Introduction to Python Programming

This is an informal introduction to Python programming. It introduces the beginner to some fundamental programming concepts such as: input/output, variables, control flow, file I/O and functions. The high-level language Python, is used to illustrate the concepts. Python was chosen because of its widespread popularity and use as a programming language However, you should not be overly concerned with the details of the C language.

Overview

One point about programming must be clarified immediately: **Anyone can learn to program computers.** This may or may not surprise you. Many people have misconceptions about what skills you need to write computer programs e.g. do you need to be logical or mathematical or interested in electronics? The author's contention is, as already stated, that anyone can learn to program. However, you must be willing to spend some time studying and practising. The same applies to acquiring any skill such as driving a car, learning to swim, learning to play poker and so on.

You can view programming as a skill acquired from study, training and practice. In order to learn how to program, you will sooner or later have to use a computer. Learning to use the computer is a separate and independent skill. In fact, strange as it may seem, you need to know very little about using computers in order to program them! At the minimum, you need to know how to switch on the computer, enter your program and have it executed. This can be mastered in about an hour! The important point is, that **using a computer is a separate skill to programming one**. When you have problems in your early days of programming, try to identify whether they have to do with using the computer or with your programming ability.

There are two aspects to programming that must be mastered. One concerns **problem solving** and the other concerns the **programming language** that is to be used. You must learn how to solve problems. This is the core of programming. But you also must learn a programming language to express your problem solution in, so that it can be carried out on a computer. Again, these are two separate skills. You must try not to confuse them. It is difficult, however, to explain one without reference to the other. In summary, a programmer must acquire three skills:

1. Computing skills - (how to switch on and use a computer).
2. Problem solving skills.
3. Programming language skills.

Computing Skills

These are the easiest to acquire and you most likely have them already. For the purposes of this introduction, you need to know how to start your computer system, how to use a word processor (or editor) to enter your programs and save them. Finally you need to be able run your programs in Python.

Problem Solving

Computer programming is about problem solving. Every computer program solves some particular problem, even programs to play games. It is impossible to write a computer program unless you understand the problem you are being asked to solve. In programming, **you solve** the problem, not the computer. A computer program describes to the computer **what** it must do and **how** it is to be done. You give the computer instructions in the form of a program, telling it what to do and how to do it. **The set of instructions required to solve a problem is a computer program.** It is also the solution to the problem, because when the computer follows these instructions, it will produce the answer i.e. the required results (assuming the program is correct).

The term **algorithm** is used to describe the **set of steps that solve a problem**. An algorithm may be expressed in English or in a formal computer language, whereas a computer program must be expressed in a programming language. In programming, we first develop an algorithm for the problem at hand, and then we translate this algorithm to a computer program, so that it can be executed on a computer.

Sometimes we make mistakes in telling the computer what to do. We overlook part of the problem or do not understand what to do ourselves. In these cases, the computer program will not produce the "right answer". It is, however, still solving a problem. It is simply not the problem we wanted to solve. For this reason, it is important to thoroughly check that your programs do indeed solve the problem you intended. It is important to note that this testing does not prove that programs are correct, it shows that they are correct for the tests used. Program correctness is a major area in computer programming and is not addressed here.

An important principle concerning problem solving is that we can, and should, solve problems independently of any programming language. Only when we have solved a problem should we consider the programming language. Beginners find this hard to understand, but it is worth repeating that we do not need a programming language to solve problems. Of course, when it comes to implementing our solution and testing it, then we must use a programming language. We distinguish two phases in programming: the **problem-solving phase** (analysis phase, design phase), and the **implementation** phase. It is very important to distinguish between these phases and keep the two separate. Beginners (and others!) continually make the mistake of rushing into the implementation without fully considering the problem to be solved. The seriousness of this mistake is not too obvious when we write short programs of a few dozen lines of code where we can easily scrap our ideas and start all over. But in large programs consisting of hundreds or thousands of lines of code taking months to design and implement, such an error can be extremely costly i.e. very time-consuming to correct.

Problem Solving Techniques

Given a problem to solve, where do we start? One important principle here is to **"divide and conquer"**. In problem solving terms this means taking your problem and dividing it into subproblems. Then you tackle each one of these separately. If necessary, divide the subproblems into further subproblems. Continue this process of dividing into subproblems and tackling each one separately until you can write the solution to each subproblem. When you have solved all your subproblems, you have in effect, solved your initial problem. This technique is used in many aspects of problem solving. For example, if you have to decorate an entire house (a large problem), you typically divide the job into tasks to decorate the individual rooms. Taking each room, you create separate tasks of decorating the walls, ceilings and floors. When you have finished all these smaller tasks, the whole house will have been decorated and the initial large problem solved. This same approach when applied to programming is called **top down programming**. It is one of several programming techniques.

Another useful technique, which may be used in the conjunction with the above is to write down all the input your program is going to work on. Then write down the output you expect the program to produce. Solving the problem then becomes a question of transforming or processing the input to produce the output. In order to write computer programs you must learn to instruct the computer how to perform input, processing, and output operations. You must learn the instructions that correspond to these operations. These instructions are given to the computer in the form of a computer program which is written in a programming language.

1.1 Programming Languages

Just as people use differnetent languages to talk to each other, so you must use a language to communicate with a computer. There are many programming languages which can be used including C, C++, Java, Perl, Python and BASIC. These are called high-level languages because they are problem oriented i.e. they are designed to help you solve problems. This means that they have facilities that make it easier to implement the solution to problems than so called low-level languages. There are also many low-level languages or assembly languages which depend on the computer that is being used.

Most programmers will, at any time, typically be working with one language, although they may be familiar with several. It is important to note again that a *knowledge of programming techniques is independent of any programming language*. Armed with such a knowledge and a knowledge of one programming language, it is quite straightforward to learn another language. Put another way, **the first programming language you learn will be the most difficult.** It is worth noting, that the experience of learning a second or third language, will serve you well in increasing your understanding of programming.

One of the reasons for the variety of languages is that some languages are designed for particular kinds of problem. Assembly languages are used where efficiency is extremely important or where other languages cannot be used, because certain operations can be carried out using assembly language that cannot be carried out using a high level language.

As we noted earlier, a given problem can be solved independently of a programming language. You then have the choice of which language to use. Almost all languages will allow you do the job but some make it easier by providing facilities that are useful for the particular task at hand. Often, the decision as to which language to use is easy - if you only know one language, you have no choice! Similarly, if your employer only wants Java programs, then you must program in Java. Ideally, you should choose the language with the facilities best suited for the problem at hand.

Language Structure

All languages have a grammar. This is a set of rules about what constitutes a valid sentence in the language. A grammar helps people to communicate with each other. However, people can still understand each other when they make grammatical mistakes. For example, if I write in English: “I am going to town and I will bought some things”, my grammar is incorrect I should have written “buy” in place of “bought” but you understand what I meant. This is a fundamental aspect where computers differ from people. Computers **do not understand instructions**, they simply carry them out. They must be given instructions that specify exactly what they must do. If you make a mistake entering an instruction, a computer will not understand the instruction and it will display an error message. This message means that there is a mistake in your instructions - often a spelling error or a missing bracket or quotation mark. For example, in a programming, just as in English, you must always have the correct number of quotation marks and brackets. **Left** brackets such as (, { and [are called **opening** brackets. The quotation marks at the start of a phrase:"(double) and ' (single) are called **opening** quotes. **Right** brackets such as) , } and] are called **closing** brackets and quotation marks at the end of a phrase **closing** quotes.

*A simple rule is that **for every opening** bracket or quotation mark you must have a **corresponding closing** bracket or quotation mark.*

Example 1.1: Matching brackets and quotes:

```
“Hello“
( a > 10 )
'x'
a[10]
```

Leaving out a bracket or quote, or misspelling, is called a **syntax error**. Each programming language has its own syntax. When you make a syntax error you may get an unhelpful message from the computer such as:

line 23 syntax error

which tells you that there is an error on line 23. You must figure out what is wrong. In fact, the error may be on a previous line such as 21 or 22! The computer only detected that there was an error when it reached line 23.

What to do when an error occurs

When a syntax error occurs, you must work out what mistake(s) you have made. This means checking the statements of your program and seeing where the syntax is incorrect. You then **edit** your program to correct the mistake. When you have corrected your program, it can be translated into machine language and executed. To execute a program or to run a program means that the computer carries out the instructions making up the program. This is just the same as starting a word processor or spreadsheet. They too are programs which you run. They are no different from the programs you write. **To a computer, all programs are the same,** in the sense that they are all simply sets of instructions, telling the computer what to do.

Statements
When people communicate in any language, they use sentences to convey information. When you write down a sentence in English, you have a full stop at the end. This tells us where the sentence ends. Without it, it would be difficult to read English text. You could call the full stop a sentence terminator i.e. it indicates the end of a sentence.

Similarly, when you write programs you also use sentences to communicate with the computer. In programs, sentences are called **statements**. They also have a terminator. Different languages use different terminators. Python statements use the **end of line** to terminate a statement:

```
pension_age = 65
my_age = 42
```

After you entered 65 you pressed the Return key on the keyboard to put star a new line – we call this the **newline** character. Similarly after you entered 42, you also pressed Return thus starying a new line and ending the current one.

You can use the semicolon ";" as the statement terminator, which will allow you put two or more statements on the same line.

```
pension_age = 65;  my_age = 42
```

It is easier to read programs if we write statements **one per line** and I would strongly recommend this approach.

Machine Code/Language –(Historically, computers were called **machines)**
Computers can only process instructions and data that are in binary form i.e. made up of 1's and 0's. They cannot directly process instructions written in programming languages such as Java and Python. The language understood by a computer is called **machine code.** The first computer programs were written in machine code. Writing programs in machine code is a difficult and error prone task. As computing advanced, better languages which could be understood by people were developed. The first of these were the assembly languages. Then followed languages such as FORTRAN (for Science and Engineering applications) and COBOL (for Business applications).

Remember that the machine can still only process machine code. So, to write programs in Java or Python (which we can more easily understand), these programs must be **translated** into the computer's machine code. This translation is carried out by computer programs called **translators**. There are different types of translators called **assemblers**, **compilers** and **interpreters**. Compilers and interpreters take programs written in high level languages such as Java, C and Python and translate them to machine code. In the case of **Python we use an interpreter**. They also check them for syntax errors before producing the machine code translation. If a translator discovers a syntax error, it displays the error message (as mentioned earlier). The translator will only produce a machine code program if there are no syntax errors. The machine code program is usually called an **executable program**. Assemblers are translators that are used for assembly languages such as x8086 or M68000 assembly language.

Summary
To summarise, programming involves:

- solving problems
- expressing this solution in a computer language in the form of a program
- translating this program into machine language
- executing and testing the program

The Python Programming Language

Note: There are different versions of Python and we are using **Python 3**.

A Python program is made up of a group of statements. These statements allow us to control the computer. Using them, we can **display information on the screen**, **read information from the keyboard, store information on disk and retrieve it** and we can **process information** in a variety of ways.

We can classify statements as:
I/O statements,
variable manipulation statements (e.g. to do arithmetic) and
conditional statements (described later)

In this section we will look at I/O and variable manipulation. Associated with the different types of statement is a set of special words called **reserved** words (**keywords**). Every programming language has its own set of reserved words. These words have a special meaning in the language and can only be used for particular purposes. The following are some of the reserved words of the Python language that will be used in this text: `int`, `if`, `for`, and `while`. All program code and variable names will be printed using the `Courier font` in this text.

I/O Statements: Output
Output is the term used to describe information that the processor sends to peripheral devices e.g. to display results on a screen or to store information in a disk file. One of the commonest forms of output is that of displaying a message on your screen. In Python, we use `print` to display output on the screen. The following `print` statement will display the message `My name is Beth. This is my first program on the screen`

```
print( "My name is Beth. This is my first program" )
```

This is a single Python program statement. To have it executed, it is stored in a file which we call program1.py. This file was created by an text editor. The file contains the one line:

```
print( "My name is Beth. This is my first program" )
```

To execute the program we use the command python3 which translates and runs the program:

```
% python3 program1.py
My name is Beth. This is my first program
```

Such a message is called a string constant as it will never change.

I/O Statements: Input and Variables

Input is the term used to describe the transfer of information from peripheral devices to the computer e.g. input may come from the keyboard or from a disk file. Before we describe input statements, let us consider where to store the information to be read in. We must arrange to store the input so that it can be processed. This introduces the concept of **variables**. **A variable may be viewed as a container for a value**.

Therefore to take input into a program we input data into a variable(s). But how do we identify this variable and distinguish it from other variables? The solution is simple, **we give each variable a unique name**, which we use to identify it. The following are examples for variable names we could use in a Python program:

```
colour
my_age
pension_age
name
taxcode22
temperature6
```

We can use any name we wish for variables with the exception of the **reserved** words that Python uses.

Fundamental principle of writing clear programs

Choose meaningful names for variables,
because it makes your programs easier to understand.

For example, if you are writing a program which deals with pension ages then you could use any of the following names to store the pension age but which one makes is easiest to understand:

```
pension_age
pa
p
x
pna
```

The variable name `pension_age` is the obvious choice. When you see this name you automatically know what it the variable is being used for. If you use a name like `p` or `x` then the name gives you no idea what the variable is being used for.

Tip: Use long variable names where they make sense.

Comments
In a Python program, any text after **#** is called a comment and **is ignored by Python**. They are intended as text (**documentation**) to help explain to someone reading the program, what the program does and how the program works. **Comments are an important component of programs**. This is because when you read your programs some time after writing them, you may find them difficult to understand, if you have not included comments to explain what you were doing. They are even more important if someone else will have to read your programs e.g. your tutor who is going to grade them! It is a useful idea to give the name of the file containing the program, the authors name and the date on which the program was written, as the first comments in any program as shown in the example above.

Example 1: Write a program to prompt the user to enter their favourite colour. The program reads this colour and displays a message followed by the colour entered by the user. The program may be written in Python as follows:

```
# colour.py: Prompt use to enter colour and display a message
# Author: Joe Carthy
# Date: Oct 20 2022

favourite_colour = input("Enter your favourite colour: ")

print("Yuk ! I hate ", favourite_colour )
```

If we execute the program the following appears on the screen (the bolded text is that entered by the user. We will use this convention throughout the text).

```
Enter your favourite colour: blue
Yuk ! I hate blue
```

The variable `favourite_colour` is going to be used to store the characters that the user types on the keyboard that is, it will store a list of characters. A list of characters is called a **string**. Strings are used in almost every program we write.

Strings can also be represented by characters inside quotes (single or double):

```
"Enter your favourite colour: "
'Enter your favourite colour: '
```

The `input()` statement does two tasks: it displays the string in quotes and then reads text from the keyboard, (for example the word *blue* may be entered), and it places the text in the variable `favourite_colour`.

The `print()` statement is used to display output on the screen. It can be used to display strings and numbers.

```
print("Yuk ! I hate ", favourite_colour )
```

instructs the computer to display the message *Yuk ! I hate* followed by the value of the variable `favourite_colour` i.e. *blue* in this example.

When you use a variable name with print() it will display the value of a variable.

We can use **input()** to give values to variables and print() to display the value contained in any variable. We use the expression “the value of a variable” to mean “the value contained in a variable”. We take the phrase “the value of favourite_colour is blue” to mean “the value contained in the variable called favourite_colour is blue”. We will use the shorter form from now on.

From the program above, we see that print() has the ability to display messages enclosed in quotation marks. But print() can also display the values of variables. For example:

```
print ( favourite_colour )
```

displays the value of the variable favourite_colour, which in this example is also a string i.e. a list of characters.

Make sure you understand the difference between:

```
print( "favourite_colour" )
```

and

```
print( favourite_colour )
```

In the first case, a string constant is displayed, i.e. the word *favourite_colour* appears on the screen. Any message inside quotes is called a **string constant** because every time you run the program, it remains constant i.e. it does not change.

In the second case, the value of a variable called favourite_colour is displayed which could be anything, for example the word *blue* or whatever value the user has given the variable like *red*, *pink* and *orange*. You can store **many words** in a string variable.

More about variables: Assignment Statement
In Example 1, we saw that we can directly input a value into variable. There is also another way to give variables a value. It is called **assignment**. It allows us to give a value to the variable directly in a program without input. We may give the variable a constant value or compute a value based on the values of other variables. For example, suppose we have a variable called metres, to which we wish to give the value 12. In Python we write:

```
metres = 12
```

This can be read as “metres is assigned the value 12” or “metres becomes 12”. Of course, we could use any value instead of 12. Other examples of assigning values to variables might be:

```
centimetres = 50
litres = 10
metres = 4
```

Example: Write a program to convert metres to centimetres. A simple (and fairly useless) Python program to do this is given below. This is version 1 of the program, other versions are developed as we proceed through the chapter.

```
#convert.py: converts metres to centimetres
#Author: Joe Carthy
#Date: 21/10/2022

metres = 5
centimetres = metres * 100
print("The number of centimetres is ", centimetres )
```

Executing this program produces as output:

```
% python colour.py
The number of centimetres is 500
%
```

Here we use the value of the variable `metres` to compute the value of the variable `centimetres`.

Other examples of such an assignment are:

```
          pints = gallons * 8
          kilom = 4
          metres = 18
          cms = (kilom * 100000) + (metres * 1000)
```

where the values of variables on the right hand side are used to compute the values assigned to the variables on the left hand side of the assignment.

The program to convert metres to centimetres as presented in Example 2 is very limited in that it always produces the same answer. It always converts the same quantity of metres (5) to centimetres. A more useful version would prompt the user to enter the number of metres to be converted and display the appropriate result:

Example 3: Converting metres to centimetres, version 2.

```
#convert2.py: converts metres to centimetres version 2
#Author: Joe Carthy
#Date: 21/10/2022

metres = int (input("Enter number of metres: "))

centimetres = metres * 100

print(metres, "  is ", centimetres )
```

Executing this program produces as output:

```
% python convert2.py
Enter number of metres: 4
4 metres is 400 centimetres
%
```

Important note: The `input` function reads from the keyboard and returns a `list of characters` i.e. a string. Thus if we write

```
metres = input("Enter metres" )
```

the variable `metres` will contain the string "4" as opposed to the number 4. This is very confusing for beginners to programming. A fundamental aspect of variables is that they have a **type**. The type of a variable tells you what kind of data it stores. In our early programs we will use three types: **int** (whole numbers), **float** (numbers with decimal point) and **string** (list of characters).

When you are working with numbers and wish to do arithmetic with them (add, subtract, multiply and divide) then you must use the type **int** or **float**.

If you are storing a mobile phone number then you use a string because you will never use arithmetic on a mobile phone number.

So it is crucial to understand the difference between the number 42 and the string "42" as used in the following:

```
a = 42
b = b * 2
```

This results in `b` having the value 84.

```
x = "42"
y = x * 2
```

This results in `y` having the value `4242`.

When you "multiply" a string variable by `n` you get `n` copies of the string e.g.

```
x = "bye"
y = x * 3
```

gives `y` the value `"byebyebye"`

This brings us back to the statement

```
metres = int (input("Enter number of metres: "))
```

The `int` function converts the string from `input` to a number, in this case an `int`. This means that `metres` now contains a number which we can do arithmetic with.

However, an int variable can only hold whole numbers i.e. number without a decimal place. Thus our conversion program will only work when we enter whole numbers. If we wish to work with floats (number with a decimal point also called real numbers) we need to convert the type to float:

```
metres = float (input("Enter number of metres: "))
```

The full programs is shown in Example 4.

Example 4: Converting metres to centimetres, version 3 using floats.

```
#convert3.py: converts metres to centimetres version 3
#Author: Joe Carthy
#Date: 21/10/2022

metres = float (input("Enter number of metres: "))

centimetres = metres * 100

print(metres, "metres is ", centimetres, " centimetres"  )
```

```
% python convert3.py
Enter number of metres: 3.5
3.5 metres is 350.0 centimetres
%
```

The output above is "crowded" in that there is no blank line before or after the output or between the two lines of output. This makes it hard to read the output. You can use the "\n" character in strings to start new lines.

The version below fixes this issue by putting one "\n" in the input() function and 3 in the print() function.

```
# convert4.py: converts metres to centimetres version 3
# Outputs extra blank lines to make it easier to read the output

#Author: Joe Carthy
#Date: 21/10/2022

metres = float (input("\nEnter number of metres: "))

centimetres = metres * 100

print(“\n”, metres, "metres is ", centimetres, " centimetres\n\n"  )
```

When you run it, notice the extra blank lines

```
% python convert4.py

Enter number of metres: 3.5

3.5 metres is 350.0 centimetres

%
```

Some Fun making the computer beep!

When you use the “\a” character in the print() function, the computer makes a beep sound – it does not display anything. So the program below simply plays 3 beeps.

```
# beep.py: Just for fun - beep 3 times !!

print("\a \a \a")
```

Example: As another example of the use of I/O and variables consider a simple calculator program. This program prompts for two numbers, adds them and displays the sum:

```
# calc.py: Calculator program to add 2 numbers
# Author: Joe Carthy
# Date: 01/10/2022

number1 = float(input("\nEnter first number: "))

number2 = float(input("\nEnter second number: "))

sum = number1 + number2

print("\n\nThe sum of", number1, "and", number2, "is", sum, "\n\n")
```

`calc.py` outputs:

```
Enter first number: 2.4

Enter second number: 5.76


The sum of 2.4 and 5.76 is 8.16
```

Note in this program, we illustrate that a single print() can display the value of a number of variables, in this case the values of three variables are displayed.

Arrays
An array is a named list of items such as characters, floats or integers. In programming terminology, an array is an example of what is called a **data structure**.

Python does not have arrays ! But it does have strings and lists which are very similar.

It is easy to create a string in Python and we have already done in several earlier programs:

```
primary_colours = “red, orange, yellow, green, blue, indigo, violet”

colours = “pink, white, black, brown”
```

We can also use input() to create a string”

```
address = input(“\nEnter your address on 1 line”)
```

A string is made up of elements, in the examples above, the elements are the individual characters that make up the string. We can access an element of a string, by using its position also called its **index** or **subscript** e.g. address[0] refers to the first element in the address string, colours[1] refers to the 2nd element in the colours string.

Python specifies that strings begin with index 0. While this seems unnatural, it is quite common in computing to count from 0. So `colours[0]` is the character ‘p’, `colours[1]` is the character ‘i’. You can access the characters in a string, one at a time using the index. The index indicates which letter is required from the string. For example, consider the following code:

```
animal = “elephant”

letter = animal[1]

# letter now contain ‘l’

print(“first 3 letters are:”, animal[0], animal[1], animal[2])
```
displays the string

```
ele
```

Python allows you break strings into components called **segments** or substrings. In Python, a segment of a string is called a **slice**. Selecting a slice is done in a similar way to selecting a character, for example:

```
colours = “pink, white, black, brown”

seg1 = colours[0:4]          # “pink”

seg2 = colours[6:8]          # “wh”

seg3 = colours[10:14]        # “e, b”

print (seg1, seg2, seg3)
```

`colour[0:4]` means extract the substring starting at 0 and ending at 3 (4-1) thus giving the substring “`pink`”

`colour[6:8]` means extract the substring starting at index 6 and ending at 7 (8-1) thus giving the substring “`wh`”

`colour[10:14]` means extract the substring starting at index 14 and ending at 13 (14-1) thus giving the substring “`we, b`”

So in general we can use the formula

```
any_string[start:stop]
```

which means extract the substring starting at position **`start`** and ending at position **`stop -1.`**

We will see in later programs that being able to break strings into slices is very useful.

Conditional Statements

People are used to making decisions. For example, consider the following sentences:

> If I get hungry, I will eat my lunch.
> If it gets cold, I will wear my coat.

These two sentences are called **conditional sentences**. Such sentences have two parts: a **condition part** ("If I get hungry", "If it gets cold") and an **action part** ("I will eat my lunch", "I will wear my coat").

The action will be only be carried out if the condition is satisfied. To test if the condition is satisfied we can rephrase the condition as a question with a yes or no answer. In the case of the first sentence, the condition may be rephrased as "Am I hungry ?" If the answer to the question is yes, then the action will be carried out (i.e. the lunch gets eaten), otherwise the action is not carried out.

We say the condition **is true** (**evaluates to true**) in the case of a yes answer. We say the condition **is false** (**evaluates to false**) in the case of a no answer. Only when the condition is true will we carry out the action. This is how we handle decisions daily.

In programming, we have the same concept. We have **conditional statements**. They operate exactly as described above. One of the most fundamental of these is known as the i**f statement**. This statement allows us evaluate (test) a condition and carry out an action if the condition is true.

In Python, the keyword **if** is used for such a statement. As an example, we could modify the program to convert metres to centimetres to test if the value of metres is positive (greater than 0) before converting it to centimetres.

The action statement(s) are indented in Python. In the program below, both if statements have action parts with 2 statements. The action statements end with the first non-indented statement follow the if. Note you put a "**:**" after the condition in an if statement

```
# convert5.py: converts metres to centimetres version 3
# check quantity of metres is positive
# Outputs extra blank lines to make it easier to read the output
# Author: Joe Carthy
# Date: 21/10/2022

metres = float (input("\nEnter number of metres: "))

if metres > 0:
    centimetres = metres * 100
    print("\n", metres, "metres is ", centimetres, " centimetres\n\n")

if metres <= 0:
    print("\nPlease enter a positive number for metres\n")
    print("\nYou entered: ", metres \n\n")
```

Executing this program with -42 as input produces as output:

```
Enter number of metres: -42
Please enter a positive value for feet

You entered -42
```

The first if statement tests if the value of metres is greater than 0 (metres > 0). If this is the case, then the conversion is carried out and the result displayed. Otherwise, if the value of metres is not greater than 0, this does not happen i.e. the 2 action statements are skipped.

The second if statement tests if metres is less than or equal to 0. If this is the case, then the message to enter a positive value is displayed and the value entered is displayed. If this is not the case the print is skipped and the program terminates.

In this particular example, only one of the conditions can evaluate to true, since they are **mutually exclusive** i.e. metres cannot be greater than 0 and at the same time be less than or equal to 0. Because this type of situation arises very frequently in programming i.e. we wish to carry out some statements when a condition is true and other statements when the same condition is false, a special form of the if statement is provided called the if-else statement.We rewrite the above program to illustrate its usage:

```
# convert6.py: converts metres to centimetres version 3
# check quantity of metres is positive
# Outputs extra blank lines to make it easier to read the output
# Author: Joe Carthy
# Date: 21/10/2022

metres = float (input("\nEnter number of metres: "))

if metres > 0:
    centimetres = metres * 100
    print(“\n”, metres, "metres is ", centimetres, " centimetres\n\n")

else:
    print(“\nPlease enter a positive number for metres\n”)
    print(“\nYou entered: ”, metres \n\n")
```

This program operates in the same way as the previous example. However, it is more efficient, in that the condition has only to be evaluated once, whereas in first example, the condition is evaluated twice.

Another example: The program below prompts the user to enter the number of hours worked in a week and the rate of pay per hour. Workers can only work a maximum of 100 hours per week and the maximum hourly pay rate is 50.

```
# pay.py: Calculate and display hourly pay

hours_worked = float(input("\nEnter number of hours worked: "))

if hours_worked > 100.0 :
    print("\nHour worked too large:", hours_worked)
else:
    rate_per_hour = float(input("\nEnter rate per hour: "))
    if rate_per_hour > 50:
        print("\nRate per hour too high ", rate_per_hour)
    else:
        pay = rate_per_hour * hours_worked
        print("\nPay = ", pay, "for ", hours_worked, "hours")
```

```
Enter number of hours worked: 20

Enter rate per hour: 20

Pay =  400.0 for  20.0 hours
```

There are only six types of condition that can arise when comparing two numbers.

They can be tested for
1. equality - are they the same ?
2. inequality – are they different ?
3. is one greater than the other ?
4. is one less than the other ?
5. is one greater than or equal to the other ?
6. is one less than or equal to the other ?

The following illustrates how to write the various conditions to compare the variable feet to the number 0 in C:

(feet == 0)	is feet equal to 0?
(feet != 0)	is feet not equal to 0?
(feet > 0)	is feet greater than 0?
(feet < 0)	is feet less than 0?
(feet >= 0)	is feet greater than or equal to 0?
(feet <= 0)	is feet less than or equal to 0?

Technically, the symbols ==, <>, <, >, <=, and >=, are called **relational operators**, since they are concerned with the relationship between numbers.

We call a condition (e.g. feet < 0) a **Boolean expression** or a **conditional expression**. This simply means that there are only two possible values (true or false) which the condition can yield.

The term **expression** is widely used in programming. Informally it means something that yields a value. We are familiar with arithmetic expressions such as 2+2 which evaluates to 4.

A Boolean expression is one which evaluates to either true or false.

Examples of expressions include constants (0, 100, ‘a’), variables (feet, inches) and arithmetic expressions (feet * 12, 4 / 8).

The right-hand side of an assignment statement is always an expression.

As an example, let us modify the calculator program to handle either subtraction or addition. The user is prompted for the first number, then for a ‘+’ or ‘-’ character to indicate the operation to be carried out, and finally for the second number. The program calculates and displays the appropriate result:

```
# calc2.py: Calculator program to add 2 or subtract numbers
# Author: Joe Carthy
# Date: 01/10/2022

number1 = float(input("\nEnter first number: "))

operation = input(“\nEnter operation + or –“)

number2 = float(input("\nEnter second number: "))

if operation[0] == ‘+’:
    sum = number1 + number2
    print("\n\nThe sum of",number1,"and",number2, "is", sum, "\n\n")
else:
    diff = number1 - number2
    print("\n\nTaking ",number2,"from",number1, "is", diff, "\n\n")
```

Executing this program produces as output:

```
Enter first number:  9

Enter operation (+ or -): -

Enter second number: 4

Taking 4.0 from 9.0 is 5.0
```

Note we use the array element **operation[0]** to check the first character the user entered.

The above programs “assumes” that if the operator is not ‘+’ then it must be ‘-‘ but te user could have hit the wrong key. The following version checks for ‘+’, ‘-‘ and the possibility that it was neither ‘+’ or ‘-‘ that is the user made a mistake. User data entry mistakes are very common and professional programs always check that the user input is as was expected.

We use a third variant of **if** in the program below called *if .,. elif..else*

```
# calc3.py: Calculator program to add or subtract 2 numbers
# Author: Joe Carthy
# Date: 01/10/2022

number1 = float(input("\nEnter first number: "))

operation = input(“\nEnter operation + or –“)

number2 = float(input("\nEnter second number: "))

if operation[0] == ‘+’:
    sum = number1 + number2
    print("\n\nThe sum of",number1,"and",number2, "is", sum, "\n\n")
elif operation[0] == ‘-’:
    diff = number1 - number2
    print("\n\nTaking ",number2,"from",number1, "is", diff, "\n\n")
else:
    print(“\nInvalid operation only + and – allowed\n”)
    print(“You entered: “, operation[0])
```

Executing this program produces as output:

```
Enter first number:  9

Enter operation (+ or –): *

Enter second number: 4

Invalid operation – only + and – allowed
You entered: *
```

Fundamental Principles about conditionals

• A condition can only **evaluate to true or false**.

• The action(s) associated with a **condition is carried out only if the condition is true**.

Tip:
To evaluate a condition, simply re-phrase it as a question. The answer is yes for true and no for false.

Conditions are basically comparisons. We compare two things and based on the comparison (whether it is true or false) we take a certain course of action. Conditional statements allow you alter the **control flow** in a program are thus called **control structures**. Control flow means the **order in which statements are executed**. In our first programs, we had a linear control flow – statements were executed in sequence one after another.

There are two basic types of control structure in programming. The if statement is called a **selection control structure**. It allows you select an alternative action i.e. make a decision as to what to do next. The other type of control structure is the **loop** (also called the **iteration** control structure). A loop allows you repeatedly execute a statement(s).

More on print() function and displaying variables

Take the following variables and how we want to display them:

Name = 'Joe Bloggs'
rate = 10.00
num_hours = 40
pay = rate * num_hours

```
print("Pay for ", name, "at ", rate, "per hour is", pay)
```

outputs

```
Pay for Joe Bloggs at 10.0 per hour is 400.0
```

There is a simpler way to display this message with print using **f-strings**:

```
print(f"Pay for {name} at {rate} per hour is {pay}")
```

displays the same output as the first `print()` above.

```
Pay for Joe Bloggs at 10.00 per hour is 400.00
```

Note: We must **put the character f before** we start the string in `print()`**.**

When using an f-string, we enclose any variable we wish to display in `{}` brackets.
`print()` The will display the value of each variable in `{}`.

Displaying a fixed number of decimal places

In most of our calculations it is common to display the results with 2 decimal places. We can use an f-string to do this.

```
x = 19/3.768
```

```
On my Mac computer if I print x it will display as 5.042462845010616
```

To print `x` to 2 decimal points

```
print(f"x = {x:.2f}")
```

outputs

```
x = 5.04
```

`x:.2f` specifies to print the value of x to decimal places. You can change the number from 2 to whatever you wish, to have that number of places displayed after the decimal point.

Loops: Repeating parts of a program

So far, all our programs have carried out one major task such as converting a single quantity of metres to centimetres. Frequently, we want to repeat such a calculation. Say we have thirty values for metres which we want to convert to centimetres. Using the program described earlier, we would have to run it 30 times to achieve the desired result. Programming languages provide loops to allow us repeat part of the program as many times as we wish. For example, in the conversion program we can write the program to repeat the process of reading a value to be converted and displaying the result, 30 times or any number of times. This is called **looping** (or iteration).

The while loop

There are a number of looping techniques, but basically all program looping can be performed using one particular looping construct called a **while** loop. The other mechanisms are provided for convenience. Loops are another form of conditional statement. In the case of a loop, we use the condition to decide whether to repeat a statement or not. We repeat the statement based on the evaluation of the condition in a similar fashion to carrying out the action part of an if statement. The action part of a loop is referred to as the **loop body**. This may be a simple or compound statement (group of statements). The loop body is executed only if the condition evaluates to true, the condition is then re-evaluated to test if it is still true. If it is, we repeat execution of the loop body and test the condition again. This process continues until the condition evaluates to false.

In certain situations, the condition will never evaluate to false and the loop will continue to execute endlessly. Such a loop (usually the result of a programming error) is called an **endless** or **infinite** loop. An endless loop may be terminated by interrupting the program or switching off the computer, both of which terminate the program as. To interrupt a program, a combination of keys is pressed, such as pressing the control key and the C key simultaneously (denoted by Ctrl/C). The operating system detects the interrupt and terminates the program.

Modify the calculator program to sum 5 pairs of numbers. In other words we wish to read in the two numbers to be summed, calculate the sum and display the result, ten times, by running the program once. We use a while loop to repeat the necessary statements:

```
# calc4.py: Calculator program to add 2 numbers, five times
# Author: Joe Carthy
# Date: 01/10/2022


count = 1

while count <= 5:
    number1 = float(input("\nEnter first number: "))
    number2 = float(input("\nEnter second number: "))
    sum = number1 + number2
    print("\nThe sum of", number1, "and", number2, "is", sum, "\n\n")
    count = count + 1

print (“Finished summing\n”)
```

`calc4.py` outputs:

```
Enter first number:  4
Enter second number: 6
The sum of 4 and 6 is 10

Enter first number:  20
Enter second number: 30
The sum of 20 and 30 is 50

Enter first number:  65
Enter second number: 30
The sum of 50 and 65 is 115
     .........
     .........
Finished summing
```

The while statement tests the condition (count <= 5) and if it evaluates to true, the statements in the loop body are executed and the condition is re-evaluated.

We assigned count the value 1 which is called initialising count or giving count an initial value. When we first assign a value to a variable, we say we have initialised the variable.

Because count has the value 1, then the condition will evaluate to true and the loop body is executed, increasing count by 1, so now it has the value 2.

When the condition is false i.e. when count reaches 6, we skip the action specified by the loop body, and in this example, we execute the final print() statement and the program terminates.

Each time we execute the loop body (go around the loop), we process one pair of numbers and **add 1 to count**. The variable count is used in this example to control how many times we execute the loop body. Such a variable is called the **loop counter**.

So after executing the loop action 5 times, count will have the value 6. Each time you execute the loop, the condition is tested. You only execute the loop body if the result is true. So when count has value 6, we leave the loop (the loop terminates), i.e. we go to the next statement after the loop body if any.

What would happen if we omitted the statement

```
count = count + 1 ;
```

from the loop body?

This is a very common error to make when using loops. If we omit the statement to increment count, the loop will never terminate, **as count will always be less than 5**.

Such an error is a **logical or runtime error**. These differ from syntax errors because the program can be executed but produces incorrect results. For this reason, they are a more serious error than syntax errors. In large programs, it is very difficult to ensure that there are no logical errors. Thorough testing of programs may increase our confidence that a program is correct, but such **testing on its own, can never establish the correctness of a program**. It is important to bear this fact in mind and it is worthwhile to investigate the area of program correctness.

Write a program to sum the integers 1 to 99 (i.e. calculate the sum of 1+2+3+...+99) and display the result.

```
# sum.py: calculate 1+2+3+.....+99
# Author: Joe Carthy
# Date: 01/10/2022

sum = 0               # contains the sum we wish to compute
i = 1                 # the loop counter

while i <= 99:
      sum = sum + i
      i = i + 1

print("Summation is: ", sum)
```

Executing this program produces as output:

```
Summation is: 4950
```

The loop body is executed only if the condition (i <= 99) evaluates to true. Since we have initialised i to 1, the condition evaluates to true and the loop body is executed.

In the loop body, a running total for sum is calculated by adding the value of i to sum. The variable sum is assigned the value sum + i. The variable i is then increased by 1.

We then test the condition again. The variable i now has the value 2 and the condition (i <= 99) remains true so we execute the loop body assigning sum the value 3 (1+2) and increasing i to 3. Next time around the loop, sum becomes 6 (3+3) and i becomes 4. We test the condition again and continue in this manner until i eventually reaches the value 100. When we test the condition in this case, it evaluates to false (i.e. i is greater than 99) and so the loop body is not executed. Instead we continue at the first statement after the loop body i.e. the print() statement.

Sometimes it is useful to put a print() in the loop body so you can see what's happening and also to get a better understanding of looping.

```
# sum2.py: calculate 1+2+3+.....+99
# Author: Joe Carthy
# Date: 01/10/2022

sum = 0               # contains the sum we wish to compute
i = 1                 # the loop counter

while i <= 99:
     sum = sum + i
     print("\nSum = ", sum, " i = ", i)
     i = i + 1

print("\n\nSummation is : ", sum, "\n\n")
```

Executing this program produces as output:

```
Sum = 1  i = 1

Sum = 3  i = 2

Sum = 6  i = 3

Sum = 10  i = 4
….
Summation is: 4950
```

Note: **If a condition evaluates to false before executing the loop body, the loop body will not be executed**. In this case, the while loop behaves like an if-then statement.

Programmers often use the short variable names i, j, k, ... as loop counters.

Variable Initialisation
In the last two examples it is crucial that the variables count and i are initialised to appropriate values for the loop to operate correctly. As a general programming principle, all variables should be initialised to appropriate values, usually at the beginning of a program.

How Many Loop Iterations ?
Frequently we will not know in advance, how many times to repeat a loop, so that we can not use the while loop in the manner presented above. Let's rewrite the calculator program to continue calculating for as long as the user requires. The user may wish to sum 1 pair of numbers or 100 pairs. The user indicates if they wish to finish by entering 0 as the first number. This type of loop is sometime referred to as a **non-deterministic loop**, as you do not know in advance how many times it will be repeated.

Example: Sum pairs of numbers until 0 entered as first number

```
# calc5.py: Calculator program to add 2 numbers until 0 entered
# Author: Joe Carthy
# Date: 01/10/2022

number1 = -1     # Any non-zero value will do
while number1 != 0
    number1 = float(input("\nEnter first number: "))
    number2 = float(input("\nEnter second number: "))
    sum = number1 + number2
    print("\n\nThe sum of", number1,"and",number2,"is",sum, "\n\n")
```

`calc5.py` outputs:

```
Enter first number: 1

Enter second number: 4

The sum of 1.0 and 4.0 is 5.0

Enter first number: 0

Enter second number: 8


The sum of 0.0 and 8.0 is 8.0
```

The program above is a poor one. It does stop after 0 has been input for the first number but it first reads the second number and adds it to 0 and displays that result. We want the program to stop after 0 has been entered as the first number:

In this example, we continue to execute the loop body as long as the user enters a non-zero value for number1. The loop body will always be executed once in this example. Why? Because the loop condition will always be true when the program begins execution, since number1 is initialised to be -1 at the start of the program. The program below is an improved version:

```
# calc6.py: Calculator program to add 2 numbers until 0 entered
# Author: Joe Carthy
# Date: 01/10/2022

number1 = -1        # Any non-zero value will do

while number1 != 0:
     number1 = float(input("\nEnter first number: "))
     if number1 != 0:
          number2 = float(input("\nEnter second number: "))
          sum = number1 + number2
          print("\n\nThe sum of", number1,"and",number2,"is",sum, "\n\n")

print("\n\nCalculator program terminated \n")
```

This program runs as follows:

```
% python3 calc6.py

Enter first number: 3

Enter second number: 3

The sum of 3.0 and 3.0 is 6.0


Enter first number: 0


Calculator program terminated
```

As you can see from the above program, you can use any statement in the loop body including more conditionals. This time, once we read the 1st number, we check if it is 0 and only if it is not 0, will we read the 2nd number and display the result.

Debugging with Loops

As mentioned earlier, if you have difficulty understanding loops, it is a good idea when you implement any of the above programs to put a print() statement in the loop body, so that you can see how often the loop is repeated. For example statements such as the following could be used in the examples presented earlier:

```
     print ("\ncount = ", count) ; #  display  value  of  count
each time around the loop
     print ("\nsum = ", sum ) ;    # display value of sum each
time around the loop
```

This is also a useful debugging technique. Debugging is the term used for finding and correcting errors (bugs) in your program. By placing print() statements in your code, you can

trace (follow) the execution of your program, inspecting the values of variables and checking if loops are executed the correct number of times. A print() in the action part of an if statement allows you verify that the action was indeed carried out. When your program is working correctly, these debugging print() statements are removed.

There are programs called **debuggers** which allow you execute your programs in an editor-like environment. They allow you to stop your program at any statement you wish (called a break point) and display the value of variables. You may even change the value of a variable and resume the execution of your program. They are very useful tools for programmers.

Nested Loops
A loop may contain as part of its loop body any statement including another loop (called an **inner loop** or **nested loop).** The nested loop may in turn contain a loop as part of its loop body and so on.

Write a program to display 4 lines with 1 star ('*') character on line 1; 2 stars on line 2, 3 stars on line 3 and 4 stars on line 4. The output should appear as follows:

```
*
**
***
****
.......
```

```
# tri.py: displays triangle composed of *'s

num_lines = 1

while num_lines <= 4:
    num_stars = 1
    while num_stars <= num_lines:     # inner loop
        print("*")
        num_stars = num_stars + 1     #end inner loop

    print("\n")               # start new line
    num_lines = num_lines + 1     # end outer loop
```

The inner loop displays the correct number of * characters on each line. The outer loop controls the number of lines displayed.

However, the above program does not work as intended. It displays the stars on new lines:

```
*

*
*


*
*
*


*
*
*
*
```

This is because the print() function adds the newline character at the end of the string. To stop print() doing this, we add a new argument called **end** to the function, as follows:

```
print("*", end = "") # instructs print not to output newline at end
```

If we use this version of print() in the above program we get the following output:

```
*

**

***

****
```

The range() function

This function returns a sequence of numbers in a given range for example

`range(6)` returns: `0,1,2,3,4,5  # integers up to but not including 6`

`range (4, 10)` returns `4, 5 , 6, 7, 8, 9`

`range (2, 12, 3)` returns 2, 5 ,8, 11   # from 2 up to 12 in steps of 3

We can write these 3 forms of range in a general form:

`range (stop)`	generate list from **0** to **stop**, ***not** including stop*
`range(start, stop)`	generate list from **start** to **stop**, **not** including stop
`range(start, stop, step)`	generate list from **start** to **stop**, **not** including stop, by increments of size **step**

To generate a list starting at 0 up to 7

```
range (8)  # yields 0, 1, 2, ,3 4, 5, 6, 7
```

To generate a list from 1 to 9:

```
range(1, 10) # yields 1, 2, ,3 4, 5, 6, 7, 8, 9
```

To generate list from 10 to 1

```
range(10, 0, -1) yields 10, 9, 8, … 2, 1
```

for Loop

There is another form of loop construct called the **for loop**. It is used when we know the number of times we wish to repeat the loop body. We often use the for loop to process a list of items in combination with the range() function.

The general form may be written as

```
for val in sequence:
     loop body statements
```

```
# print first 5 integers
# using python range() function

for i in range(5):
    print(i, end=" ")
print()

outputs: 0 1 2 3 4
```

```
# print numbers from 5 to 20

for i in range(5, 20):
    print(i, end=" ")

outputs: 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

```
# display range in steps of 2

for i in range(0, 10, 2):
    print(i, end=" ")
print()

outputs: 0 2 4 6 8
```

We can re-write the program to sum the integers 1 to 99 using a for loop as follows

```
# sum3.py: Sum 1 + 2 + 3 + ... +99

sum = 0          # contains the sum we wish to compute
for i in range(1, 100):
    sum = sum + i

print("\nSummation is:", sum, "\n")
```

In this case, variable i starts with value 1 which is added to sum, then I becomes 2 which is added to sum and so on until i becomes 99. Remember that range (1,100) generates the list from 1 to 99 – the stop vale of 100 is NOT included in the list.

Modify the program tri.py to use **for** loops, to display 4 lines with 1 star ('*') character on line 1; 2 stars on line 2, 3 stars on line 3 and 4 stars on line 4.

```
#!/usr/bin/python3
# tri3.py: displays triangle composed of *'s

num_lines = 1

for num_lines in range (1, 5):                    # outer loop
    for num_stars in range (1, num_lines+1):      # inner loop
        print("*", end = "")
    print("\n")                 # start new line
```

The inner loop displays num_lines stars on each line.

String Processing

We have used string variables and constants in our programs. In programming, we often wish to “process” the elements of a string in various ways. We will show how to access and process strings in the following examples. We need to know how long a string is when we are going to process it. The **len()** function gives us the length of a string e.g.

```
l =  len(“abcd”)
print (“l = “, l)
print(len(“123456”)

outputs
l = 4
6
```

Write a program to output the characters in a short string on separate lines.

```
# str.py: Output each characters on a newline

string = "abc"
length = len(string)

for i in range (0, length):
    print( string[i] )

print("\n\n")
```

outputs

```
a
b
c
```

Remember that strings always start at element 0.

Write a program to output the characters in a short string, separating them with the “&” character.

```
# str2.py: Output each characters of string followed by &

string = "abcdef"
length = len(string)

for i in range (0, length):
    print(string[i] + "&", end="")
print()
```

outputs

```
a&b&c&d&e&f&
```

Write a program to read a string and display it in reverse i.e. Joe is displayed as oeJ

```
# str3.py: Read a string and display it in reverse

string = input("Enter a string: ")
length = len(string)

for i in range (length-1, -1, -1):
    print(string[i],  end="")
print()
```

outputs

```
Enter a string: ABCDEF
FEDCBA
```

Pay particular attention to the range function used here.

Note the first element of a string in Python is **element 0**. In this example we entered a 6 character string. This means the elements are from 0 up to 5 (not 6). Thus to display the string backwards we need to range from element 5 to element 0. This is why we subtract 1 from length in the range function.

Also we need to include element 0 in the output, so we need to set the stop value in range to be -1 e.g. range (5, 0, -1) will give us elements 5,4,3,2, 1 but NOT element 0. To include 0 we need range(5, -1, -1) which means start at 5, step down by 1 each time and stop at 0.

Lists

We encounter lists in our daily lives such as

- shopping list of things to buy
- list of students in a class
- list of employees in a company

Programming languages provide us with a facility to handle lists. In some languages (C, C++, Java) we call them **arrays** but in Python we use lists.

We give the list a **name** and we can access the items in the list using an **index** (subscript):– `0, 10, 1, n, 22` and i are examples of an index in the lists below

- `shop_list[0], shop_list[10]`
- `student[1], student[n]`
- `employee[22], employee[i]`

Python is very flexible in what can be in a list – much more so than C or Java.

```
shop_list = ['bread', 'milk', 'coffee', 'sugar']

student = ['joe carthy', 'mary smith', 'tom', 'jack dunne']

grades = ['joe carthy', 'Maths', 60, 'Science', 70, 'History', 55]

employee = ['John Dun', 12, 40, 'Mary S', 10, 35, 'Jack Doyle', 10, 35]

li = []
```

The last example `li` is an empty list.

We access the elements of the list as follows using an index

`shop_list[0]` has value `'bread'`

`student[2]` has value `'tom'`

`grades[4]` has value `70`

`employee[7]` has value `10`

We can use a loop to process all of the items in a list as follows:

```
shop_list = ['bread', 'milk', 'coffee', 'sugar']

i = 0
while i < 4:
    print( shop_list[i] )
    i = i+ 1
```

outputs

```
bread
milk
coffee
sugar
```

The first element in a list is always element 0

```
grades = ['joe carthy', 'Maths', 60, 'Science', 70, 'History', 55]

print('Grades for: ', grades[0], 'are')
i = 1
while i < 6:
    print( grades[i], grades[i+1] )
    i = i+ 2
```

outputs

```
Grades for:  joe carthy are
Maths 60
Science 70
History 55
```

We can have lists in side list ! For example

```
shop_list = ['bread', 'milk', 'coffee', 'sugar']
sweets = ['chocolate', 'mints', 'gums']

new_list = [shop_list, sweets]
```

new_list has value: [['bread', 'milk', 'coffee', 'sugar'], ['chocolate', 'mints', 'gums']]

```
L1 = [1, 2, 3, 4]
L2 = [L1, 9,10]
```

```
L2 has value: [[[1, 2, 3, 4], 9,10]
```

We can use a for loop to process the otems in a list

```
for i in range(4):
     print('Element ', i, 'of list is: ', shop_list[i])
```

will output:

```
Element  0 of list is:  bread
Element  1 of list is:  milk
Element  2 of list is:  coffee
Element  3 of list is:  sugar
```

Lists are mutable – this means that you **can change individual elements** of the list e.g.

```
shop_list = ['bread', 'milk', 'coffee', 'sugar']

shop_list[1] = 'tea'
shop_list[3] = 'cake'
for i in range(4):
     print('Element ', i, 'of list is: ', shop_list[i])
```

will now output:

```
Element  0 of list is:  bread
Element  1 of list is:  tea
Element  2 of list is:  coffee
Element  3 of list is:  cake
```

You can see we have changed elements 1 and 3.

List comprehension

We often want to apply an operation to the elements of a list. Python provides what is called a **list comprehension** to do this. For example, create a list of the integers squared from 0 to 6:

```
L = [x ** 2 for x in range(7)]   # list comprehension
print('L is:', L)
```

produces the following output:

```
L is:  [0, 1, 4, 9, 16, 25, 36]
```

Sample Program using lists to count the individual digits 0 to 9 that a user enters.

```
# Program to use a list to count the number of different
digits entered
# Uses the number as an index into the list

#Initialise the counter list
countList = [0 for x in range(10)]

# Prompt the user for a digit

number = int(input('Enter a digit between 0 and 9: '))

while number >= 0 and number <= 9:
    countList[number] += 1
    number = int(input('Enter a digit between 0 and 9: '))


for i in range(10):    # display results
    print('Number of ', i, ': ', countList[i])
print('Finished!')
```

produces:

```
Enter a digit between 0 and 9:  1
Enter a digit between 0 and 9:  2
Enter a digit between 0 and 9:  3
Enter a digit between 0 and 9:  1
Enter a digit between 0 and 9:  6
Enter a digit between 0 and 9:  7
Enter a digit between 0 and 9:  3
Enter a digit between 0 and 9:  0
Enter a digit between 0 and 9:  1
Enter a digit between 0 and 9:  9
Enter a digit between 0 and 9:  8
Enter a digit between 0 and 9:  4

Number of 0: 1
Number of 1: 2
Number of 2: 1
Number of 3: 2
Number of 4: 1
Number of 5: 0
Number of 6: 1
Number of 7: 1
Number of 8: 1
Number of 9: 1
Finished!
```

We often want to initialise all elements of a list to a set value e.g. 0.

```
countList = [0 for x in range(4)]
```

can also be written as

```
countList = [0] * 4
```

`countList` in both cases is `[0, 0, 0, 0]`

Operations on Lists

`len (list)` returns the length of the list – the number of elements in it.

```
shop_list = ['bread', 'milk', 'coffee', 'sugar']
len( shop_list ) is 4
```

`list1 + list2` returns the concatenation of the two lists – the second list is added to the end of the first list

```
shop_list = ['bread', 'milk', 'coffee', 'sugar']
sweets = ['chocolate', 'mints', 'gums']
```

```
new = shop_list + sweets
```

new is `['bread', 'milk', 'coffee', 'sugar', 'chocolate', 'mints', 'gums']`

`n * list` returns a list that repeats `list` `n` times

```
L = [“abcd”]

2 * L is [“abcd”, “abcd”]
```

`list[start:end]` returns a slice of the `list` from position `start` to `end` but **NOT** including `list[end]`

```
L = [1, 2, 33, 4, 8, 6]

L[2:5] is [33, 4, 8]          # elements 2, 3, 4
```

`e in list` is `True` if `e` is contained in the `list` and `False` otherwise

new = `['bread', 'milk', 'coffee', 'sugar', 'chocolate', 'mints', 'gums']`

```
if 'bread' in new:
     print('Yes bread in list')
```

will print

```
Yes bread in list
```

```
if 'car' in new:
     print('Yes car in list')
else:
     print(‘Car not in list’)
```

will print

```
Car not in list
```

Methods associated with Lists

`List.append(e)` adds the object `e` to the end of the list `List`

```
List = [1, 2, 3, 1, 16]
```

`List.append(44)` adds the object `44` to the end of `List`

Now `List is [1, 2, 3, 16, 1, 44]`

`List.count(e)` returns the number of times that `e` occurs in `List`

`n = List.count(1)` returns the number of times `1` occurs in `List`

`n` is now `2` because `1` occurs twice in `List`

`List.insert``(i, e)` inserts the object `e` into `List` at index `i`

```
List = [1, 2, 3, 1, 16]
```

`List.insert(2, 99)` inserts the object `99` into `List` at index `2`

Now `List` is `[1, 2, `**`99`**`, 3, 16, 1, 44]`

`List.extend``(L1)` adds the items in list `L1` to the end of `List`

```
L = [1, 2, 3]
L1 = [‘a’, ‘b’, ‘c’ ]

L.extend(L1)
```

Now L is `[1, 2, 3, ‘a’, ‘b’, ‘c’]`

`List.remove(e)` deletes the first occurrence of `e` from `List`
(This method causes an error (raises an exception) if `e` is not in `List`)

```
L = [1, 2, 3, ‘a’, ‘b’, ‘c’]

L.remove(3)
```

Now L is `[1, 2, ‘a’, ‘b’, ‘c’]`

`List.index``(e)` returns the index of the first occurrence of `e` in `List`
(This method causes an error (raises an exception) if `e` is not in `List`)

```
L = [1, 2, 3]
i = L.index(3)
```

`i` is now 2 because 3 occurs at position 2

`List.pop(i)` ***removes and returns*** the item at index `i` in `List`
If `i` is omitted, it defaults to −1, to remove and return the last element of `List`

```
L = [1, 2, 3, 4, 6]

x = L.pop()
```

Now L is `[1, 2, 3, 4]`

`x` is 6

```
y = L.pop(0)
```

Now L is `[2, 3, 4]`

`y is 1`

`List.reverse``()` reverses the order of elements in `List`

```
L = [1, 2, 3]

L.reverse()
```

Now L is `[3, 2, 1]`

`List.sort``()` sorts the elements in in `List` in ascending order

```
L = [1, 222, 3, 45, 6]

L.sort()
```

Now L is `[1, 3, 6, 45, 222]`

More about Strings

We have already looked at strings but now look at some more functions/methods associated with strings.

len(str) returns the length of the string str

```
len("abcd") returns 4
```

s1 **+** s2 concatenates s2 onto end of s1

```
s1 = 'abc'
s2 = 'xyz'

s1 + s2 returns 'abcxyz'
```

n * str   returns a string that repeats str n times

```
str = ‘abcd ’

2 * str returns ‘abcd abcd abcd ‘
```

e in str   is `True` if `e` is contained in the `str` and `False` otherwise

```
str = 'bread gums blue black’

if 'bread' in str:
     print('Yes bread in string')
```

will print

```
Yes bread in string
```

```
if 'car' in str:
     print('Yes car in string')
else:
     print(‘Car not in string’)
```

will print

```
Car not in string
```

for x in str iterates over the string str

```
new = "ab ab cd"
for x in new:
      print(x)
```

```
will output the individual characters of the string
a
b

a
b

c
d
```

Methods on Strings

s.count(s1) returns the number of times that the string s1 occurs in s

s.find(s1) returns the index of the first occurrence of the substring s1 in s, and returns −1 if s1 does not occur in s

s.rfind(s1) the same as find, but starts from the end of s (the "r" in rfind stands for "reverse")

s = "ABBA"

t = **s.lower**() converts all uppercase letters in s to lowercase and stores them in t the string s is unchanged

t now contains abba

s = s.lower() converts all uppercase letters in s to lowercase

s now has value: abba

t = **s.upper**() converts all lowercase letters in s to uppercase and stores them in t

t now contains ABBA

s.replace(old, new) returns list with all occurrences of the string old in s **replaced** by the string new stores them in t

s = "ABBA abba"

t = s.replace('bb', 'xxxx')

t now contains ABBA axxxxa

```
s = s.replace(‘A’, ‘a’)
```

s now contains `aBBa abba`

`s.rstrip()` removes trailing whitespace from s

Whitespace refers to the *space* character, *tab* character, *newline, return* character and *formfeed* i.e. characters that you cannot see on the screen. Sometimes when you read a string from a file the newline character will be part of the string at the end and you may want to remove it.

`s.split(d)` splits string s using d as a delimiter and returns the list of substrings making up s

If d is omitted, the substrings are separated by arbitrary strings of whitespace characters.

This is a really useful and commonly used method

```
s = ’Joe, John, Bill, Mary’

L = s.split(‘,’)
```

We split the string using the comma character as delimiter

Now L is `[‘Joe’, ‘John’, ‘Bill’, ‘Mary’]`

```
s = ’Joe John Bill Mary’

L = s.split()
```

In the above example we split the string using the space character as delimiter

Now L is `[‘Joe’, ‘John’, ‘Bill’, ‘Mary’]`

Say we have a string made up of a Name, rate of pay and hours worked. We can break the string into its components:

```
s = ’Joe Bloggs 10.5 40’

L = s.split()
```

We can access the components in L

```
Name = L[0]

Rate_per_hour = float( L[1])

Hours_worked = float( L[2])
```

`Name` is now `‘Joe Bloggs’`

`Rate_per_hour` is now `10.5`

`Hours_worked` is now `40`

We can now process the components for example to calculate the pay for the employee.

Finally strings are **immutable** – this means that you **cannot** change the individual characters of a string e.g.

```
s=’AbbA’
```

You **cannot use**
s[0] = ‘b’

to overwrite character [0] in the string.

Write a guessing game program

```
# guess3.py: Guess the secret word
# Ignores case of words e.g. BLUE matches bluE


secret = "Blue"
guess = " "
num_chances = 1
secret = secret.lower()  # convert to lowercase
while (guess != secret) and ( num_chances <= 3 ) :
    guess = input("Guess the secret word:  ")
    guess = guess.lower()   # convert to lowercase
    if guess != secret:
        print("\nWrong guess: ", guess)
        num_chances = num_chances + 1
    else:
        print("Well done !")
if num_chances > 3:
    print("Sorry you have used all of your guesses")
    print("The secret word was: ", secret)
```

Running guess3.py:

```
Guess the secret word:  man
Wrong guess:  man
Guess the secret word:  dog
Wrong guess:  dog
Guess the secret word:  cat
Wrong guess:  cat
Sorry you have used all of your guesses
The secret word was:  blue
```

Running guess3.py:

```
Guess the secret word:  black
Wrong guess:  black
Guess the secret word:  BLUE
Well done !
```

File I/O

So far we have read input from the keyboard and displayed output on the screen. We will now look at using files in our programs. Every computer system uses files to store data. This allows information to be saved from one computation to another. Each operating system (eg Unix, Linux, Windows, MAC OS, Android, . . .) comes with its own file system. A file system has operations for creating, accessing, reading from, writing to and deleting files.

Accessing a file from within a Python program is done by using a **file handle**. Consider the Python statement:

```
fileHandle = open('junk.txt', 'w')
```

This invocation of the open function instructs the operating system to create a file with the name *junk.txt* and returns a file handle for that file that is bound to the variable `fileHandle`. We can use any variable name we wish e.g `fh`.

The second argument to the open function, "**w**", indicates that the file is opened for **writing**. This means that we wish to store information in the file - in programming terminology we **write** to the file.

If the file junk.txt already exists then any previous contents of the file will be **overwritten** -— take care not to destroy an existing file! If the file does not exist, it will be created.

We can open a file for **reading** which means we wish to read information from the file, using "**r**" in open().

When we are finished using a file in a program, we should close the file eg.

```
fileHandle.fclose()
```

We can only have a limited number of files open in a program at any time (sometimes around 20, depending on the operating system). By closing files when we are finished with them, a program can access 100's of files but not all at the same time.

We can write a string `address` to a file by:

```
fileHandle.write( address )
```

To read from a file, we must first call the `open` function with a second argument of "r", indicating that the file is opened for reading

```
fh1 = open('names.txt', 'r')
```

The function `readline()` reads a line from a file e.g.

```
line = fh1.readline()
```

`readline()` returns the empty string "" if the file is empty or when you have reached the end of the file i.e. there is no more data in the file.

It is good practice to make sure that a file exists before we open it for reading, because if the file does not exist the open function fails and your programme will display an error such as :
Traceback (most recent call last):
File "/home/john/Documents/dept/comp10280/2015 fh1 = open(filename, ’r’)
IOError: [Errno 2] No such file or directory:

One technique to check if a file exists is to use the function

```
os.path.isfile(filename)
```

This returns True if filename is an existing file and eturns False otherwise

We need to include the line **import os** to access this function e.g. the following code fragment prevents you from opening a file that does not exist.

```
import os

if not os.path.isfile(filename):
   print('File:' + filename + ' does not exist')
else:
   fh1 = open( filename, 'r’)
```

Terminating a Python script

There are times when you wish to terminate (quit, exit) a program immediately, for example, when a data file you need to access does not exist, then stopping your program is the sensible thing to do.

There are several ways to do this in Python but we will only use the `sys.exit()` function, which “tidies up” before quitting your program – this means that for example any output to the screen will be done before quitting and any open files will be closed. You need to import sys to use this function.

```
import os
import sys
……
….

if not os.path.isfile(filename):
   print('File:' + filename + ' does not exist\n')
   print('Terminating program \n')
   sys.exit()

else:
   fh1 = open( filename, 'r’)
```

Some File I/O sample programs.

Program to create a file with 3 lines of text. We ask the use to specify the name of the file to be created.

```
# create.py: Create file with some lines of text

fname = input("\nEnter filename to be created: ")

fout = open( fname, "w")        # Create new file

fout.write("Line 1 in the file\n")
fout.write("Line 2  in the file with more text\n")
fout.write("Line 3  Some more words and text 1 2 3 4 5 \n")
fout.close()                        # Close the file
```

Program to read and display the contents of a file specified by the user

```
# read.py: Read lines from the file created by create.py
# and prints them out

import os                       # Need this for path.isfile() function
import sys                      # Need this for sys.exit()

# Get name of file to be read

filename = input("\nEnter file name: ")

# Check whether the file exists

if not os.path.isfile(filename):
     print('File: ' + filename + ' does not exist')
     print('Quitting program')
     sys.exit()

else:
     fh1 = open( filename, 'r')

     line = fh1.readline()                # read 1st line from file
     while line != "":                    # "" means end of file reached
          print(line, end = "")
          line = fh1.readline()           # read next line from file

      fh1.close()
```

Program to read and display the contents of a file 10 lines at a time.

```
# display10.py: Display a file 10 lines at a time

import os
import sys

finput = input("\nEnter name of file to display: ")

if not os.path.isfile(finput):
     print('File: ' + finput + ' does not exist \n')
     print('\nQuitting ..\n')
     sys.exit()

fin = open( finput, "r")

linecount = 1
finished = ""

text = fin.readline()

while (text != "") and (finished != 'q'):
     print( text, end = "" )
     linecount = linecount + 1
     if linecount == 10:
          linecount = 1          # reset line count to 1 for next 10 lines
          finished = input("Enter q to quit or Press Return to continue  ")
     text = fin.readline()

fin.close()
```

This program opens the file specified by the if it exists. It then reads a line from the file and enters a loop:

```
while not at end of file and user has not entered q
     Print the line from the file
     Count number of lines printed
     If count == 10 then
          ask the user to quit or continue
          reset number of lines printed to 1
     read next line from the file
```

Program to count and display the number of uppercase, lowercase and digits in a file specified by the user

```
# wc.py: Count uppercase, lowercase and digits in a file

import os
import sys

fname = input("\nEnter filename: ")

if not os.path.isfile(fname):
     print('File: ' + fname + ' does not exist \n')
     print('\nQuitting ..\n')
     sys.exit()

fin = open(fname, "r")

line = fin.readline()            # Read 1st line

num_digits = 0                   # Number of digits
num_lc = 0                  # Number of lowercase letters
num_uc = 0                  # Number of uppercase letters

while line != "":           # while line not empty - not end of file
     for i in range(0, len(line)):
          if line[i] >= "0" and line[i] <= "9":   # count the digits
               num_digits = num_digits + 1
          elif line[i] >= "A" and line[i] <= "Z":
               num_uc = num_uc + 1
          elif line[i] >= "a" and line[i] <= "z":
                         num_lc = num_lc + 1

     line = fin.readline()       # read next line from file

fin.close()

print("\nThe file ", fname, "contains: ")
print("\nUppercase letters: ",num_uc)
print("\nLowercase letters: ",num_lc)
print("\nDigits: ",num_digits, "\n\n")
```

Write a program to make a **copy** of a file, specified by the user.

```
# copy.py: Make a copy of an existing file

import os
import sys
finput = input("\nEnter name of file to be copied: ")

if not os.path.isfile(finput):
     print('File: ' + finput + ' does not exist \n')
     print('\nQuitting ..\n')
     sys.exit()

fin = open( finput, "r")

foutput = input("\nEnter name of new file: ")
fout = open( foutput, "w")      # Create new file

text = fin.readline()

while text != "":
     fout.writelines( text )
     text = fin.readline()

fout.close()                # Close the files
fin.close()

print("\nFile ", finput, " copied to ", foutput, " \n")
```

Using a text editor create a “telephone directory” text file called “tel.dat” with entries of the form

```
Joe Bloggs      087 6767676767
Fred Smith      085 567812345678
Mary Anyone     085 12345657789
```

[**Text editor:**
A text editor produces a text file e.g. Notepad (Windows), Textedit or Gvim (Mac) (if using MS Word, save the file as a text file not a “.doc” file).]

Write a Python program called **tel** to search the file for any text in the file.

Usage:

```
    $ python3 tel
What are you searching for or Press Enter to quit:  joe
```

Output:

```
    Joe Bloggs  087 6767676767
     $
```

or

```
    $ python3 tel

What are you searching for or Press Enter to quit:  085
```

Output:

```
    Fred Smith  085 567812345678
    Mary Anyone    085 12345657789

    $ python3 tel

What are you searching for or Press Enter to quit:  xxx
```

Output:

```
    xxx not found in file


    $ python3 tel

What are you searching for or Press Enter to quit:  j
```

Output:

```
    Joe Bloggs   087 6767676767
```

```
# tel.py: Search list for what user is looking for

import os
import sys

fname = "tel.dat"

if not os.path.isfile(fname):
    print('File: ' + fname + ' does not exist')
    print('Quitting ...\n')
    sys.exit()

fh1 = open( fname, "r")   # Open data file
inline = fh1.readline()

search = input("\nEnter text you are searching for or Press Enter to quit:")
search = search.lower()     # convert to lowercase

while inline != "":
    line = inline.lower()     # convert to lowercase
    if line.find(search) != -1:     #if search text in current line
        print(inline)
    inline = fh1.readline()

fh1.close()                       # Close the file
```

The `tel.py` program first checks if the data file "tel.dat" exists and quits if the file does not exist. Otherwise it opens the file and reads the first line.

It then askes the user to enter the search text or Enter to quit. It converts the search text to lowercase.

```
It then goes into a loop:
    Converts the line form the file to lowercase
    Compares the line with the search text (both are in lowercase).
    If they match, it prints out the line from the file (inline) **in its original case**
    Reads next line from the file
When all lines have been read from the file, the program terminates.
```

The version of `tel.py` above only allows the user to search for one string before quitting. We now rewrite the program to allow the user to continue searching until they decide to quit.

To do this, we add an outer loop:

```
While user has not pressed Enter
    Go back to the start of the file
     ask the user to enter the search text or Enter to quit
    Go into inner loop until end of file reached:
        Converts the line form the file to lowercase
        Compares the line with the search text (both are in lowercase).
        If they match, it prints out the line from the file (inline) **in its original case**
        Reads next line from the file
    Check if user text was found in file and print message if it was not found
```

We use the method (function) **`seek(0)`** to go back to the start of the file.

The call

```
fh1.seek(0)
```

brings us to the start of the file that `fh1` is associated with. When you read from a file, the operating system remembers where you finished reading. Your next read will start from that position. In our program, we read to the end of the file in the inner loop. When we wish to start a new search then we must go back to the start of the file. The `seek(0)` function tells the operating system to do this.

```
# tel2.py: Search list for what user is looking for

import os
import sys

fname = "tel.dat"

if not os.path.isfile(fname):
     print('File: ' + fname + ' does not exist')
     print('Quitting ...\n')
     sys.exit()

fh1 = open( fname, "r")   # Open data file
search = " "
while search != "":
    found = False
    fh1.seek(0)                  # Go to start of file
    inline = fh1.readline()      # read 1st line from file

    search = input("\nEnter text you are searching for or Press Enter to quit:  ")
    lower_search = search.lower()     # convert to lowercase

    while (inline != "") and (lower_search != ""):    # Search file
        line = inline.lower()                         # Convert lowercase
        if line.find(lower_search) != -1:             # if text in current line
            print(inline)
            found = True
        inline = fh1.readline()                      # Read next line from file
                                                     # & end of inner loop

    if ( inline == "") and (found == False):
        print("\n", search, "not found in file\n")   # end of outer loop

fh1.close()                                           # Close the file
```

The above program is inefficient in that it reads the entire file for every search. Accessing a file on disk is very slow compared to accessing the same information in the computer's memory. We can make the program more efficient by read all the lines of the file into a list of lines – we only need to do this once. We then search this list for the information as often as we wish. Since the list is in the computer's memory, it is more efficient. However, for small files you will not notice any difference in performance as computers are very fast! We now implement a version of the program above with lists.

```
# tel3.py: Search list for what user is looking for using lists

import os
import sys

fname = "tel.dat"

if not os.path.isfile(fname):
     print('File: ' + fname + ' does not exist')
     print('Quitting ...\n')
     sys.exit()

fh1 = open( fname, "r")          # Open data file
numlines = 0                     # number of lines in the file

# Read file into list
list = []

inline = fh1.readline()          # read 1st line from file
while inline != "":
     list.append(inline)              # add line to end of list
     inline = fh1.readline()          # read next line
     numlines = numlines + 1          # count lines

# list now contains all lines from the file

fh1.close()                           # close file

search = input("\nEnter text you are searching for or Press Enter to quit:  ")
lower_search = search.lower()     # convert to lowercase

while search != "":
    found = False
    i = 0                        # index into list - start at list[0]

    while ( numlines > i ) and (lower_search != ""):
        line = list[i].lower()
        if line.find(lower_search) != -1:
            print(list[i])
            found = True
        i = i + 1
    if ( numlines == i ) and (found == False):
        print("\n", search, "not found in file\n")

    search = input("\nEnter text you are searching for or Press Enter to quit: ")
    lower_search = search.lower()     # end of outer loop
```

Notes: We start with an empty list - `list = []` and append on to the end of the list each line we read from the file using `list.append(inline).` When we have read all lines from the file, `numlines` will record the number of lines in the file. We now close the file and search `list` for the user entries. The inner loop uses `numlines` to detect when it has reached the end of the list.

In the above program, instead of counting the lines in the file in the first while loop, after we have read in the list, we could use the `len` function compute the length of the list which is the number of lines in the file:

```
numlines = len( list )
```

Functions

Programming languages provide a facility to break large tasks into smaller ones. This is done by using *subprograms.* Programming languages provide the programmer with the ability to define and use subprograms and different languages use different names for subprograms such as *subroutines*, *procedures* and *functions*. Python provides only one kind of subprogram - ***function***.

Functions allow us break our programs into smaller more manageable units. They are fundamental to the development of programs longer than a few dozen statements. A function is simply a facility for giving a name to a group of one or more statements.

Functions are **defined once**, but can be **called** (used, **invoked**) as **often as desired**. We have already used functions extensively. The statements used for I/O e.g. `print, input, readline write` are all examples of function usage. These functions have been predefined so that we do not have to define them in our programs. They are part of a functions library that is available when programming in Python.

Functions make our programming task easier for two main reasons. Firstly they allow us to reuse the same group of statements many times by referring to them by name rather than repeating the code. From our programs to date, it can be seen that functions such as `print()` and `input()` are used very frequently.

Secondly they make our programs easier to read and understand. This is because the *name of the function usually describes the purpose of the statements making up the function*. Thus, even, if we only use a function once in a program, it is useful from a documentation viewpoint, to make the program easier to read. A **meaningful name** should be used for each function. Documentation refers to the comments, variable names and function names we use in our programs. By using comments that explain what the program is doing and by choosing meaningful variable and function names – we are documenting our program.

Variables may be declared inside functions and are said to be **local** variables i.e. they are separate to variables with the same name used outside the function..

In order to pass information to functions we use **parameters**. These are the values inside the parentheses when the function is called. For example, take the statement

```
        print(f“{feet} feet = {inches} inches“);
```

In this case, the `print()` takes three parameters: a string and two variables are passed as parameters.

In the function definition, we can use any name for the parameter, which is called a **formal** parameter. When we call the function, we pass an **actual** parameter (argument) to it. The value of the **actual parameter** is processed by the function.

We do not have to pass parameters to a function. As an example, we could define a function called `newline()` which outputs the `newline` character as follows:

```
def newline():      #No parameters required
    print(“\n“)
```

This function requires no parameters. Such a function will always carry out the same task.

The use of parameters allows us to vary the task a function carries out. So we could rewrite `newline()` to take a number as a parameter which specifies how many newline characters to output:

```
def newline( n ):      # Output newline character n times
    for in range( n )
        print(“\n”, end = “”)
```

We call the function wherever wish on our program after we have defined it:

```
    newline(1)      # print 1 blank line

    x = 10

    newline(x)      # print x blank lines
```

You may define as many functions as you wish. Usually the code of the functions is included in the same file as the program that calls them and they must be defined in the program before you can call them.

Example : Complete program that defines and calls `newline().`

```
# call.py: calls the function newline()

def newline( n ):            # Output newline character n times
    for in range( n )
        print(“\n”, end = “”)


    print(“This program calls the newline() function“)
    newline(2)
    print(“This appears two lines after the first message above”);
```

This program produces as output:

```
This program calls the newline() subprogram
This appears two lines after the first message above
```

It is possible to store functions in separate files to the one used for the main program. For example, the functions `input(), open(), print()` and so on are stored in a file referred to as a function library or simply library. When you run Python programs you can access these function be default. We will see later that you can create your own library (module) of Python functions that you define.

Formal Arguments/Formal Parameters.
These are the names used inside a function to refer to its parameters or arguments e.g. `n` in the def `newline( n )` function definition

Actual Arguments/Actual Parameters.
These are the names or values used as arguments when the function is actually called; in other words, the values that the **formal arguments** will have on entry to the function e.g. the `2` or the `x` in the call below are actual arguments

```
      newline(4)
      newline(x)
```

The function `power` takes two arguments, `f` and `p` and computes `f` to the power `p`

```
def power(f, p):
      res = 1;
      for i in range( p+1):
            result = result *f

      return result

x = 10
y = 3

print(f'2 to power 4 = {power(2, 4)}\n')
print(f'x: {x} to power y: {y} = {power(x, y)}\n')
```

Outputs:

```
2 to power 4 = 16
x: 10 to power y: 3 = 1000
```

The **return** statement

The `return` statement causes a value to be returned from the current function to its caller. It is possible to omit return in a function in which case the function terminates "falling through" the last statement. In this case, an value **None** will be returned e.g.

```
print('Before calling newline()')

x = newline(2)

print(f'x = {x} \n')

outputs

Before calling newline()


x = None
```

The form of the return statement is as follows:

```
        return  expression
```

The *expression* is optional; if it is omitted, the value None is returned.

Top-Down Programming

In solving this program we have used a particular **programming methodology** called **top-down programming** or **stepwise refinement (divide and conquer).**

This methodology advocates breaking your problem into smaller problems (subproblems). Then you take each subproblem independently and further refine it to smaller steps. The process continues until you cannot refine the subproblems further. You then combine the final refinements to give an entire solution. You should then check the entire solution to see that it makes sense. Then you work through this solution with some sample data to test that it will work. Finally, you translate your solution into the language of your choice (Python, C++, Java, etc). Thus we speak of programming **into** a language and **not** programming **in** a language !

It is critical that before attempting to refine your problem, you **understand the problem fully**.

You should be able to specify precisely what inputs you expect and what outputs are to be produced.

You should also understand what processing is required to transform the inputs to give the required outputs.

At this stage you may begin your stepwise refinement process. This is made considerably easier if your problem specification is clear and unambiguous. Major problems arise in practice due to poor problem specifications and misunderstandings between the programmer and the problem specifier.

Functions and Variables
The declaration of variables inside function brings up an interesting issue. How are we to distinguish between variables declared inside a function from those declared inside other functions and the rest of the program. What happens if we use the same name for a variable in different functions. This is called a **name conflict.** A simple rule is used to avoid name conflicts.

Variables declared inside a subprogram can only be accessed inside that subprogram. We say that they are **local** to the function and call them **local variables.**

Technically we say that the **scope** of the variable is the function in which it is declared.

Scope rules allow us determine what variables can be accessed at any point in a program.

Another important point regarding functions is that when a function finishes execution, its local variables effectively disappear. Local variables only exist while the function where they are declared is executing. Each time the function is executed they come into existence and they cease to exist when the function terminates.

Examples
Write a program to allow the user check if a car registration is in a list of stolen cars.

The list is read from a file which has the car registration number and the owner name for a number of stolen cars.

The programs checks if a registration number or an owner name is on the list. This is basically the same as the `tel` program we have presented earlier.

The list of stolen cars is in a file `stolen.dat` which has the form:

```
2020WXY1976  Joe Carthy
2012DNK7768  Bill Jones
2023Ky1024 Mary Smith
2020tn123 Jack Jones
anyreg Joe Bloggs
```

Firstly we present the program without using functions:

```
# stolen.py: Search a file of stolen cars

fh1 = open('stolen.dat', 'r')

# Read names and car reg from file into a list

list = []
inline = fh1.readline()       # read 1st line from file

while inline != "":
    list.append(inline)           # add line to end of list
    inline = fh1.readline()

# list now contains all lines from the file

fh1.close()

num_entries = len( list )


search = input("\nEnter Car Reg Number or Press Enter to quit:  ")
lower_search = search.lower()     # convert to lowercase

while search != "":
    found = False
    i = 0                # index into list - start at list[0]

    while ( num_entries > i ) and (lower_search != ""):
        line = list[i].lower
        if line.find(lower_search) != -1:
            print(list[i])
            found = True
        i = i + 1

    if ( num_entries == i ) and (found == False):
        print("\n", search, "not found in file\n")

    search = input("\nEnter Car Reg Number or Press Enter to quit:  ")
    lower_search = search.lower()     # end of outer loop

print('\n\nFinished \n\n')
```

We now present version 2 of the program using functions to:

Check that we can open the data file
Read the list of entries from the file into a list
Search the list for an entry

```
# stolen2.py: Search a file of stolen cars
import os
import sys

# open_and_quit(): function to check if file cannot be opened and
# quit with error message otherwise return the file handle

def open_and_quit( fname ):

    if not os.path.isfile(fname):
        sys.exit('File: ' + fname + ' does not exist')
    else:
        fh1 = open( fname, "r")   # Open data file

        return fh1

def read_list_of_registrations( filename ):

    fh1 = open_and_quit( filename )

    # Read names and car reg from file into a list
    inline = fh1.readline()       # read 1st line from file

    while inline != "":
        list.append(inline)           # add line to end of list
        inline = fh1.readline()

        # list now contains all lines from the file

    fh1.close()
    return           # end of function


def search_list( list ):

    num_entries = len( list )
    search = input("\nEnter Car Reg Number or Enter to quit:  ")
    lower_search = search.lower()
    while search != "":
        found = False
        i = 0

        while ( num_entries > i ) and (lower_search !=
            line = list[i].lower
            if line.find(lower_search) != -1
                print('\n', list[i])
                found = True
            i = i + 1

        if ( num_entries == i ) and (found == False):
            print("\n", search, "not found in file\n")

        search = input("\nEnter Car Reg Number or Enter to quit:  ")
        lower_search = search.lower()     # end of outer loop
    return
```

```
# main program

list = []                        # Create list to store entries

read_list_of_registrations( 'stolen.dat' )

search_list (list )

print('\n\nFinished \n\n')
```

The above program shows how functions can be used.

Note the line:

```
sys.exit('File: ' + fname + ' does not exist')
```

The `sys.exit()` function closes any open files and terminates your program but displays the string passed as a parameter before the program quits. So if the file `stolen.dat` did not exist the program would display

```
    File: stolen.dat does not exist
```

and the program would terminate,

Running stolen2.py produces the following outputs:

```
% python3 stolen2.py
Enter Car Reg Number or Enter to quit:  xxx

 xxx not found in file


Enter Car Reg Number or Press Enter to quit:  joe

 2020WXY1976  Joe Carthy


 anyreg Joe Bloggs


Enter Car Reg Number or Enter to quit:


Finished
```

Creating your own function library or module

You can define your functions and store them in a file e.g. we could store the functions `read_list_of_registrations(),search_list()` and `open_and_quit()` in a file called `car.py` separate from the program file stolen2.py.

The file car.py is a **module** (library) of functions which we can now use in any program we write.

To use any of these functions in a program, we **`import`** the module into the program file where we wish to use the functions.

In the program where we wish to use the functions, we call them by putting the module name, in this case, "`car.`" as the first part of the function name i.e. `car.search_list()`.

So our main program stolen4.py has the form below:

```
# stolen4.py: Search a file of stolen cars

import car           # allows us use functions from car.py

# main program

list = []                     # Create list to store entries

car.read_list_of_registrations( 'stolen.dat', list )

car.search_list (list )

print('\n\nFinished \n\n')
```

The file `car.py` will contain the code of the functions as we used earlier but we need to add one parameter to the `read_list_of_registrations` for the `list.`

```
# car.py: functions for stolen cars program

import os
import sys

def open_and_quit( fname ):

    code of the function as above

def read_list_of_registrations( filename, list ):

    code of the function as above but with added parameter list

def search_list( list ):

    code of the function as above
```

Conclusion

We have now covered the main topics in programming that allow you write an infinite variety of useful programmes. The code written to put the first men on the moon did not have many of the useful features that have been covered!

Having said that, there is still a lot to be learned. The best way to learn is to practice, to read other people's code and of course read books and web articles on programming.

There is no substitute for your own practice. Practice writing short programs. Develop your own library of functions that you can use in your programs.

Keep going and good luck !!

Joe Carthy