An Introduction To

python

by Mark Clarkson

Version 2.3, June 2014

Thanks

Many thanks go to Peter Thompson for his invaluable help, particularly with the sections on lists and dictionaries, and also to Brian Lockwood for proof reading and suggestions for improvements.

Copyright Notice

This book is released under a Creative Commons BY-NC-SA 3.0 licence

(http://creativecommons.org/licenses/by-nc-sa/3.0/)

You are free:

To share:

You may copy, distribute & transmit the work. You may provide digital or printed copies for students.

To remix:

You may adapt or improve the work.

Under the following conditions:

Attribution:

You must attribute the work to Mark Clarkson by including my name on the front cover and in the footer.

Noncommercial:

You may not use this work for commercial purposes. If you are providing printed copies you may only charge a fee to cover production costs, not profit. You must not charge for a digital version of the work.

Share alike:

If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar licence to this one. Please let me know if you make significant changes so that I can include them in my own copies.

Version History

v2.3 Added colour coding to all code examples

Added a section in chapter 1 on errors

v2.2 Tidied up a few typographical errors

v2.1 Added sections on turtle programming & more list activities

Reworded most code to give meaningful names to variables

v2.0 Reworded Arrays section to lists

Added sections on dictionaries & regular expressions

Moved WHILE loops ahead of FOR loops

An Introduction to Python

Table Of Contents

Section 1 - Getting Started

- 1.1 - Meeting Python
- 1.2 - Writing Programs
- 1.3 - Handling Errors
- 1.4 - Arithmetic
- 1.5 - Comments

Section 2 - Data Types

- 2.1 - Variables
- 2.2 - Strings
- 2.3 - Numbers
- 2.4 - Boolean
- 2.5 - Lists

Section 3 - Control Flow

- 3.1 - Selection
- 3.2 - Iteration
- 3.3 - Procedures

Section 4 - Graphical Tools

- 4.1 - Turtle

Section 5 - Extra techniques

- 5.1 - Regular Expressions
- 5.2 - File Handling
- 5.3 - Dictionaries

Section 6 - Additional Exercises

1.1 Meeting Python

1.1a - Downloading Python

Python is a freely available programming language. You will need to install the Python compiler in order to make your own programs, although you don't need any special software to WRITE Python programs.

Go to the python.org website and download a standard installation of the latest version of Python (currently 3.2). This will provide you with everything you need.

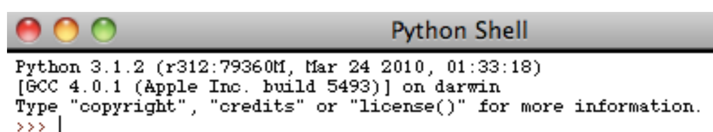
NB: Do NOT download Python 2.x as some of the programs in this course will NOT work! It must be version 3 or higher!!!

The documentation for Python is very good, and you can find it at http://docs.python.org/py3k/

1.1b - IDLE

You can write Python code in notepad, or any other simple text editors - but it is easier if you use a program like IDLE (which comes with the standard Python installation) because it helps with things like indentation and debugging (things we will cover later). We will be using IDLE throughout most of the course.

When you run IDLE you will see a window like this:



NB: The (few) screenshots in this booklet are taken from Mac OS X. Windows users may see something slightly different, but largely similar.

You can type simple commands at the prompt (>>>).

Try typing:

2+2

17-9

16/4

3*7

Check that the answers are correct!

1.2 Writing Programs

Quick note: you are going to be making lots of Python programs. Create a Python folder somewhere so that you can save your files here!

1.2a - Hello World

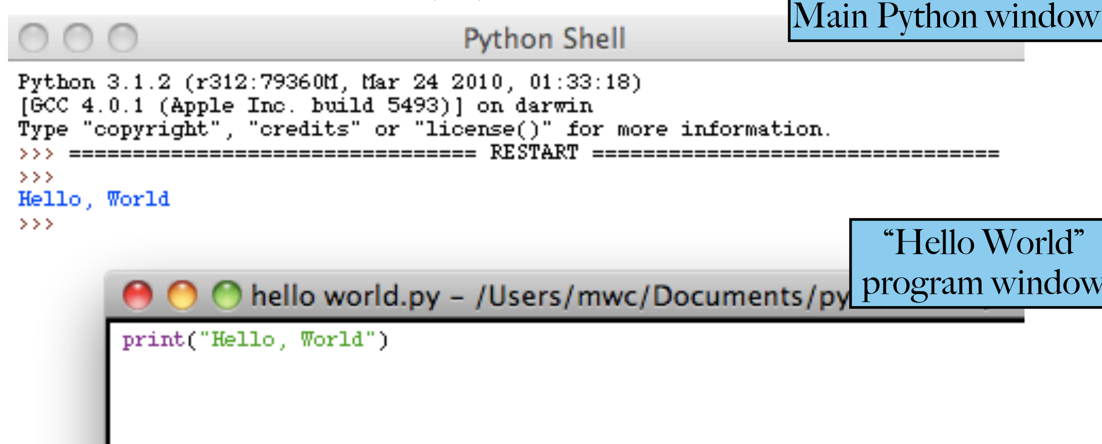
We can put simple instructions into the IDLE window, but we can't write and save complex programs. Open IDLE and click 'file >> new' - this will give you a new window that you can use to write and save your programs!

In this new window, type the following (exactly):

```
print("Hello, World")
```

Save this file in your Python folder (call it "hello world.py") and then press F5 to run it.

The main IDLE window should show you your output:



Congratulations! You have written your first Python program!

You can run your program on any computer with Python installed, no matter what kind of computer it is.

1.2b - Hello Dave

Lets try something a little more... interactive now.

Create a new file and type in the following:

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

What do you think will happen?

Save this file as "hello dave.py" and press F5 to run it. See if you were correct.

1.2c- A note on colours

Colour coding in Python is really helpful.

IDLE uses the following colours. If you're using a different system then your colours might be different.

Colour	Meaning	Examples	Notes
Purple	A function - telling the program to do something	print( ), input( )	Functions always have brackets attached
Green	Text	"This is some text"	Often called a "string literal" because it literally repeats whatever it says
Orange	A key word	if, while, True, import	If statements and loops use orange text.
Black	Everything else	name, score = score + 1	Anything in black is usually either a variable or some calculation
Red	Comments	# This is a comment	Python ignores comments, they're only for the programmer

Try to keep an eye out for colour coding - it will help you spot syntax errors when you make them.

1.3 Errors

Have you ever played Angry Birds? The game where you use a catapult to launch birds and try to knock over some pigs. The idea is that you try to get them all in the first go, but if you get it wrong you get to try again - as many times as you like.

Programming is a lot like Angry Birds. You'll often come close, but not quite manage it on the first try. The trick is not to give up, but to keep trying until you get it right.

There are two types of error you will make when you are programming:

1.3a - Syntax Errors

A syntax error means that you've typed something wrong. The most common syntax errors are:

- Forgetting to close a bracket:

```
name = input("What is your name? "
```

- Forgetting to close a speech mark:

```
name = input("What is your name? )
```

- Capital letter issues

```
Print("What is your name? )
```

- Forgetting a colon (:) with an if statement or a loop (we'll look at those later)

IDLE will try and help you spot those errors by complaining at you and sometimes by highlighting a piece of code. In the third example above, the `print()` function should be purple. The capital letter means it's not quite right, so spotting the colout codes can be helpful.

1.3b - Logic Errors

A logic error is when the program runs, it just doesn't do what you want.

Can you spot the logic error here?

```
if shot:
    lives = lives + 1
```

Usually in a game you lose a life when you get shot, but here it says to add one.

Logic errors can be a bit trickier to spot, so always make sure you test programs carefully!

1.4 Arithmetic

1.4a - Addition

Create a new file and type in the following code:

```
firstNumber = int(input("Enter a number between 1 and 100: "))
secondNumber = int(input("Enter another number between 1 and 100: "))
print("firstNumber + secondNumber =", firstNumber+secondNumber)
```

Save this file as "addition.py" and press F5 to run it.

1.4b - Subtraction

Repeat the last program, but this time make it subtract the two numbers. Save it as "subtraction.py" and test it works.

1.4c - Multiplication

Repeat the last program, but this time make it multiply the two numbers. Save it as "multiplication.py" and test it works.

1.4d - Division

Repeat the last program, but this time make it divide the two numbers. Save it as "division.py" and test it works.

1.4e - Square

Repeat the last program, but this time make it calculate x^2. Save it as "square.py" and test it works.

1.4f - Powers

Repeat the last program, but this time make it calculate x^y. Save it as "powers.py" and test it works. (Hint: use x**y)

1.4g - Mod

Repeat the last program, but this time make it calculate the modulus (remainder) of a division. Save it as "mod.py" and test it works. (Hint: use x%y)

1.4h - Order of Operations / BIDMAS

Try writing a program that will take a number, multiply by three and then add four.

Try writing a program that will take a number, add four and then multiply by three.

Put the number 7 into both programs and check that they work correctly.

1.5 Comments

1.5a - Simple Comments

Comments are bits of text in the program code that aren't used by the computer, but help to explain what is going on.

You can write a comment using a # symbol (Alt-3 on a Mac).

Try this program:

```
#This is a comment
print("This is not a comment.")
```

1.5b - Inline Comments

You can include a comment at the end of a line of code. But not the start:

```
#This is a comment
#This is also a comment. It won't print(“anything”)
print("This is not a comment.") #although this is...
```

1.5c - Block Comments

Sometimes you might want to have a large block for a comment. This is often the case at the start of programs where programmers put useful information such as who wrote it, when and what it is supposed to do.

Try to guess what the output will look like:

```
#This is a comment
#This is also a comment. It won't print(anything)
print("This is not a comment.") #although this is...
"""
These
are
all
comments
"""
print("But the comments are finished now.")
```

Comments make it MUCH easier to see what you have done when you come back to your code several months later and you should use them regularly.

2.1 Variables

2.1a - Creating Variables

A variable is a space where you can store something - a number, a word... anything you want the program to remember. Each variable has three parts - a NAME, a DATA TYPE and a VALUE.

NAME: It is REALLY important to choose a good name that shows what the variable is for. So in a computer game you might use numberOfLives, score, bulletsRemaining and health. You probably wouldn't want var1, var2, var3 and var4.

TYPE: Python tries to hide the data type to make life easier for you, but read the rest of this chapter to see what the data types are - it really is important.

VALUE: This is what you have saved into the variable at the minute - the whole point of using them!

When we create a variable we call it 'declaring' a variable. When we give the variable its first value we call it 'initialising' a variable.

If we want to tell the computer to 'Create a variable called "age" and give it the number 12', we say:

```
age = 12
```

2.1b - Assignment

One of the most frequent things you will do in programming is to store values.

The following code will work out an answer and print it to the screen:

```
print(3+4)
```

But this code will work out an answer and store it in a variable called 'answer':

```
answer = 3 + 4
```

Assignment always works to the left, so work out the answer on the right and save it last.

You can also perform calculations that use a variable

```
numberOne = 3
numberTwo = 4
answer = numberOne + numberTwo
```

You can use a variable in a calculation, and use the same variable to store the answer. Try the following code:

```
score = 112
score = score + 1
print(score)
```

2.2 Strings

2.2a - Adding Strings

A string is a block of text. It could be a single word, a sentence or an entire paragraph. In Python we use the term “str” to mean a string.

Try out the following code:

```
start = "Hello, "
name = input("What is your name? ")
end = ". How are you today?"

sentence = start + name + end

print(sentence)
```

If you add strings together, you end up with one long string.

2.2b - When is a number not a number?

Try this one:

```
number = str(2)

print("2 + 2 = ", number + number)
```

This makes the value “2” into a string - so it is a word rather than a number.

2.2c - Line Breaks / New Lines

You can also write your string over several lines by using “\n”. Like this:

```
firstLine = "I got a feeling,\nThat tonight's gonna be a good night.\n"
secondLine = "That tonight's gonna be a good, good night."

print(firstLine + secondLine)
```

Try writing a few of your own!

2.2d - Combining strings and numbers

You can use a comma to combine different bits of text or numbers. You will use this a lot:

```
start = "2 + 2 ="
number = 4
#NB: Use a comma instead of a + if you have two different types of data
print(start, number)
print("Two plus two = ",number)
```

2.3 Numbers

2.3a - Integers

There are two main types of numbers in Python. We'll start with Integers (whole numbers).

We can force Python to make something a whole number by saying `int(number)`. Like this:

```
number = 3.7
newNumber = int(number)
print(newNumber)
```

Note that this won't round the number up, it just takes the whole number part.

2.3b - Floats

The second main type is a floating point, or just "float" for short (a decimal). We can force Python to make something a float by saying `float(number)`. Like this:

```
number = 3
print(number)
newNumber = float(number)
print(newNumber)
```

2.3c - Casting

Casting is the technical term for changing the type of data you are working with. You've already been casting by saying `int()`, `float()` and `str()` to change the data type. But now you know what it's called!

2.3d - Automatic Casting

Python will automatically change the data type for you a lot of the time:

```
x = int(22)
y = int(7)
z = x/y
print(z)
```

Here, the two integers have provided a floating point answer and Python has changed the data type of `z` for you.

2.3e - Long

Another number type you might need occasionally is `long`, or a long integer. Integers in Python can only hold numbers from -2billion to +2billion (actually from -2,147,483,646 to +2,147,483,647). If your number is going to be outside that range then you will have to use a long to keep count.

2.4 Boolean

2.4a - True or False?

A boolean (`bool` for short) can store one of only two possible values - True or False. (NB: Note the capital letters!)

This is useful for logical tests (e.g. is 22/7 bigger than 3?) or to check if something has been completed (this becomes useful in section 3).

Here is an example:

```
bigger = False
print("22/7 > 3")

if (22/7 > 3):
    bigger = True
print("bigger = ", bigger)
```

The main advantage of booleans is that they are very small and simple.

2.5 Lists

2.5a - What's a list?

NB: In some languages you would do this stuff with ARRAYS, which work in a very similar way

A list is a collection of something, like integers:

```
numbers = [3,5,9,6]
print(numbers)
print(numbers[0])
print(numbers[1])
print(numbers[2])
print(numbers[3])
```

You use square brackets to write the list and then square brackets again to say which LIST ELEMENT you want. We start counting from 0 so the first number in our list is numbers[0] and the last one is numbers[3].

2.5b - Editing Lists

As well as reading individual elements we can write to individual elements too:

```
numbers = [3,5,9,6]
print(numbers)
numbers[1] = 2
print(numbers)
```

2.5c - Lists of items

Lists are really good when you use two or more together. Lets say we want to store a load of people's ages:

```
names = ["Paul","Phillip","Paula","Phillipa"]
ages = [12,15,11,14]

print(names[0], "is" ,ages[0])
print(names[1], "is" ,ages[1])
print(names[2], "is" ,ages[2])
print(names[3], "is" ,ages[3])
```

2.5d - Strings (Redux)

Strings are really arrays of characters.

As such you can do the following:

```
name = "alex"
print(name[2])
```

Although you can't edit each list element as you can with ordinary lists.

2.5e - Some useful things you can do with lists

Sometimes you will want to create an empty list, and fill it with data first.

```
names = [""]*5
print(names)
```

If you find your list is too short, you can add one more value at a time by using the APPEND procedure.

```
names = ["Rita", "Sue"]
names.append("Bob"]
print(names)
```

If you want to add more than one value at a time you can use EXTEND procedure.

```
names = ["Graham","Eric","Terry G."]
extraNames = ["Terry J.", "John", "Michael"]
names.extend(extraNames)
print(names)
```

Try the code above with APPEND instead of EXTEND and see what happens.

You can search a list using IN:

```
names = ["Graham","Eric","Terry"]
if "John" in names:
      print("John present")
if "Eric" in names:
      print("Eric present")
```

Try the code above

If you want to know where in the list it is, you can use INDEX:

```
names = ["Graham","Eric","Terry"]
position = names.index("Eric")
print(position)
```

BUT be careful because if the thing you're searching for isn't there, you'll get a nasty error. Perhaps try:

```
names = ["Graham","Eric","Terry"]
if "Eric" in names:
      position = names.index("Eric")
      print("Eric in position",position)
else:
      print("Eric not found")
```

3.1 Selection

Selection means selecting (or choosing) what to do next. Should I cycle to school, or ask for a lift? If it's a sunny day I might cycle. If it's raining, I'll ask for a lift.

3.1a - IF ... ELSE

Create a new file and type in the following code:

```
answer = int(input("How many hours a day do you play computer games? "))
if answer < 2:
      print("That seems a fairly healthy balance. Well done!")
else:
      print("You're probably good enough by now with all that practice.")
```

Save this file as "games.py", press F5 and test it.

Notice how the colon (:) is used to say what should happen in each case.

Also notice that the indentation is VERY important. Python only knows when your IF statement is finished by looking at the indentation!

Try this program to see what I mean:

```
answer = int(input("How many hours a day do you play computer games? ")
if answer < 2:
      print("That seems a fairly healthy balance. Well done!")
else:
      print("You're probably good enough by now with all that practice.")
print("Xbox 360s are better than PS3s")
```

That last line will be printed no matter which option you chose.

3.1b - IF ... ELIF ... ELSE

Sometimes there are more than two options. I could walk OR cycle OR get the bus OR get a lift. As well as IF and ELSE, we can stick an 'ELSE IF' (or ELIF) in the middle:

```
answer = int(input("How many hours a day do you play computer games? ")
if answer < 2:
      print("That seems a fairly healthy balance. Well done!")
elif answer < 4:
      print("You're probably good enough by now with all that practice.")
else:
      print("Put the controller down and get some fresh air once in a while!")
```

3.1c - IF ... ELIF ... ELIF ... ELIF ... ELSE

You can include an unlimited number of ELIFs if you need to. Try the following:

```
menu = "What would you like:\n\
    1. A complement?\n\
    2. An insult?\n\
    3. A proverb?\n\
    4. An idiom?\n\
    9. Quit\n"

answer = int(input(menu))

if answer == 1:
    print("You look lovely today!")
elif answer == 2:
    print("You smell funny.")
elif answer == 3:
    print("Two wrongs don't make a right. But three lefts do...")
elif answer == 4:
    print("The pen is mightier than the sword.")
elif answer == 9:
    print("Goodbye!!!")
```

There are a couple of important bits here:

- You can put a line break in your string by using "\n".
- You can continue a line of code by putting a "\" at the end.
- If you are testing for equality, use a double equals (is 3x2 = = 6?)

Try making your own menu system.

3.1d - Inequalities

This seems a good point to mention some of the most important inequalities used in selection:

<	Less Than	e.g. 3 < 5	
>	Greater Than	e.g. 5 > 3	
<=	Less Than Or Equal To	e.g. 3 <= 5	5 <= 5
>=	Greater Than Or Equal To	e.g. 5 >= 3	5 >= 5
!=	Not Equal To	e.g. 5 != 3	3 != 5

3.1e - Selection Challenge

Try making a calculator.

The program will ask for two numbers and then give you a menu, so you can add, subtract, multiply, divide, square either number or find out the power.

3.2 Iteration

Iteration is a posh way of saying "loop" (iteration literally means to do something again).

Loops are absolutely vital in programming in order to avoid having to write sequences of code out again and again. And they come in several forms:

3.2a - While Loops

A WHILE loop is a way of repeating a section of code. It is easy to understand because it uses something that looks a lot like an IF statement at the top.

We use the TAB key to INDENT the code to show which bit of code we repeat.

How many times do you think this loop will repeat? Try it and find out:

```
number = 1
while number < 10:
      print("This is turn", number)
      number = number + 1
print("The loop is now finished")
```

Make sure you understand why the last line only prints out once, at the end of the loop.

A WHILE loop is ideal if you don't know exactly how many times you will need to repeat the code.

This example will keep asking for a number until it gets the right one:

```
targetNumber = 7                                # this is the correct number
guess = int(input("Guess a number between 1 and 10"))# ask for a guess
while guess != targetNumber:                    # repeat if the answer is wrong
      print("Wrong, try again")
      guess = input("Guess a number between 1 and 10")
print("Congratulations - that's right!")        # do this after the loop
```

A motorbike costs £2000 and loses 10% of its value each year. Print the bike's value each year until it falls below £1000

```
value = 2000

while value > 1000:
    print("£",value)
    value = value * 0.9
```

3.2b - FOR Loops

A FOR loop is ideal if we now how many times we want to repeat. Try this code:

```
for loopCounter in range(10):
    print(loopCounter)
```

There are two things to note:

Computers start counting at 0. This is actually better than starting at 1, but isn't always obvious straight away.

You have to finish the loop one number HIGHER than the last number you actually want

- so it has repeated 10 times, but goes from 0 to 9.

You can be a bit more specific with your loop by setting a start and an end number:

```
for loopCounter in range(5,10):
    print(loopCounter)
```

Remember that loops are closed (or finished) by removing the indentation:

```
for loopCounter in range(1,4):
    print(loopCounter,"potato")
print("4")
for loopCounter in range(5,8):
    print(loopCounter,"potato")
print("more")
```

You can also specify how the counter will count:

```
for loopCounter in range(2,9,2):
    print(loopCounter)
print("Who do we appreciate?")
```

Or the three times table:

```
for loopCounter in range(3,37,3):
    print(loopCounter)
```

3.2c - FOR Loop Challenge

1. Look closely at the three times table example above. Write a similar program to calculate the four times table.

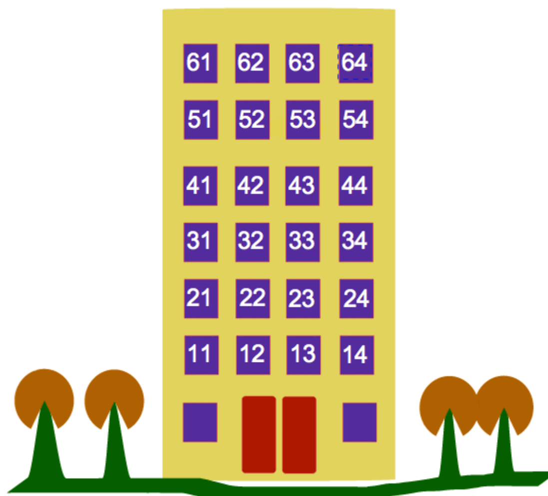
2.Write a program that will calculate the 5 times table.

3.Try writing a program that will prompt for an integer (whole number) and print the correct times table (up to 12x). Test with all of the tables up to 12.

3.2d - Nested Loops

Sometimes you need to put one loop inside another loop. The logic can get pretty complicated, so you have to be careful - but it does work!

A window cleaner has to clean all the windows in a hotel.



First, he (or she) cleans all the windows on the first floor:

```
for room in range(1,5):
      print("Room",room,"cleaned")
```

Then he (or she) does the 2nd floor and the 3rd, and so on:

```
floor = 1
for room in range(1,5):
      print("Room",floor,room,"cleaned")
floor = 2
for room in range(1,5):
      print("Room",floor,room,"cleaned")
floor = 3
for room in range(1,5):
      print("Room",floor,room,"cleaned")
```

A better way is to say

```
for floor in range (1,7):
      for room in range(1,5):
            print("Room",floor,room,"cleaned")
```

The OUTSIDE loop says we are dealing with floor one. The INSIDE loop deals with each room.

Then the OUTSIDE loop goes up to floor 2, and the INSIDE loop repeats again for all four rooms.

The whole process repeats until floor 6 has been finished.

Think about what this program will do. Decide what you think will happen. Then try it.

```
for x in range(1,11):
    for y in range(1,11):
        print(x,"x",y,"=",x*y)
```

Were you correct? Was it easy to follow the logic? How many lines of code would you need to write it with only 1 loop at a time?

3.2e - Break Out

Sometimes you need to stop a loop before coming to the end of the original instructions. To do this you can use a BREAK:

```
for number in range(10000000000):
    print(number)
    if number == 20:
        break
```

Write a program that will start at 1 and end at 1,000,000. The program should print each number followed by its square but break once this square is bigger than 4096.

3.2f - Loop Challenge

A factorial means the number and all of the preceding numbers multiplied together.

So 4 factorial (written 4!) = 1 x 2 x 3 x 4

5! = 1 x 2 x 3 x 4 x 5

And so on...

Write a program that will prompt for an integer and then calculate the factorial of that number.

To test it works, 8! > 40,320

3.3 Procedures

If the same piece of code needs to be used several times we can use a loop - but only if those times are all together. If you need to run the same bit of code again later, you can use a procedure.

3.3a - A Simple Procedure

In this example I want to print the squares of the numbers from 1 to 20:

```
#This is a procedure called output
#It will print a given number and its square
def output(number):
    print(number,"squared =",number*number)

#This is the start of the main program
number = 1

while number < 21:
    output(number)
    number = number + 1
```

Here the top two lines are a procedure called `output`. The procedure expects to be given one variable (which it calls `number`) and will happily do its thing when told to.

The main program is underneath, and the line that says `output(number)` will run the output procedure and give it the variable `number` to work with.

You don't have to use the same variable name in the procedure that you do in the main program, although we have here to make it easy to read.

3.3b - Multiple Parameters

A parameter is the variable (or variables) that get passed to the procedure. You can require more than one parameter:

```
def output(number1,number2):
    print(number1,"+",number2,"=",number1+number2)

for counter in range(20):
    number = counter**2
    output(counter,number)
```

You can pass as many parameters as you like to a procedure as long as you set it up correctly.

You might have noticed that this time we have used different names between the main program and the procedure.

3.3c - Optional Parameters

Once you have defined a procedure, you can call it in a number of ways:

```
def output(grade,score=50,feedback="Well done!"):
    print("You scored",score,"which is a grade",grade,feedback)

output("C")
output("A",87)
output("E",12,"Rubbish!")
```

You can pass as many (or few) of the parameters as you like - as long as they're in the right order. This means you have to plan the procedure quite carefully.

3.3d - Functions

So far our procedures have just printed an answer to the console. Sometimes a procedure will have to send some data back. This is called a `return`. Procedures that return something are called functions. In Python that isn't a huge distinction, but in some programming languages it is very important.

```
def calculate(number):
    newnumber = number / 100
    return (newnumber)

for counter in range(5,101,5):
    answer = calculate(counter)
    print(counter,"% = ",answer)
```

The line `print(newnumber)` passes the value of the newnumber variable back to the main program.

The line `y = caluclate(x)` shows that the main program expects a value to be returned and has somewhere to store it.

3.3e - Modules

There are some procedures already built in to Python (print, for example). Modules (sometimes known as libraries in other languages) are collections of extra procedures and functions that are pre-written.

Try this program first. It won't work, but I'm trying to prove a point (sqrt is short for square root):

```
number = 49
answer = sqrt(49)
print(answer)
```

Now try this one:

```
import math

number = 49
answer = math.sqrt(49)
print(answer)
```

You can find a list of modules at http://docs.python.org/py3k/modindex.html

4.1 Turtle

4.1a - Creating a Turtle

If you've ever used Logo, Roamer or BeeBots then you'll be familiar with the concept of a Turtle program. If you're not, then try this code:

```
import turtle                 #import the turtle module
window = turtle.Screen()      #create a window
timmy = turtle.Turtle()       #create a turtle called timmy
timmy.forward(150)
timmy.right(90)
timmy.forward(100)
window.exitonclick()          #close the window when clicked <-- VERY important
```

The commands forward, right and left are pretty straightforward.

Keeping the first 5 lines the same, and keeping the last line last, add some more code to draw a square.

4.1b - Creating shapes

You probably came used several forward and right commands to make the square, but there is an easier way:

```
import turtle                 #import the methods to do with turtle programs
window = turtle.Screen()      #create a window
timmy = turtle.Turtle()       #create a turtle called timmy
for loopCounter in range(4):  #repeat the next bit 4 times
      timmy.forward(150)
      timmy.right(90)
window.exitonclick()
```

Remember that we use a FOR loop if we know how many times we want to repeat something. Check it works!

A square has 4 sides, of length 150, and with 90° turns. A hexagon has 6 sides, of length 150, and with 60° turns. Try to draw a hexagon.

4.1c - Creating shapes challenge

A circle has 360°. For each shape the turtle has to do one full circuit (i.e. turn 360°).

A square has 4 turns, and 360° ÷ 4 = 90°.

A hexagon has 6 turns, and 360° ÷ 4 = 60°.

A triangle has 3 turns, and 360°÷ 3 = ???

Try drawing a triangle. And a pentagon (5 sides). And an octagon (8 sides). And a decagon (10 sides).

4.1d - Procedures

It would be really helpful if, instead of writing:

```
for loopCounter in range(4):  #repeat the next bit 4 times
      timmy.forward(150)
      timmy.right(90)
```

We could simply say:

```
drawASquare()
```

Try this program:

```
import turtle

def drawASquare(whichTurtle):
      for loopCounter in range(4):
            whichTurtle.forward(150)
            whichTurtle.right(90)

window = turtle.Screen()
timmy = turtle.Turtle()
drawASquare(timmy)
window.exitonclick()
```

It just draws a square, right? OK, now in the main program replace :

```
drawASquare(timmy)
```

with :

```
for loopCounter in range(72):       #repeat 72 times
      drawASquare(timmy)
      timmy.left(5)                 #turn 5⁰. Note that 360⁰ ÷ 5 = 72⁰
```

If that took a bit too long, try inserting this line straight after you have created Timmy :

```
timmy.speed(0)
```

4.1e - Spirograph

There used to be a toy called a Spirograph that let you draw lots of coloured patterns like this.

To make the pictures more interesting, we can try changing the colour:

```
timmy.color('blue')
```

On its own, that might not look too amazing, but try this:

```
colourCounter = 1
for loopCounter in range(72):
      drawASquare(timmy)
      timmy.left(5)
      if colourCounter == 1:
            timmy.color('blue')
      elif colourCounter == 2:
            timmy.color('red')
      elif colourCounter == 3:
            timmy.color('yellow')
      elif colourCounter == 4:
            timmy.color('green')
            colourCounter = 0
      colourCounter += 1
```

Try using pentagons instead of squares, or triangles, or octagons . Try turning amounts other than 5^0.

4.1f - Lots of turtles

It is very easy to create lots of turtles.

Make sure you have procedures for a square and a hexagon first, then try this:

```
window = turtle.Screen()
timmy = turtle.Turtle()
tina = turtle.Turtle()
timmy.color('blue')
tina.color('pink')

drawASquare(timmy)
drawAHexagon(tina)

window.exitonclick()
```

See what other patterns you can come up with using 2, 3 or even more turtles.

5.1 Regular Expressions

5.1a - Regular Expressions as an input mask

Regular expressions are a really easy way to perform validation checks on data. If you've used a database before, they're very similar to input masks.

First of all we need to import the regular expressions module:

```
import re
```

The function `re.search` works like this :

```
re.match(rule, stringToCheck)
```

For a simple example, our user has to enter the correct password ('p@ssword'):

```
import re
userInput = input("Enter your password: ")
if re.search("p@ssword", userInput):
      print("Correct password entered")
else:
      print("Error, incorrect password")
```

Rules can be a bit more detailed. If we need a postcode that looks like TS16 0LA we can say:

```
import re
#postcodeRule is "letter, letter, number, number, space, number, letter, letter"
postcodeRule = "[A-Z][A-Z][0-9][0-9] [0-9][A-Z][A-Z]"
userInput = input("Enter a postcode: ")
if re.search(postcodeRule,userInput):
      print("Valid postcode!")
else:
      print("Invalid postcode!")
```

The `[A-Z]` check looks for an upper case letter. The `[0-9]` check looks for a number. The space in the middle IS recognised, and if you don't leave a space in the postcode you type in, the check will fail.

There are several codes you can use to create your rules:

a	- the letter 'a'	+	- allow multiple instances
[a-z]	- a lower case letter	^	- only match at the start
[A-Z]	- an upper case letter	$	- only match at the end
[a-zA-Z]	- an upper or lower case letter		
[0-9] OR \d	- a number		
[a-zA-Z0-9] OR \w	- a number or letter		
.	- any character		
\.	- a full stop		

Some examples:

```
import re
string = "Now for something completely different"
if re.search("^N",string):
      print("Test 1 - Success!")     #Will pass, it starts with "N"
if re.search("for",string):
      print("Test 2 - Success!")     #Will pass, the phrase "for" will be found
if re.search("^for",string):
      print("Test 3 - Success!")     #Will fail, it doesn't start with "for"
if re.search("\w$",string):
      print("Test 4 - Success!")     #Will pass, it ends with a number or letter
if re.search("^.+t$",string):
      print("Test 5 - Success!")
#Will pass, it starts with a number of characters (^.+) and ends with a "t" (t$)
```

5.1b - Regular Expression challenge

Create a program that asks the user to type in a URL (website address) and validates it. A valid URL should have "some characters, a full stop, some characters, a full stop and some characters".

5.2 – Dictionaries

5.2a - Introduction

Dictionaries are a type of data structure that are a little like lists in that they are a sequence of elements. However, elements are not accessed using indexes as are lists, but using keys. Each element in a dictionary consists of a key/value pair separated by a colon. For example "george":"blue". The string "george" is the key and "blue" is the value. Keys can be other datatypes, for example numbers. However, the datatype of all keys in a dictionary must be the same.

Because elements are retrieved using their keys looking them up is done in just one go. This means it is a very fast operation which takes the same time no matter whether the dictionary has 1 or 1 million elements.

5.2b - An Example

```
 # Note the use of curly brackets at the start and end of the dictionary, and
colons to separate the key/value pairs.
logins  = {"john":"yellow", "paul":"red", "george":"blue", "ringo":"green"}
print(logins["george"]) => "blue"
```

And

```
# Change george's password to purple
logins["george"] = "purple"
print(logins)
```

Gives

```
{"ringo": "green", "paul": "red", "john": "yellow", "george": "purple"}
```

Note how the order of the dictionary has changed. It is not possible to stipulate the order in a dictionary, but this does not matter as sequential access it not needed because element values are accessed using keys.

5.2c - Iteration

Dictionaries can be iterated over using a for loop like so:

```
# Print all the keys
for item in logins:
      print(item)
# Print all the values
for item in logins:
      print(logins[item])
# Print both
for key, value  in logins.items():
      print("The key is:", key, "and the value is:", value)
```

Prints:

```
The key is: ringo and the value is: green
The key is: paul and the value is: red
The key is: john and the value is: yellow
The key is: george and the value is: blue
```

There are many methods and functions associated with dictionaries. Look at the official Python documentation. Also, values stored in a dictionary can be virtually any data type. For example, athletes and their list of running times as values. Note that samantha has one less time than mary.

```
training_times = {"mary":[12, 45.23, 32.21], "samantha":[ 11.5, 40.1]}
```

5.3 File Handling

BE CAREFUL!!

It is possible to do some serious damage here if you're not careful!

Before we start, an important note about directory structures:

Under Windows the default file location will be in the Python installation directory (typically C:\python3.1). On a school network you can use file locations such as "h:/temp" to refer to your home directory - note which slash is used!.

Under Mac OS X the default file location is your home directory and if you store your work in a sub-directory you could use something like "pythonFiles/temp".

To keep the examples simple the file details here are assumed to be in the default directory. But be careful, and ask for help if unsure!

5.3a - Do This First

All will become clear, but before we can do anything else we need to create a data file:

```
file = open("temp","w")
for loop in range(1,11):
    text = "This is test number " + str(loop) + "\n"
    file.write(text)
file.close()


file = open("temp","r")
print(file.read())
file.close()
```

You should have the following printout:

```
>>>
This is test number 1
This is test number 2
This is test number 3
This is test number 4
This is test number 5
This is test number 6
This is test number 7
This is test number 8
This is test number 9
This is test number 10
>>> |
```

5.3b - Opening Files

To open a file, you have to make it a variable (just as a string or an integer must be a variable). We do this with the following command:

```
file = open("filename","mode")
```

Where filename is the name of the file and the mode must be one of the following:

r	read only
w	write only - NOTE, this destroys any previous version of the file!
r+	read AND write
a	append mode (will write only after the existing data)

Once you have finished with a file you MUST get into the habit of closing it, or problems will occur. To do so simply write file`.close()`.

5.3c - Reading From Files

Once your file is open you can read from it using the functions:

`file.read()` - read the entire file

`file.readline()` - read the current line

It is important to remember that Python keeps track of where you are up to.

Try the following code:

```
file = open("temp","r")
print(file.readline())
print("That was the first line")
print(file.readline())
print("That was the second line")
print(file.read())
print("That was all the rest")
print(file.readline())
print("I bet that was blank")
file.close()
```

You can tell Python to go back to the start by using the seek() procedure:

```
file = open("temp","r")
print(file.readline())
print("That was the first line")
file.seek(0)
print(file.readline())
print("That was the first line again")
file.close()
```

You can read one character at a time (or 5 characters at a time) by passing a parameter:

```
file = open("temp","r")
print("The first letter is:")
print(file.read(1))
print("The second letter is:")
print(file.read(1))
file.close()
```

5.3d - Reading Challenge

Write a program that will read each character and append it to a string until the first space is found, then print that string.

5.3e - Writing

At some point you will want to write some data to a file. You can only write strings to file, so you will have to convert any numbers before you can write them:

```
number = 3
text = " is the magic number"

file = open("temp","w") #remember “w” means the old file will be overwritten
number = str(number)
file.write(number + text)
file.close()

file = open("temp","r")
print(file.read())
file.close()
```

5.3f - Writing Newlines

Don’t forget that you can use the “\n” character to force a new line.

```
#The first part will write the "1 potato, 2 potato..." rhyme.
file = open("temp","w")
for loop in range(1,4):
    file.write(str(loop) + " potato\n")
file.write("4\n")
for loop in range(5,8):
    file.write(str(loop) + " potato\n")
file.write("More!")
file.close()

#The second part will print it to the screen to check it looks OK.
file = open("temp","r")
print(file.read())
file.close
```

5.3g - Editing or Deleting Data

The easiest way to edit or delete data is to copy the bits you want and to rewrite the file from scratch:

```
#The first part will write the "1 potato, 2 potato..." rhyme.
file = open("temp","w")
for loop in range(1,4):
    file.write(str(n)+ " potato\n")
file.write("\n")
for loop in range(5,8):
    file.write(str(n) + " potato\n")
file.write("")
file.close()

#The second part will print it to the screen to check it looks OK.
file = open("temp","r")
print(file.read())
file.close

#This part creates two strings either side of the 4th line.
file = open("temp","r")
string1 = ""
for loop in range(0,3): #Copies the first 3 lines into string1
    string1 = string1 + file.readline()
file.readline() #This command makes Python skip the 4th (blank) line
string2 = ""
for loop in range(0,3): #Copies the next 3 lines into string2
    string2 = string2 + file.readline()
file.close()

#Next we rewrite all this data to a new file (with the same name).
file = open("temp","w")
file.write(string1)
file.write("4\n") #writes in the missing line in the middle
file.write(string2)
file.write("More!") #writes in the missing line at the end
file.close()

#Finally we print the whole file again to check it.
file = open("temp","r")
print(file.read())
file.close
```

6.1 Straightforward Exercises

Here are some fairly straightforward exercises. They're not completely easy, but should be quite manageable.

All of these programs should include the following comment block at the start (preferably completed) as well as appropriate comments within the code to explain what is going on:

```
"""
Filename:
Author:
Date:
Description:
"""
```

6.1a - Pythagoras' Calculator

Write a program that will do the following:

- Print a menu:

    Pythagoras' Calculator

    1 - Find a hypotenuse

    2 - Find another side

    9 - Exit

    Enter an option:

- If '1' is entered, prompt for the two other sides, calculate the hypotenuse and print the answer. Reprint the menu.
- If '2' is entered, prompt for the hypotenuse and the other side, calculate the third side and print the answer. Reprint the menu.
- If '9' is entered, print a goodbye message and exit (`break`)
- If another value is entered, print an error message and print the menu again.

NB: Remember you will need to import the math module (`import math`) and use the `sqrt()` function.

5.1b - Primary Division

Write a program that will do the following:

- Prompt the user for two numbers.
- Calculate, and print the result of the division in the format x remainder y (e.g. 17 / 4 = 4 remainder 1).
- Ask the user if they would like to run the program again or quit.

6.1c - Random Arithmetic

The `random` module lets you generate random numbers (actually pseudo-random, but that's another story) using the function random.randint(x,y) where x and y are the lower and upper boundaries.

To get you started, try this program:

```
import random #import the random module

for loop in range(20): #repeat 20 times
    print(random.randint(1,100)) #print a random integer between 1 and 100
```

Write a program that will do the following:

- Generate two random numbers
- Calculate and print the results if you add, subtract, divide and multiply the two numbers verbosely (e.g. "2 + 3 = 5")
- Ask the user if they would like to run the program again or quit.

6.1d - Code Maker (and Breaker) - Lite Version

Write a program that will do the following:

- Print a menu:

  Code Maker (and Breaker)

  1 - Encode a letter

  2 - Decode a letter

  9 - Exit

  Enter an option:

- If '1' is entered, prompt for a letter and use a selection statement to convert the letter into an integer (a = 1, b = 2, etc.).
- If '2' is entered, prompt for an integer, check it is valid (prompt again if invalid) and convert the integer into a letter.
- If '9' is entered, print a goodbye message and exit (`break`)
- If another value is entered, print an error message and print the menu again.

6.1e - Time Calculator

Write a program that will do the following:

- Print a menu:

    Time Calculator - Arithmetic Mode

    1 - Add 2 times

    2 - Find the difference between 2 times

    8 - Conversion mode

    9 - Exit

    Enter an option:

- If '8' is entered, print a menu:

    Time Calculator - Conversion Mode

    1 - Convert Time to Days

    2 - Convert Time to Hours

    3 - Convert Time to Minutes

    4 - Convert Minutes to Time

    5 - Convert Hours to Time

    6 - Convert Days to Time

    8 - Arithmetic mode

    9 - Exit

    Enter an option:

- If '8' is entered, return to the first menu.
- Complete all of the required procedures.
- Times should be stored as strings in the format DD:HH:MM.
- Days, Hours and Minutes should be stored as integers.

6.1f - ATM - Lite Version

Northern Frock needs you to write a program for their new ATMs (or Automatic Teller Machines). Assuming the starting balance is £67.14, Write a program that will do the following:

• Print a menu:

Welcome to Northern Frock

1 - Display balance

2 - Withdraw funds

3 - Deposit funds

9 - Return card

Enter an option:

• If '1' is entered, display the current balance and also the maximum amount available for withdrawal (must be a multiple of £10)

• If '2' is entered, provide another menu with withdrawal amounts of £10, £20, £40, £60, £80, £100, Other amount, Return to main menu & Return card.

• Check that the requested withdrawal is allowed, print a message to show that the money has been withdrawn and calculate the new balance.

• If 'Other amount' is selected then prompt the user for an integer value. Check this number is a multiple of 10 and that the withdrawal is permitted (as above).

• If '3' is entered, provide another menu that will allow the user to enter an amount to deposit(does not need to be a multiple of £10), return to main menu or return card. If funds are deposited, provide appropriate feedback and update the balance.

• If '9' is entered, print a goodbye message and exit (`break`)

• If another value is entered, print an error message and print the menu again.

6.2 Advanced Exercises

Here are some more challenging exercises. These are a bit more difficult, but are very good practice.

All of these programs should include the following comment block at the start (preferably completed) as well as appropriate comments within the code to explain what is going on:

```
"""
Filename:
Author:
Date:
Description:
"""
```

6.2a - Prime Candidate

Prime numbers are numbers that are only divisible by 1 and itself. The first few prime numbers are 2, 3, 5, 7, 11 & 13.

The easiest way to test for primes is to try dividing it by every number up to the square root of that number.

e.g. tests for 27 and 31. Divide by every number up to the square root of that number (5.196 & 5.57)

27 / 2 = 13 r 1	31 / 2 = 15 r 1
27 / 3 = 9 r 0 (we could stop now, but we won't)	31 / 3 = 10 r 1
27 / 4 = 6 r 3	31 / 4 = 7 r 3
27 / 5 = 5 r 2	31 / 5 = 6 r 1
No point going past 5.196	No point going past 5.57
3 & 9 are factors, so NOT prime	No factors found, so IS prime

The easiest way to do this is to use the % operator (27 % 2 = 1 & 27 % 3 = 0)

Write a program that will do the following:

- Prompt the user for a lower and upper number (limited to a maximum range of 200 and going no higher than 10,000).
- Print all of the prime numbers in that range.
- Ask the user if they would like to run the program again or quit.

6.2b - Binary Search

A binary search is the most efficient way to find a value. It involves splitting the available values into two equal halves and testing which half it is in, and then refining this until the correct value is found.

e.g. A number must be between 1 and 100, in this case it is 53.

Midpoint between 1 and 100 is 50 [`int((1+100)/2)`]. Target number is higher.

Midpoint between 50 and 100 is 75. Target number is lower.

Midpoint between 50 and 75 is 62. Target number is lower.

Midpoint between 50 and 62 is 56. Target number is lower.

Midpoint between 50 and 56 is 53. Target number is found.

Write a program that will do the following:

- Prompt the user for an integer between 1 and 100 (validating and prompting if invalid)
- Use a selection statement to test the value 50 (below, equal, above) and print this guess.
- Repeat the above until the correct number is found.
- Ask the user if they would like to run the program again or quit.

Extension 1: Count and output the number of steps required to find the number.

Extension 2: Generate a random number as the target.

Extension 3: Print all of the values that require exactly 7 steps to find them.

Extension 4: Find the mean average number of steps required to find every number frpm 1 to 100.

Extension 5: Calculate the maximum and mean number of steps required if the range of numbers is 200, 1,000, 2,000, 10,000 & 100,000. Are you surprised by the results?

6.2c - Advanced Strings

You can use the following functions to do interesting things with strings:

```
string = "Strings are sequences of letters"
x = len(string)      #Returns the length of the string                   x = 32
x = string[0]        #Returns the 1st character of the string            x = S
x = string[3]        #Returns the 4th character of the string            x = i
x = string[3:5]      #Returns the 4th and 5th characters of the string   x = in
x = string[3:6]      #Returns the 4th - 6th characters of the string     x = ing
x = str.lower(string[0])      #Returns a lower case string               x = s
x = str.upper(string[3])      #Returns an upper case string              x = I


x = str.islower(string[3])
#Returns a bool depending on whether the character is lower case     x = True

x = str.isupper(string[0])
#Returns a bool depending on whether the character is upper case     x = True

x = str.isdigit(string[5])
#Returns a bool depending on whether the character is a digit        x = False

x = str.isspace(string[7])
#Returns a bool depending on whether the character is a space        x = True
```

Write a program that will do the following:

- Prompt the user for a string
- Calculate and print:

    The number of characters (with spaces)

    The number of characters (without spaces)

    The number of upper case characters

    The number of lower case characters

    An upper case version of the string

    A lower case version of the string

    Whether the string is a palindrome (the same backwards as forwards)

    Print the string in reverse

6.2d - Code Maker (and Breaker) - Full Version

Remind yourself of the lite version of this program (4.1d). This program encoded (or decoded) one character at a time.

Write a program that will do the following:

- Print a menu:

  Code Maker (and Breaker)

  1 - Encode a sentence

  2 - Decode a sentence

  9 - Exit

  Enter an option:

- If '1' is entered, prompt for a string and convert the string into a series of string of digits (a = 1, b = 2, etc.) separated by 1 space per character (space = 27). Print this string.
- If '2' is entered, prompt for a string of numbers separated by 1 space per character and convert into a string. Print this string.
- If '9' is entered, print a goodbye message and exit (`break`)
- If another value is entered, print an error message and print the menu again.

6.2e - Bubble Sort

A bubble sort is a simple (but not particularly efficient) method of putting a list in order.

A bubble sort looks at the first two elements in a list (or array) to see if the first is bigger than the second. If it is, they get swapped round. Either way, the 2nd and 3rd are checked, then the 3rd and 4th, etc...

At the end, the whole thing is done again.

Only when a complete run through happens without any swaps is the list correct.

e.g.

x = [3,9,5,2]

Compare x[0] and x[1], 3 < 9 so no swap

Compare x[1] and x[2], 9 !< 5 so swap [3,5,9,2]

Compare x[2] and x[3], 9 !< 2 so swap [3,5,2,9]

Run again:

Compare x[0] and x[1], 3 < 5 so no swap

Compare x[1] and x[2], 5 !< 2 so swap [3,2,5,9]

Compare x[2] and x[3], 5 < 9 so no swap

Run again:

Compare x[0] and x[1], 3 !< 2 so swap [2,3,5,9]

Compare x[1] and x[2], 3 < 5 so no swap

Compare x[2] and x[3], 5 < 9 so no swap

Run again:

Compare x[0] and x[1], 2 < 3 so no swap

Compare x[1] and x[2], 3 < 5 so no swap

Compare x[2] and x[3], 5 < 9 so no swap

No swaps, so sorting is complete!

Write a program that will do the following:

- Generate an array of 6 random integers.
- Print the list, unsorted.
- Perform a bubble sort.
- Print the sorted list.

6.3 File Handling Exercises

6.3a - Simple Database

Your task is to use a simple database that uses 5 fields - ID, shade, red, green & blue (to represent the RGB colour values for that particular shade, in the range 0-255) with a space between each value. The data file would typically look like this:

1 black 0 0 0

2 white 255 255 255

3 red 255 0 0

4 green 0 255 0

5 blue 0 0 255

- Create a program with a procedure that will create a new file called "colours" and populate it with the data above.
- Create a procedure that will output the entire contents of the file.
- Call both procedures and check that they work.
- Create a function that will print a menu to the screen asking which colour is to be looked up, prompt for a value and return it as an integer.
- Create a function that will look up a given colour (given its ID) and return 4 strings for the shade and RGB values.
- Write a main program that will populate the file, print the menu and print the RGB values for the chosen shade.

Extension 1: Write the menu procedure so that the shade will automatically be shown (rather than being hard-coded)

Extension 2: Improve the menu further so that it will produce as many menu items as there are lines in the data file (it might be useful to use the string "EOF" to signify the end of the file.

Extension 3: Include an extra menu item to allow the user to add their own shades with associated RGB colours. This should include suitable validation to prevent impossible or incomplete entries.

6.3b - ATM - Full Version

Northern Frock needs you to write a program for their new ATMs (or Automatic Teller Machines).

In this version you will need to use a customer database that will initially look like this (ignore the top row):

ID	Title	First Name	Surname	Balance
1057	Mr.	Jeremy	Clarkson	£172.16
2736	Miss	Suzanne	Perry	£15.62
4659	Mrs.	Vicki	Butler-Henderson	£23.91
5691	Mr.	Jason	Plato	£62.17

Write a program that will do the following:

- Generate the above data file.
- Prompt the user for their ID number, validating against the list of known users.
- If the ID number 99999 (5x 9s) is given the machine should shut down.
- Print a menu:

    Welcome to Northern Frock

    1 - Display balance

    2 - Withdraw funds

    3 - Deposit funds

    9 - Return card

    Enter an option:

- If '1' is entered, display the current balance and also the maximum amount available for withdrawal (must be a multiple of £10)
- If '2' is entered, provide another menu with withdrawal amounts of £10, £20, £40, £60, £80, £100, Other amount, Return to main menu & Return card.
- Check that the requested withdrawal is allowed, print a message to show that the money has been withdrawn and calculate the new balance.
- If 'Other amount' is selected then prompt the user for an integer value. Check this number is a multiple of 10 and that the withdrawal is permitted (as above).
- If '3' is entered, provide another menu that will allow the user to enter an amount to deposit(does not need to be a multiple of £10), return to main menu or return card. If funds are deposited, provide appropriate feedback and update the balance.
- If '9' is entered, print a goodbye message and return to the initial prompt (for the next customer).
- If another value is entered, print an error message and print the menu again.

Extension 1: Add an extra field to the database for a 4 digit PIN which should be prompted for and checked following the entry of the ID number. The user should also have the option to change their PIN.

Extension 2: Add another field to record whether the card is blocked. Three incorrect PIN attempts should permanently lock the card. PIN attempts should only be reset by correctly entering the PIN. Simply removing the card and trying again should not work.

Extension 3: Create an encoding algorithm that will store the PIN in a format that cannot be read by just looking t the file. Only by running the PIN attempt through the same algorithm can the match be found.