.e. words that when reversed, are the same). So the words "nun", "radar" and "kayak" are palindromes.

For example:

```
Input a word: bob
The word bob is a palindrome!

Input a word: fred
The word fred is not a palindrome
```

E.14 histogram.py

Write a program that will accept a list of numbers and then draw a histogram using the star `*` character.

For example:

```
Input the numbers for the histogram: 1, 3, 5, 4

Here is the histogram for the numbers 1, 3, 5, 4:
*
***
*****
```

Cont...

```
****
```

E.15 `length.py`

Write a program to input a list, and print out how long that list is. Use `sentence.split` to split the sentence returned by `input` into a list of items. For example:

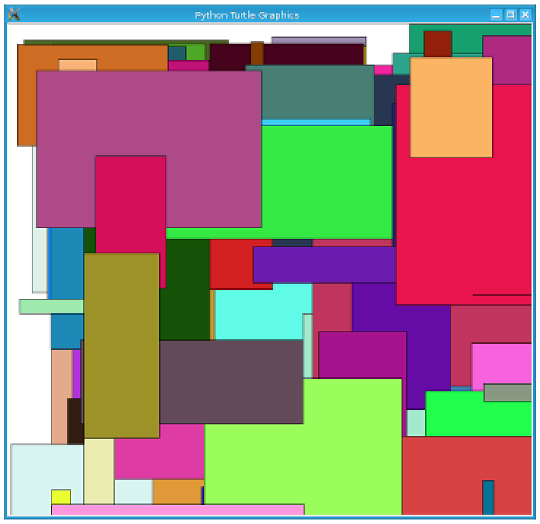
```
Please input your sentence: a b c 1 2 3
The number of items in your sentence is: 6

Please input your sentence: monday tuesday wednesday
The number of items in your sentence is: 3
```

E.16 `turtleboxes.py`

Write a program to draw 100 rectangles of a random length and width, and a random color and at random positions in the turtle window.

Hint: You will need to use the `turtle` module, and functions from the turtle module such as `goto`, `up`, `down`, `forward`, `right` (or `left`), `begin_fill`, `end_fill` and `fillcolor`. Use the help system to find out how to call these functions.

For example:



E.17 `longest.py`

Write a program to input a sentence and then print out which word is the longest.

For example:

```
Please input your sentence: The quick fox jumped over the lazy dog
The longest word in that sentence is: jumped
```

E.18 `reverse.py`

Write a program to input a sentence and then print it out in reverse.

For example:

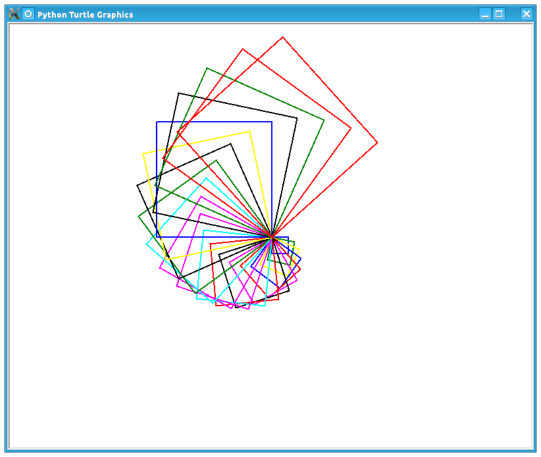
```
Please input your sentence: mary had a little lamb
The reverse of your sentence is: bmal elttil a dah yram
```

E.19 `twist.py`

Write a program that draws a number of squares, using the `turtle` module, each one larger than the last and with the drawing turtle turning after each square. Each square should also be a different color - use the `random.choice` function to select from a variety of colors.

The first square should have sides of 25 pixels in length, with each succeeding square being 10 pixels longer on each side. The turtle should turn 10 degrees to the right after every square.

For example:



E.20 **factorial.py**

Write a program to input a number, and then print out the factorial of that number. The factorial is all the numbers up to and including the actual number multiplied together.

For example:

```
Please input your number: 6
The factorial of 6 is 720

Please input your number: 10
The factorial of 10 is 3628800
```

E.21 **quiz.py**

Write a program to ask the user a number of questions, with multiple choice answers, and then print out their score at the end. You should ask 5 questions in total. You are free to make up your own questions.

For example:

```
Welcome to the QUIZ program!

Question 1: Who won the football world cup in 1970?
a) England b) Brazil c) West Germany d) Italy
Your answer: b
Correct!

Question 2: Who won the Formula 1 world championship in 2008?
a) Michael Schumacher b) Fernando Alonso c) Lewis Hamilton d) Niki Lauda
Your answer: c
Correct!

Question 3: Who has won the most Wimbledon tennis titles?
a) Roger Federer b) Boris Becker c) Andre Agassi d) Pete Sampras
Your answer: d
Incorrect, it is a!

Question 4: Who has won the most Rugby World Cups?
a) South Africa b) Australia c) New Zealand d) all three
Your answer: b
Incorrect, it is d!

Question 5: Who has won the most Olympic medals?
a) Michael Phelps b) Carl Lewis c) Usain Bolt d) Steve Redgrave
Your answer: a
```

Cont...

```
Correct!

Well done - you got 3 out of 5!
```

E.22 hangman.py

Write a program to implement a simple hangman game. Give the user 11 tries, and you can draw the hangman as you go along as follows (piece by piece):

```
 -------
 |/    |
 |     O
 |    /|\
 |    / \
---
```

However, this part of drawing the hangman is optional as it makes the program more complicated.

Hint: You will need three strings, one for the word to guess (which selects one from the word list below randomly using the `random.choice` function), one containing the letters guessed so far, and one for the letters not in the word being guesses. You can add onto a string by doing the following:

```
string_name = string_name + character_entered
```

You may use the following as your word list, or create your own:

```
WORD_LIST = ['adult', 'aeroplane', 'air', 'aircraft', 'airforce',
             'airport', 'album', 'alphabet', 'apple', 'arm', 'army',
             'baby', 'backpack', 'balloon', 'banana', 'bank',
             'barbecue', 'bathroom', 'bathtub', 'bed', 'bed', 'bee',
             'bible', 'bible', 'bird', 'bomb', 'book', 'boss', 'bottle',
             'bowl', 'box', 'boy', 'brain', 'bridge', 'butterfly',
             'button', 'cappuccino', 'car', 'carpet', 'carrot', 'cave',
             'chair', 'chess', 'chief', 'child', 'chisel', 'chocolates',
             'church', 'church', 'circle', 'circus', 'circus', 'clock',
             'clown', 'coffee', 'comet', 'compass', 'computer',
             'crystal', 'cup', 'cycle', 'database', 'desk', 'diamond',
             'dress', 'drill', 'drink', 'drum', 'dung', 'ears', 'earth',
             'egg', 'electricity', 'elephant', 'eraser', 'explosive',
             'eyes', 'family', 'fan', 'feather', 'festival', 'film',
             'finger', 'fire', 'floodlight', 'flower', 'foot', 'fork',
             'freeway', 'fruit', 'fungus', 'game', 'garden', 'gas',
             'gate', 'gemstone', 'girl', 'gloves', 'god', 'grapes',
             'guitar', 'hammer', 'hat', 'hieroglyph', 'highway', Cont...
```

```
              'horoscope', 'horse', 'hose', 'ice', 'insect', 'jet',
              'junk', 'kaleidoscope', 'kitchen', 'knife', 'leather',
              'leg', 'library', 'liquid', 'magnet', 'man', 'map', 'maze',
              'meat', 'meteor', 'microscope', 'milk', 'milkshake',
              'mist', 'money', 'monster', 'mosquito', 'mouth', 'nail',
              'navy', 'necklace', 'needle', 'onion', 'paintbrush',
              'parts', 'parachute', 'passport', 'pebble', 'pendulum',
              'pepper', 'perfume', 'pillow', 'plane', 'planet', 'pocket',
              'potato', 'printer', 'prison', 'pyramid', 'radar',
              'rainbow', 'record', 'restaurant', 'rifle', 'ring',
              'robot', 'rock', 'rocket', 'roof', 'room', 'rope',
              'saddle', 'salt', 'sandpaper', 'sandwich', 'satellite',
              'school', 'ship', 'shoes', 'shop', 'shower', 'signature',
              'skeleton', 'slave', 'snail', 'software', 'solid', 'space',
              'spectrum', 'sphere', 'spice', 'spiral', 'spoon', 'sport',
              'square', 'staircase', 'star', 'stomach', 'sun',
              'sunglasses', 'surveyor', 'swimming', 'sword', 'table',
              'tapestry', 'teeth', 'telescope', 'television', 'tennis',
              'thermometer', 'tiger', 'toilet', 'tongue', 'torch',
              'torpedo', 'train', 'treadmill', 'triangle', 'tunnel',
              'typewriter', 'umbrella', 'vacuum', 'vampire', 'videotape',
              'vulture', 'water', 'weapon', 'web', 'wheelchair',
              'window', 'woman', 'worm']
```

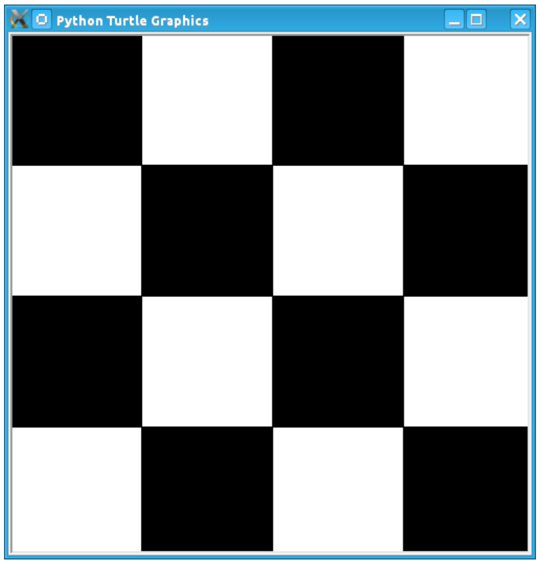
E.23 `checkerboard.py`

Write a program to input a number, and then draw - using the `turtle` module - a checkerboard with that number of squares across.

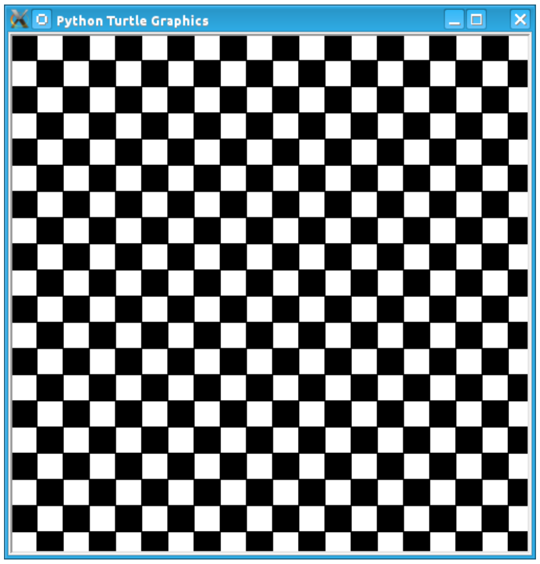
Hint: Use the `turtle` module, and to see what turtle offers you, type `dir(turtle)` in the interactive interpreter to see the available functions. You can use the `turtle.setup` function set arrange a square window for drawing.

For example:

```
Please input the number of squares across: 4
```



```
Please input the number of squares across: 20
```



E.24 `prime.py`

Write a program to input a number, and then tell the user whether the number is prime or not.

Note: A prime number is a number only divisible by 1 and itself - assume 1 is not prime, and 2 is prime.

For example:

```
Please input your number: 50
The number 50 is not prime.
                                                  Cont...
```

```
Please input your number: 29
The number 29 is prime.
```

E.25 `factors.py`

Write a program to input a number, and then print outs the factors of that number.

For example:

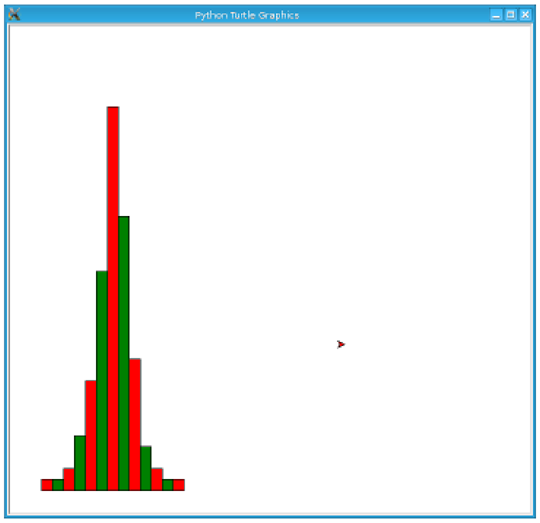
```
Please input your number: 20
The factors of 20 are: 1, 2, 4, 5, 10, 20
```

E.26 `turtlehistogram.py`

Write a program to allow the user to input a list of numbers, separated by spaces. Then, using turtle, draw a vertical histogram, with alternating colors, representing those numbers.

For example:

```
Enter the numbers for the histogram: 1 1 2 5 10 20 35 25 12 4 2 1 1
```



E.27 `kiosk.py`

Write a program that lists a set of products and their prices and allows the user to pick a number. The program should then print out the price of the item chosen.

For example:

```
1 Coke        50p
2 Fanta       45p
3 Pepsi       55p
4 Sprite      40p
5 Dr Pepper   60p
Your choice: 3
That's 55 pence please!
```

Hint: You can store your items in a list of items, with the name in the first field and the price in the second field of each item:

```
[["1 Coke", 50], ["2 Fanta", 40], ["3 Pepsi", 55],
 ["4 Sprite", 40], ["Dr Pepper", 60]]
```

E.28 order.py

Expand on the previous program to allow the user to enter a number of choices, the money they are entering, and finally the change they should get.

For example:

```
1 Coke        50p
2 Fanta       45p
3 Pepsi       55p
4 Sprite      40p
5 Dr Pepper   60p
Your choice: 2 3
Your total is 100 pence
Please enter your money: 500
Your change is 400 pence
```

Again, use the `numbers.split` command to split the original string input into a list of entries. The list you used in the previous practice can be used again for this program.

E.29 brackets.py

Write a program that reads in a sentence, and prints out whether the number of brackets match or not. You will need to account for brackets out of order as well, for example `")("`, instead of `"()"`.

For example:

```
Please input your sentence to perform a bracket match: (5 + 10) / (8 / 2
That sentence does not have matching brackets!
```

Cont...

```
Please input your sentence: 5 + 2) / (5
That sentence does not have matching brackets!

Please input your sentence: (1 + 2) * (3 + 4)
That sentence does have matching brackets!

Please input your sentence: ((10 * 2) + ((8 / 4) - 1))
That sentence does have matching brackets!
```

E.30 `calc.py`

Write a program to give the user sums to perform until he types the word "quit". Randomly chose two numbers between 1 and 10, and one operator of either addition, subtraction, division or multiplication. Use `random.randrange` function to choose the number, and also to select between your operator (1 for addition, 2 for subtraction, and so on).

For example:

```
What is 10 + 5? 15
Correct, it is 15!

What is 8 * 2? 15
Wrong, it is 16!

What is 8 - 4? 4
Correct, it is 4!

What is 10 / 2? quit
Goodbye!
```

E.31 `compound.py`

Write a program to ask the user the amount of money to invest, the interest rate per year and the number of years, and then print out total per year and the total interest earned over the investment period.

For example:

```
Please input the amount: 1000
And what is the loan period in years: 10
And finally, what is the interest rate per year in percent: 5

Total now: £1050.00 after 1 years
Total now: £1102.50 after 2 years
```

Cont...

```
Total now: £1157.62 after 3 years
Total now: £1215.51 after 4 years
Total now: £1276.28 after 5 years
Total now: £1340.10 after 6 years
Total now: £1407.10 after 7 years
Total now: £1477.46 after 8 years
Total now: £1551.33 after 9 years
Total now: £1628.89 after 10 years
Interest earned: £628.89 over 10 years
```

E.32 `temprange.py`

Write a program to ask for the start temperature, the end temperature, the type (whether Celsius or Fahrenheit), and then print out the conversion from one to the other at every degree from the start to the end.

Remember to convert from Celsius to Fahrenheit, you need to use this formula:

```
fahrenheit = (9 / 5.0) * celsius + 32
```

and to go from Fahrenheit to Celsius you use this formula:

```
celsius = (5.0 / 9) * (fahrenheit - 32)
```

For example:

```
What is the start temperature: 20
And what is the end temperature: 50
and finally, what is your reading in, celsius or fahrenheit? celsius

Celsius 20 in Fahrenheit is 68
Celsius 21 in Fahrenheit is 69
Celsius 22 in Fahrenheit is 71
Celsius 23 in Fahrenheit is 73
Celsius 24 in Fahrenheit is 75
Celsius 25 in Fahrenheit is 77
Celsius 26 in Fahrenheit is 78
Celsius 27 in Fahrenheit is 80
Celsius 28 in Fahrenheit is 82
Celsius 29 in Fahrenheit is 84
Celsius 30 in Fahrenheit is 86
Celsius 31 in Fahrenheit is 87
Celsius 32 in Fahrenheit is 89
Celsius 33 in Fahrenheit is 91
Celsius 34 in Fahrenheit is 93
Celsius 35 in Fahrenheit is 95
Celsius 36 in Fahrenheit is 96
```

Cont...

```
Celsius 37 in Fahrenheit is 98
Celsius 38 in Fahrenheit is 100
Celsius 39 in Fahrenheit is 102
Celsius 40 in Fahrenheit is 104
Celsius 41 in Fahrenheit is 105
Celsius 42 in Fahrenheit is 107
Celsius 43 in Fahrenheit is 109
Celsius 44 in Fahrenheit is 111
Celsius 45 in Fahrenheit is 113
Celsius 46 in Fahrenheit is 114
Celsius 47 in Fahrenheit is 116
Celsius 48 in Fahrenheit is 118
Celsius 49 in Fahrenheit is 120
Celsius 50 in Fahrenheit is 122
```

E.33 `cipher.py`

Write a program to read in a sentence, and then print it out with each letter shifted back by three (a Caesar cipher). So, "a" will become "x", "b" will become "y", "c" will become "z", "d" becomes "a", and so on until "z" becomes "w". You can ignore all letters apart from lowercase ones ("a" to "z") and print them out unchanged (e.g. spaces).

Having printed out the encoded message, the program should then decode it so that each letter is shifted forwards by three, and then print out the result. Obviously, the decoded message should be the same as the one originally input by the user in the first place.

Hint: Use the `ord` function to get the numerical representation of a letter, and `chr` to convert them back to their character representation. Use `string.ascii_lowercase` as a shortcut for the lowercase alphabet, although you will have to import the `string` module first.

For example:

```
Please input your sentence to encode: mary had a little lamb
The encoded sentence is: jxov exa x ifqqib ixjy
The decoded sentence is: mary had a little lamb
```

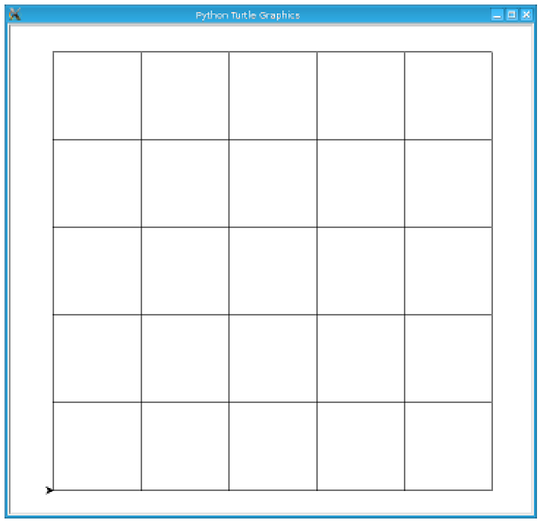
E.34 `turtlebattleship.py`

Write a program that, until the user hits the ship, does:

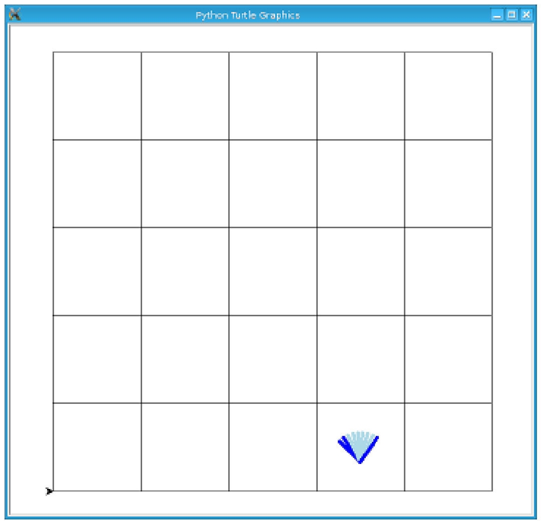
Draws a checkerboard board using turtle. The unbombarded squares should be left white, misses colored blue (or some other symbol) and hit squares colored red (or some other symbol). The user should then be asked for an x, y coordinate for his shot.

At the end, the board should be drawn one last time, then the program should wait for the user to quit.

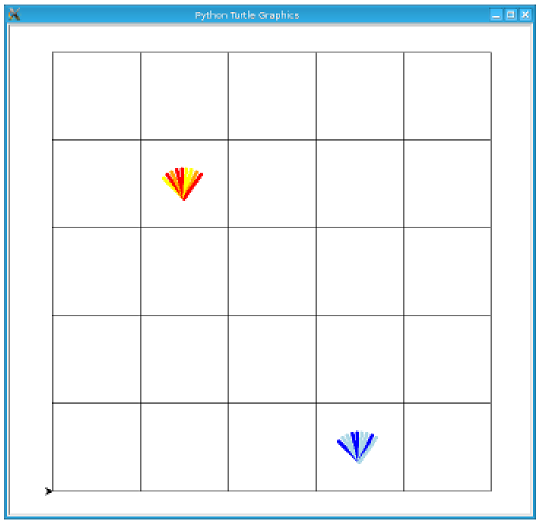
For example:



```
What is your move in the form 'x y'? 3 4
```



```
What is your move in the form 'x y'? 1 1
```



```
You sunk me, press enter to quit...
```

APPENDIX

F

THE ZEN OF PYTHON

Nuff said!

— Stan Lee

F.0 Abstract

Long time Pythoneer Tim Peters succinctly channels the BDFL's guiding principles for Python's design into 20 aphorisms, only 19 of which have been written down.

F.1 The Zen of Python

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

Special cases aren't special enough to break the rules.

Although practicality beats purity.

Errors should never pass silently.

Unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.

There should be one — and preferably only one — obvious way to do it.

Although that way may not be obvious at first unless you're Dutch.

Now is better than never.

Although never is often better than *right* now.

If the implementation is hard to explain, it's a bad idea.

If the implementation is easy to explain, it may be a good idea.

Namespaces are one honking great idea — let's do more of those!

F.2 Easter Egg

Type into the Python interpreter:

```python
import this
```

APPENDIX

G

DEVELOPMENT

G.0 TODO

G.1 Test Area

Programming today is a race between software engineers striving to build bigger and better idiot-proof programs, and the Universe trying to produce bigger and better idiots. So far, the Universe is winning.

— Rick Cook

Inline code: `[str(i) + "#" for i in range(0, len(dir(__builtins__))) if i < 24.6]`

Attention: attention

Caution: caution

Danger: danger

Error: error

Hint: hint

Important: important

Note: note

Tip: tip

Warning: warning

More code:

```
print('one')
print('two')

x = 3
to_b = True

if x == 1:
    print('one')

cond1 = x and x or x and not x
cond2 = to_b or not to_b
if cond1 and cond2:
    # do something or just
    pass
```

```
this will break
```

```
this_will_not_break = 1
```

```
this will also break!
```

Longness:

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx = 11
```

Random stuff about *argument* or *arguments* or see `/usr/lib/python2.{x}/site-packages` …

Inline test:

- `!  "#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ`
- `[\]^_‘abcdefghijklmnopqrstuvwxyz{|}~`
- `x` or `|`

1. one
2. two
3. three
4. four
5. five