POLYTECHNIC UNIVERSITY OF THE PHILIPPINES

Computer Engineering
College of Engineering
Mabini Campus, Sta. Mesa

CMPE 102

Programming Logic and Design

Engr. Julius S. Cansino

Programming Logic and Design

CMPE 102

Engr. Julius S. Cansino

Published and distributed by:

Polytechnic University of the Philippines

Address : 1016 Anonas St., Brgy. 630, Sampaloc, Sta. Mesa, Manila
Website : https://www.pup.edu.ph
Email Address : inquire@pup.edu.ph
Tel. No. **:** (+63 2) 5335-1PUP (5335-1787) or 5335-1777

The VMPGO

VISION

A Leading Comprehensive Polytechnic University in Asia

MISSION

Advance an inclusive, equitable, and globally relevant polytechnic education towards national development.

PHILOSOPHY

As a state university, the Polytechnic University of the Philippines believes that:

- Education is an instrument for the development of the citizenry and for the enhancement of nation-building; and,
- That meaningful growth and transformation of the country are best achieved in an atmosphere of brotherhood, peace, freedom, justice and nationalist-oriented education imbued with the spirit of humanist internationalism.

SHARED VALUES AND PRINCIPLES

1. Integrity and Accountability
2. Nationalism
3. Sense of Service
4. Passion for Learning and Innovation
5. Inclusivity
6. Respect for Human Rights and the Environment
7. Excellence
8. Democracy

GOALS OF THE COLLEGE/CAMPUS

1. Provide quality education through instruction, advance research and extension services
2. Produce world-class professionals as potential industry leaders and job providers
3. Develop and produce facilities through the use of adapted technology and indigenous materials
4. Maintain, upgrade or improve facilities through the applications of engineering technology

PROGRAM DESCRIPTION

Computer Engineering is a four-year degree program that deals with the study of computer systems. The curriculum covers both software and hardware and develops the student's ability to analyze computer systems, designs, construction of electronic equipment and its peripherals. Since computer science is directed to the theory and technology of computation, the curriculum does not specialize along traditional lines that divide hardware and software, systems and applications, or theory and experiment. Rather, a unified approach to the design and analysis of computers and of computing structures is employed. This background prepares the student for placements as computer engineers in government industry. It also qualifies them for related job with computer manufacturers and consulting firms as systems programmers as well as application programmers with scientific, research, and business organizations. Ethical considerations with respect to the profession is an important component of the program of study.

COURSE DESCRIPTION

This is an introductory course in computer programming logic. The student will learn algorithms applicable to all programming languages, including identifiers, data types, arrays, control structures, modular programming, generating reports, and computer memory concepts. The student will learn to use charts commonly used in business and information processing. Program logic will be developed using flowcharts and pseudo code. Programs will be written using any programming Language.

INSTITUTIONAL LEARNING OUTCOMES (ILOS)

A PUP graduate imbibed:

1. Creative and Critical Thinking
2. Adeptness in the Responsible Use of Technology
3. Strong Service Orientation
4. High Level of Leadership and Organizational Skills
5. Community Engagement
6. Effective Communication
7. Sense of Personal and Professional Ethics
8. Passion to Life-long Learning
9. Sense of National and Global Responsiveness

PROGRAM LEARNING OUTCOMES (PLOS)

1. Employed in the field of Network Engineering or Big Data or Machine Learning or System Development or other related fields

2. Committed members of professional and other allied organizations engaged in community development and nation building

3. Involved in continuous training and development in current or emerging trends, and advancement in their chosen field of specialization

COURSE LEARNING OUTCOMES (CLOS)

1. Construct structured programs using modular approach.
2. Apply decision making, repeating instructions and array manipulation to solve programming problems using pseudocode and flowchart.
3. Create structured programs in file handling and advanced techniques on modularization and data handling.
4. Create functions to modularize the approach in solving a problem.

Preface

Welcome to the world of programming! This manual is designed to be your comprehensive guide to understanding the fundamental concepts of programming logic and design. Whether you're a beginner just starting out or someone looking to refresh your skills, this manual will provide you with the foundational knowledge you need to succeed in the ever-evolving field of software development.

The primary goal of this manual is to teach you how to think like a programmer. Programming is not just about writing code; it's about solving problems in a logical, structured way. This manual will help you develop the skills to analyze problems, design solutions, and implement those solutions in code.

This manual is structured to build your knowledge progressively. Each chapter introduces new concepts, with plenty of examples and exercises to reinforce your understanding. By the end of this manual, you will have the tools to design efficient, effective programs and the confidence to tackle more advanced programming topics.

TABLE OF CONTENTS

Title Page	1
The VMPGO	3
Preface	7
Table of Contents	8
List of Figures	11
Course Guide	17
Lesson 1 Getting Started	
a. Introduction	18
b. Learning Objectives	19
c. Lecture Discussion	19
d. Lesson 1: Our First Program	22
e. Lesson 2: A poem!	29
f. Exercises 1	34
Lesson 2 Functions (Part I)	
a. Introduction	37
b. Learning Objectives	41
c. Lecture Discussion	42
d. Developing modular code using functions	42
e. A lesson in design – Green Eggs and Ham!	47
f. Exercises 1	43
g. Exercises 2	49
h. Exercises 3	58
i. Summary / Key Points	62
Lesson 3 Variables, Assignments and Expressions	
a. Introduction	63
b. Learning Objectives	73
c. Lecture Discussion	74
d. Basic Arithmetic	74
e. Reading numbers from the end-user	84
f. Exercises 1	68
g. Exercises 2	77
h. Exercises 3	83
i. Exercises 4	90
j. Appendix	93

Lesson 4 Functions (Part II)

a. Learning Objectives 94
b. Lecture Discussion 94
c. Using variables as arguments 106
d. Function Return Values 115
e. The math library 124
f. Encapsulating code in functions 127
g. Design using parameters and return values 128
h. Exercises 1 101
i. Exercises 2 110
j. Exercises 3 119
k. Chapter Exercises 129

Lesson 5 Introduction to Conditions & Decisions

a. Learning Objectives 133
b. Lecture Discussion 133
c. Exercises 1 136
d. Relational Operators 139
e. Class Activity 141
f. Syntax check 143
g. Exercise 2(single option if) 146
h. Double Option if statement (if-else) 148
i. Class Activity 149
j. Exercises 3 (if-else statement) 157
k. Multiple Option(if-elif statement) 158
l. Forming your own conditions 160
m. Syntax and Semantics 161
n. Exercises 4(if-elif statement) 169
o. Chapter Exercises (Decisions)

Lesson 6 Introduction to Loops

a. Learning Objectives 172
b. Lecture Discussion 172
c. Class Activity – counting from 1 to 10 176
d. More on the loop guard 177
e. Exercise 1 178
f. Never ending loops (Infinite Loops) 179
g. Loops that do nothing 180
h. Counter controlled repetition 181
Exercise 2 182
j. Using variables inside loops 183
k. Exercise 3 186
l. Accumulating Totals 192
m. Exercise 4

n. Indefinite Loops and Sentinels 193
o. Exercise 5 195
p. Using loops to validate data 196
q. Exercise 6 197
r. Making decisions inside loops 198
s. Calling functions from loops 200
t. Exercises 7 202
u. For loops 204
v. The for loop – syntax and semantics 205
w. for vs. while loops 206
x. Exercises 8 (for loops) 209
y. Nested loops 211
z. Exercises 9 – nested loops 215

Lesson 7 String Processing

a. Introduction 219
b. Learning Objectives 220
c. String addition and multiplication 220
d. Exercises 1 (introduction) 223
e. String Indexing 224
f. Exercises 2 (indexing) 227
g. Slicing (substrings) 228
h. Exercises 3(String slicing) 230
i. String Comparisons 231
j. String Processing 232
k. Exercise 4 (String Processing) 239
l. Chapter Exercises 241

Lesson 8 Lists (Arrays)

a. Introduction 243
b. Learning Objectives 243
c. Lecture Discussion 243
d. List methods 247
e. List Aliasing and List Cloning 249
f. Passing lists as parameters 250
g. Tips to remember 251

LIST OF FIGURES

Pages

Lesson 1 Getting Started

Figure 1.1 Start PyScripter 22
Figure 1.2 Programmer's Blank Canvas! 23
Figure 1.3 print("Hello World") 23
Figure 1.4 Run the program 23
Figure 1.5 Save it! 24
Figure 1.6 NameError notice! 25
Figure 1.7 SyntaxError notice! 26
Figure 1.8 SyntaxError 2 notice! 26
Figure 1.9 IndentationError notice! 27
Figure 1.10 PyScripter's Screen 29
Figure 1.11 Code Area 30
Figure 1.12 Line Execution 31
Figure 1.13 Console Result 31
Figure 1.14 Second Line Execution 31
Figure 1.15 Second Line Result 31
Figure 1.16 Overall Result 32
Figure 1.17 Display the Poem! 32
Figure 1.18 Python's input() 33
Figure 1.19 Diamond Asterisks 34

Lesson 2 Functions (Part I)

Figure 2.1 displayPoem() 38
Figure 2.2 Call the Function 38
Figure 2.3 Function's behavior (Part 1) 39
Figure 2.4 Function's behavior (Part 2) 39
Figure 2.5 Result of the function 39
Figure 2.6 Other behavior of displayPoem() 39
Figure 2.7 Print the result! 40
Figure 2.8 Look again at the result 40
Figure 2.9 365 in asterisks 42
Figure 2.10 Print the 365 in asterisks 42
Figure 2.11 Create function for 365 42
Figure 2.12 Result of Green Eggs and Ham! 55
Figure 2.13 365 Solution 1 55
Figure 2.14 365 Solution 2 55
Figure 2.15 365 Solution 3 56
Figure 2.16 Names using hashtags 57

Figure 2.17 Display patterns! 57
Figure 2.18 drawZero() 58
Figure 2.19 Call drawZero() 58

Lesson 3 Variables, Assignments and Expressions

Figure 3.1 firstNumber – Variable 63
Figure 3.2 secondNumber – Variable 63
Figure 3.3 Initialize firstNumber/ secondNumber 64
Figure 3.4 Add them both! 64
Figure 3.5 Include the sum 64
Figure 3.6 Print with Arithmetic Operators 65
Figure 3.7 PROGRAM 1 66
Figure 3.8 PROGRAM 2 66
Figure 3.9 PROGRAM 3 66
Figure 3.10 PROGRAM 4 66
Figure 3.11 print(message) 66
Figure 3.12 Display the time 66
Figure 3.13 NameError notice! 67
Figure 3.14 Look and Learn! 75
Figure 3.15 Arithmetic Expression 81
Figure 3.16 Enter your name… 81
Figure 3.17 Joe! 82
Figure 3.18 Convert the temperature! 84
Figure 3.19 input() 84
Figure 3.20 Statement about temperature 85
Figure 3.21 float() 85
Figure 3.22 TypeError Notice! (Arithmetic Expressions) 85
Figure 3.23 Interest calculation 87
Figure 3.24 Declare 3 variables 87
Figure 3.25 Ask for the principal! 87
Figure 3.26 ‘x’ and ‘y’ variables 88
Figure 3.27 ‘r’ and ‘l’ variables 88
Figure 3.28 Work with pi! 89

Lesson 4 Functions (Part II)

Figure 4.1 Semantics of function call 95
Figure 4.2 def homework() 96
Figure 4.3 Re-using functions 96
Figure 4.4 About Jack… 97
Figure 4.5 Function Parameters 97

Lesson 5 Introduction to Conditions & Decisions

Figure 5.1 Guess the number 133
Figure 5.2 Flow chart illustration of if-statement 134
Figure 5.3 Modified code of the game 135
Figure 5.4 Is equal to… 139
Table 5.1 139
Table 5.2 Example 1 140
Table 5.3 Example 2 140
Figure 5.5 What is wrong in this code? 143
Figure 5.6 SyntaxError1 notice! 143
Figure 5.7 What is wrong in this code? (Part 2) 143
Figure 5.8 If-Else statement 146
Figure 5.9 Flow chart illustration of if-else statement 147
Figure 5.10 Three possibilities (If-Else Statement) 157
Figure 5.11 Re-arrange if-else statement orders 158
Table 5.4 Using pseudo-code 158
Figure 5.12 Flowchart of if-else statement 160

Lesson 6 Introduction to Loops

Figure 6.1 Print("Hello World") 172
Figure 6.2 "Hello World" in While Loop 173
Figure 6.3 How While Loop works… 173
Figure 6.4 Flow Diagram of While Loops 174
Figure 6.6 Terminating Loop 179
Figure 6.7 Infinite Loop 179
Figure 6.8 Loops that do nothing 179
Figure 6.9 182
Table 6.1 Escape Sequence 183

Lesson 7 String Processing

Figure 7.1 Adding 2 string variables 222
Figure 7.2 Adding more than 2 string variables 222
Figure 7.3 String indexing 225
Figure 7.4 Indexed using negative integers 225
Figure 7.5 Indexing out of range 226
Figure 7.6 Immutable strings 226
Figure 7.7 TypeError 'str' 226
Figure 7.8 Accessing substrings 228
Figure 7.9 len() function 228
Figure 7.10 len(str) 229

Lesson 8 Lists (Arrays)

Table 8.1 Concatenation 244
Table 8.2 Indexing 244
Table 8.3 Slicing 245
Table 8.4 Method call table 247
Figure 8.1 List methods 247
Figure 8.2 List Methods 248
Figure 8.3 Pick a card! 248
Figure 8.4 6-sided dice 249
Figure 8.5 List Aliasing 249
Figure 8.6 Lists as parameters 250

This page was intentionally left blank…..

This page was intentionally left blank…..

COURSE SYLLABUS

Lesson 1

Getting Started

INTRODUCTION

A **computer program** is a list of instructions that tell a computer what to do. Everything a computer does is done in response to the instructions contained in computer programs.

Computer programming is the activity of writing computer programs. Programs are written using a special language called a **programming language**. There are hundreds of different programming languages - examples include Java, C, C++ (pronounced C plus plus), Python, Ruby, COBOL and Visual Basic.

It is very important to bear in mind that the purpose of programs is to solve problems – specific problems such as a payroll, word processing, airline reservation, book lending (as used by libraries), accounting, games, Automated Teller Machine (as operated by banks), patient tracking (as used by hospitals), social networks (e.g. Facebook) and so on. These are all examples are of real world systems. The list of problems that have been solved by computer programs is long and varied.

In reality a computer system is made up of many computer programs. Each program does a specific task and together all these programs make up the system (i.e. the solution to the original problem).

Before a computer system can be written the problem to be solved must first be fully understood. This task of getting to understand a problem is called **system's analysis**. The exact details of what the system will do are decided during the analysis phase of software development. Once we know what the system will do, we then must decide how the system will work. This is called **system design**. It is during the design phase of a project that the data to be input into the system, screen layouts, data to be generated by the system, reports, database and data processing are all decided. Programming (i.e. coding) can begin when (and only when) the system has been fully designed.

Clearly, one must learn a programming language before they can write code. The best way to learn a language is to practice using it. Once a person has mastered a programming language, they can use that language to solve problems. In other words, the programming language is just a tool used to write computer systems that do something useful.

Learning a new programming language is not difficult. In fact, it's much easier to learn a programming language such as Python or Java than it is to learn a natural language such as French, Spanish or even English. For a start the vocabulary of a natural language is much, much larger than that of a programming language – most programming languages have fewer than 50 words. Compare this to the number of English words you know. When we construct sentences using the vocabulary of a natural languages there are certain rules we have to follow. These rules make up the **grammar** or **syntax** of the language. Every language including programming languages have their own syntax. One of the main differences between learning natural languages and programming languages it that once you have mastered one programming language it is easier to learn another. This is not to say that all programming languages are the same. However, there are more similarities between Python and Java for example, than there would be between English and Irish.

The main aim of this course is to provide the knowledge necessary to develop small computer programs. Hopefully, you will learn how to:

analyze problems
design solutions
implement (write code) solutions using the Python programming language
test these solutions

LEARNING OBJECTIVES

1. **Knowledge/Remembering:** The process of problem-solving involves identifying inputs, setting goals, and creating a list of tasks to achieve the objective.
2. **Comprehension/Understanding:** To apply problem-solving steps to real-world scenarios, identify the inputs, define the goals, list the tasks, and explain how to execute these tasks to reach the goal.
3. **Application/Applying:** Given a problem scenario, develop a step-by-step plan that includes identifying inputs, defining goals, creating a list of tasks, and executing these tasks to reach the goal.

LECTURE DISCUSSION

End-users' vs Programmers

The differences between end-users and programmers need to be clearly understood.

An **end-user** is the person (or organization) for whom a software system is developed. Certain software systems such as games, operating systems, office applications etc., are developed with the public in mind as the end-user. These systems are sold as '**off the shelf**' packages in computer stores. Other software systems – called **bespoke systems** are developed specifically for a client company.

It should be clear that there is no need for a typical end-user to understand how to write computer programs to solve problems. End-users with an understanding of computers, however, may be well positioned to explain their requirements to a computer programmer.

A programmer is a person that writes the code that makes up a computer system. Programmers are usually employed by **software houses,** i.e. a company whose primary activity is software development. The business of a software house in the production of software whereas the business of an end-user is related to the area which the software is aimed at e.g. banking, insurance, finance, travel, health etc. It is worth noting that some organizations (usually large) employ software professionals to develop their own **in house software**. In any case, programmers can be employed on a permanent or contract basis.

Programmers work as part of a team of software professionals working in jobs such as system analysts, system designers, system architects, technical writers, database administrators, team leaders, managers, and of course computer programmers. Software professionals all work together towards a common end goal of developing a computer system for an end user. The role of the computer programmer is to write the code to meet a particular design specification provided.

Most software professionals – including computer programmers – are educated to degree level - typically 3-4 years post leaving certificate in Ireland. Degree courses such as Computer Science, Software Development, Software Engineering and Computing all lead directly to employment in the software industry. There are many surveys that indicate higher than average job satisfaction rates and salaries within this employment sector.

Throughout this course you will have to alternate your role between that of programmer and end-user. As a programmer you will solve problems with Python code and as an end-user you will verify that the program works by using it.

Finally, it is worth providing a cautionary note before diving into the world of Python. As a novice programmer, it is necessary to build up experience by using Python to solve small problems that are sometimes academic in nature i.e. problems for the sake of learning.

Many of the examples and exercises contained in this text are for educational purposes. They are designed so that you can improve your problem-solving skills and learn certain aspects of Python. In reality, most commercial computer systems are complicated in nature and can take months and even years to write (so be patient – you're just beginning and here to learn!)

Getting Started

In order to get down to the business of *writing and testing your own Python programs* you first need to download and install two pieces of software:

The Python Programming Language

This is the software that runs your Python programs.

Download version 3.3.2 (32-bit) from https://www.python.org/downloads/

An Integrated Development Environment (IDE)

This is the software that you will use to type in and run your Python programs….

Download version 2.6 from https://sourceforge.net/projects/pyscripter/

Integrated Development Environments **(**IDEs) allow programmers to:

type in computer code (in whatever language they are using)
save their code as program files
open previously saved programs
compile their programs (this is not relevant for Python)
run, test and correct any errors in their computer programs
debug their computer programs.

IDEs themselves are software applications (tool) and as such there are many available – both free and to purchase - on the market. Some IDEs are designed to work with specific programming languages, and some are more general. Microsoft Visual Studio is an example of a general IDE. It is used by professional programmers to develop applications in Visual Basic, C and C++ among others. Eclipse and NetBeans are other examples of IDEs commonly used to develop Java applications.

Summary

Programmers write code. End-users use the system.

Programmers test their code by simulating the role of end-users.

Programmers understand a programming language. End-users do not need to understand how to write computer programs.

End-users come up with problems that sometimes can be solved by computer systems. Programmers develop these systems.

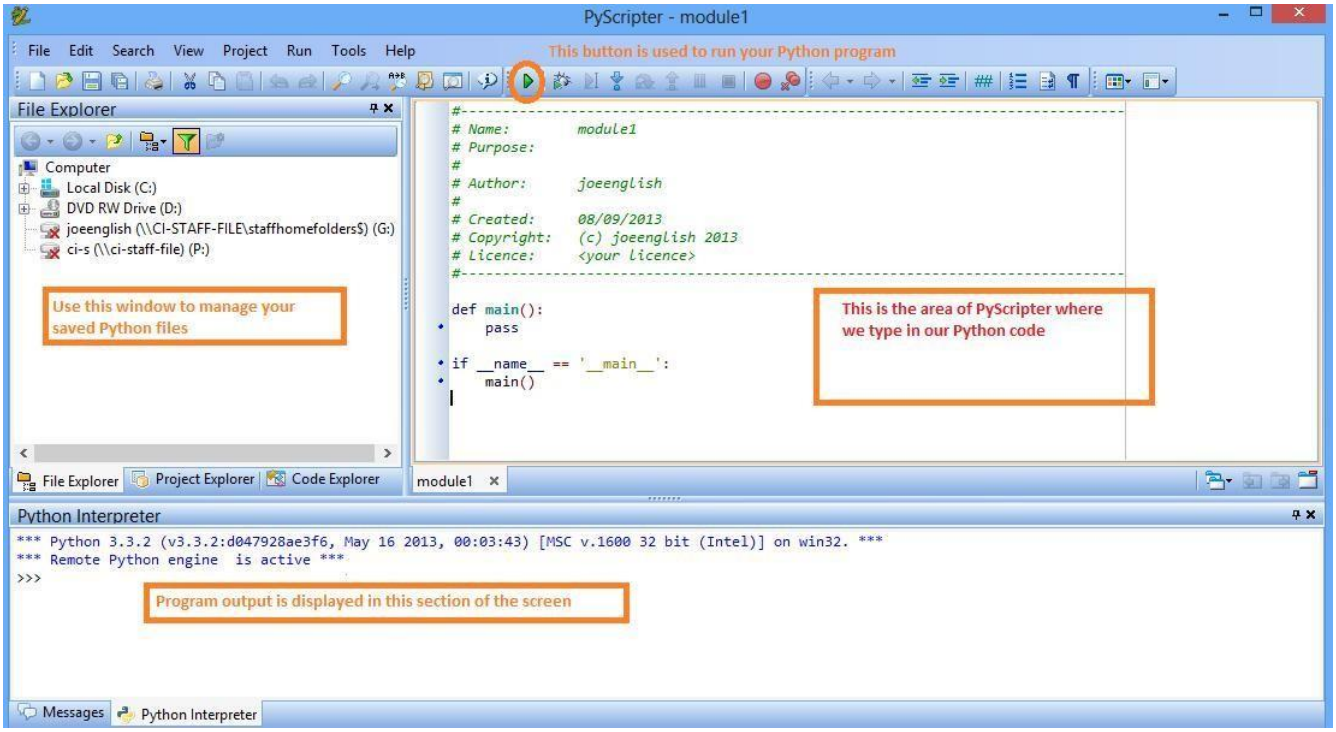
Programmers work for the end-users. This is because the end-user is the programmer's customer and therefore pay master.

Lesson 1: Our First Program

We will write, run and save a small program that displays the message *Hello World* on the screen.

The first thing we do is start PyScripter. The following screen is displayed - the most important parts of the window are highlighted.

Figure 1.1: Start PyScripter

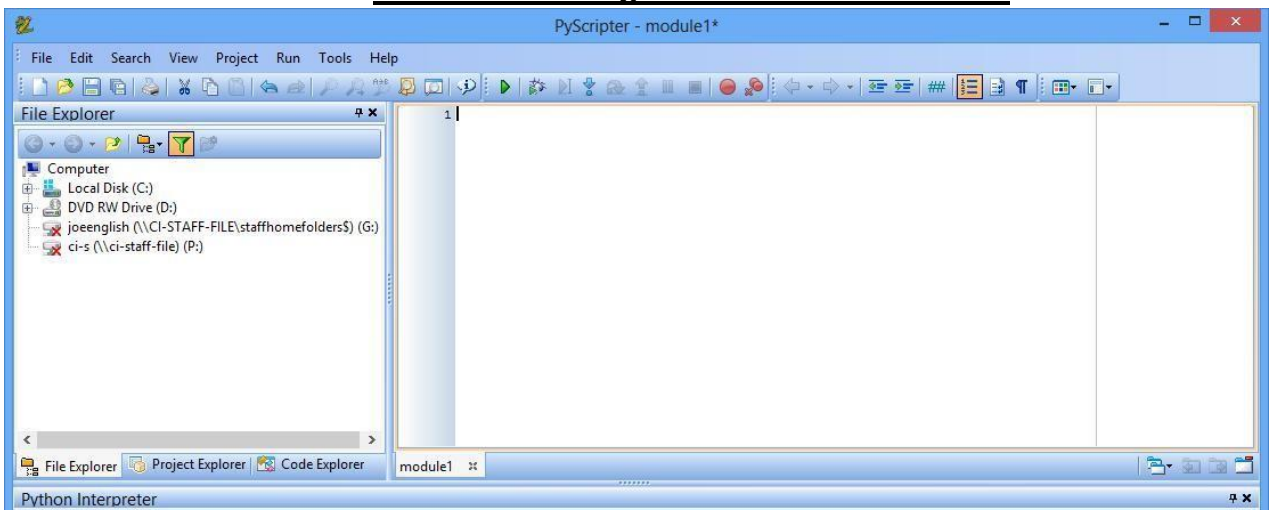


Before keying in any code I usually start off by clearing the code area. This can be done by using the 'Backspace' or 'Delete' keys but I usually press Ctrl+A and then Delete.

It is also useful to turn on line numbers. The option to do this is on the top line.

When you have the program area cleared and line numbers turned on the screen should look something like the one below – a programmer's blank canvas!

FIGURE 1.2: Programmer's Blank Canvas!



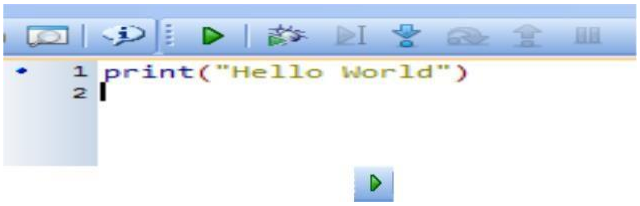
You are now ready to key in your first program.

At line 1 type the following Python statement **exactly as it appears here**.

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

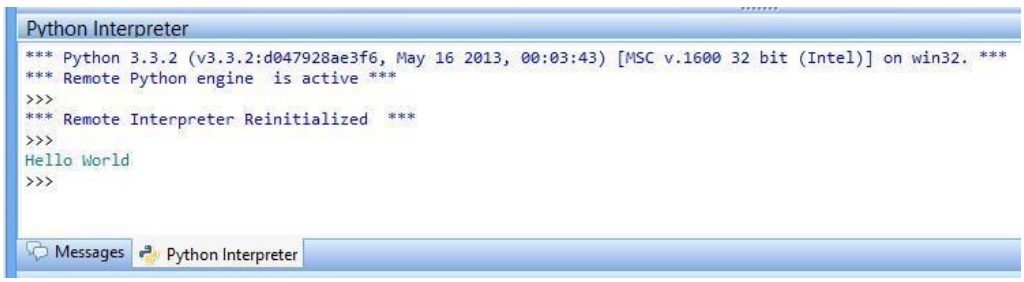
Recall, the objective is to display the message "Hello World". The program area should now look something like this:

FIGURE 1.3: print("Hello World")



Run the program by clicking on the green arrow ( ) to see what it does. You should see the message appear on the bottom of the screen.

FIGURE 1.4: Run the program

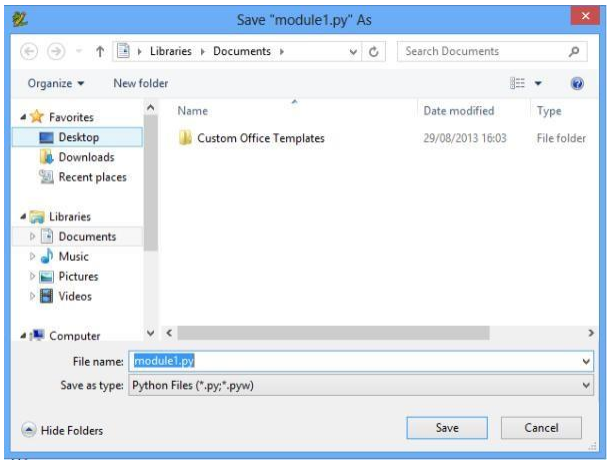


Congratulations! You have just written and run your first Python program.

The next thing to do is to save the program. To do this choose the **File->Save** option. Use the dialog displayed below to browse to the destination folder. (I usually place my files in a folder called 'src' for source.). Choose an appropriate name for your programs.

It is important to keep your source files well organised on disk (I recommend using Cloud) and save your programs (by pressing the control key and S at the same time i.e. Ctrl+S) frequently as you work.

FIGURE 1.5: Save it!



After you have saved your program close PyScripter. As a test to yourself see if you can re-launch PyScripter, Open (using **File->Open**) and run the "Hello World" program. Once you can do this confidently you are ready to start writing more programs.

A note on the `print` command

`print` is a special Python command which tells the computer to display a message on the output screen. The **`print`** command is very easy to use.

Simply type the word **print** ….	`print`
… followed by open bracket….	`print(`
… then open quotation mark…	`print("`
… enter the text to be displayed…	`print("Hello World`
… close the quotation mark…	`print("Hello World"`
… finally, close the bracket.	`print("Hello World")`

You've figured it out? It must be exactly like this

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

Python – like every programming language - is *very* fussy about the way you type in your code. You need to be careful. If you make a mistake Python will not understand and display
syntax error.

Syntax Errors
The **syntax** of any language (including natural languages) refers to rules regarding how statements and expressions (sentences) in that language can be correctly constructed.

When you try to run a program that has a syntax error Python displays an error in a message box. Sometimes these messages are difficult for programmers to understand but with time and practice you will get used to them.

KEY POINT: When you see a syntax error you need read it carefully and then click ok to make the message box 'go away' and then correct your code so that the error won't appear when you try to run it again.

We will now take a look at some of the basic syntax rules of Python and illustrate what happens when these rules are broken.

Rule 1: Python is *case sensitive*. Upper and lower case letters are different. Lowercase letters must be used for Python commands. Try running the following:

```
Print("Hello World")
```

The code contains an error and so when you try to run it the following error message is displayed.

FIGURE 1.6: NameError notice!

Read the error text carefully. This error is caused because Python does not understand the name 'Print'. To correct the error you would need to type print using lowercase only.

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

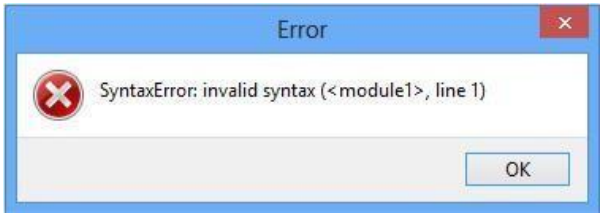
Notice that Python does not care too much about what we put *between* the quotation marks.

Rule 2: Strings must be enclosed in quotations marks. If either, or both, quotation marks are missing a syntax error message is displayed. Try running the following:

```
print(Hello World)
```

The following error message is displayed.

FIGURE 1.7: SyntaxError notice!

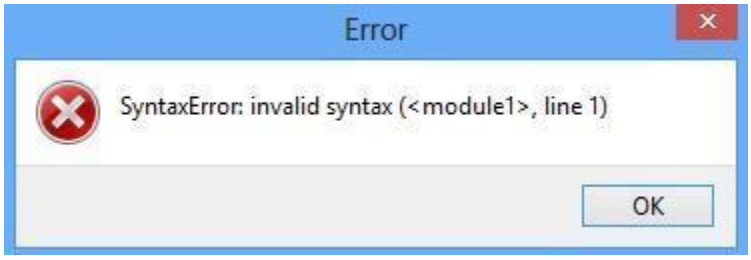


To correct the error you would need to insert the quotation marks.

Rule 3: You should always use brackets after the `print` command. Furthermore, for every opening bracket there needs to be a matching closing bracket. The following line causes a syntax error to be displayed

```
print "Hello World"
```

FIGURE 1.8: SyntaxError 2 notice!



To correct the error you would need to insert the brackets.

Rule 4: Every line of Python code must be properly **indented**.

For the moment it is best to assume that there should be no spaces before any line of code. Notice in the code below that there is a space before the print command. This

causes an error to be displayed when you try to run the program.

FIGURE 1.9:IndentationError notice!

To correct the error you would need to remove the space from the start of the line.

Summary
Python programs must be written to obey the syntax rules of the Python programming language. If they do not, they will not execute properly and a syntax error is displayed. Programs that contain syntax errors are said to be *syntactically incorrect*. Such programs need to be corrected before they can be run.

As a final point it is worth noting that just because a program does not contain syntax errors is no guarantee that it will run properly. Syntactically correct programs can contain errors (sometimes called *bugs*). Programs that contain bugs do not work in the way they are meant to work. Such programs are called *semantically incorrect*.

Exercise - test your knowledge!
Read through each of the following lines of code one by one. For each line state whether it is syntactically correct or not. If the line is incorrect state why?

```
   PRINT("Hello World")
   print("HELLO WORLD")
   print("Python is fun!")
   print"(Hello World)"
   display("Hello World")
   print(My name is Sam)
   print(("Hello World")
h) print("This    text    has plenty      of   spaces")
   pint("Hello World")
   print("99 red balloons")
   print(99 red balloons)
```

<u>And finally,</u>
Python is not too fussy about what you type inside quotation marks. Outside quotation marks Python is very limited in what it understands. One of the things Python

understands outside quotation marks is *numbers*. Numbers do not have to be enclosed inside quotations.

The statements below are all syntactically correct. (Try them!)

```
   print(99, "red balloons")
   print(99)
print("What's your age? ")
print(21)
   print("My age is 21")
   print("My age is", 21)
print("I'm", 18, "and my friend is", 21)
print("The print command", "can handle", "more than 1 string.")
```

KEY POINT: The technical word for text is *string*. A string is a sequence of characters enclosed inside quotation marks.

Notice from the last two examples how multiple strings can be separated by commas.

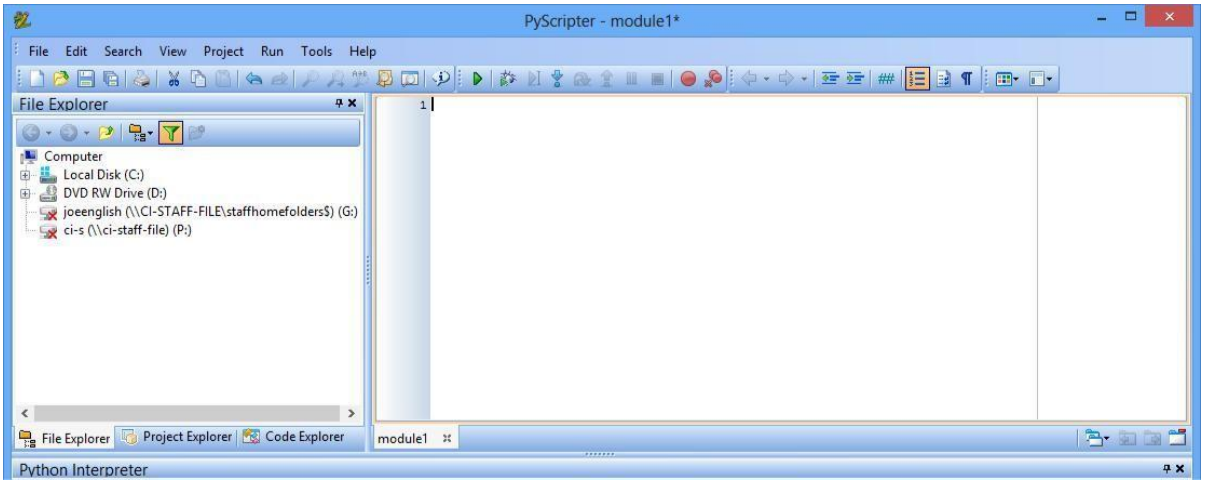
Lesson 2: A poem!

In this lesson you will write a program that causes the short poem shown below to be displayed on the output screen.

As I was going out one day
My head fell off and rolled away,
But when I saw that it was gone,
I picked it up and put it on.

As before we open PyScripter and clear the code area. You should see a screen like this.

FIGURE 1.10: PyScripter's Screen

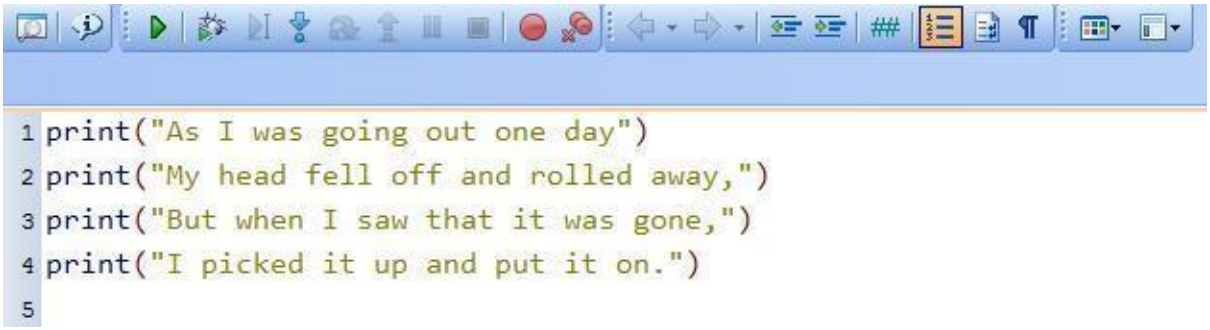


Now key in the following Python code:

```
print("As I was going out one day")
print("My head fell off and rolled away,")
print("But when I saw that it was gone,")
print("I picked it up and put it on.")
```

The code area on the screen should now look something like this.

FIGURE 1.11: Code Area



Now run your program (remember, use the green arrow ) and you should see the poem displayed in the output area of the screen.

Finally, save your program as before.

Flow of Control
It is important that programmers understand how computers run (i.e. execute) programs.

Programmers must train their brain to run their programs in the same way as the computer.

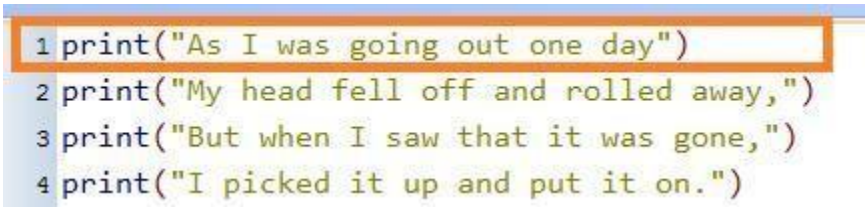
The *flow of control* means the order in which the lines of a computer program are run by the computer. For the moment we will assume that the flow of control is *sequential*. This means that program execution begins at the first line and ends at the last line. Every line in between is executed in the same sequence as they appear in the program.

To illustrate this we will 'step through' the execution of our 'display poem' program:

```
print("As I was going out one day")
print("My head fell off and rolled away,")
print("But when I saw that it was gone,")
print("I picked it up and put it on.")
```

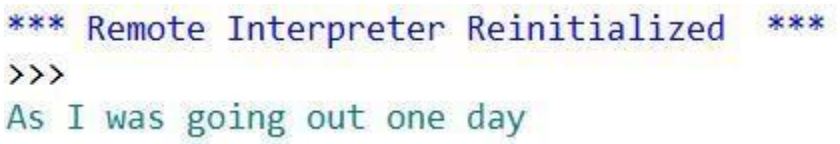
Execution starts at line 1 as highlighted below:

FIGURE 1.12: Line Execution



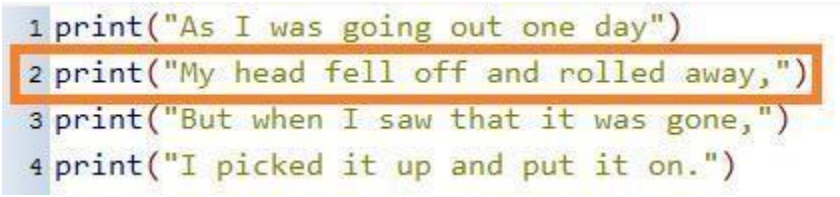
When this line is executed the output console displays the string *As I was going out one day*

FIGURE 1.13: Console result



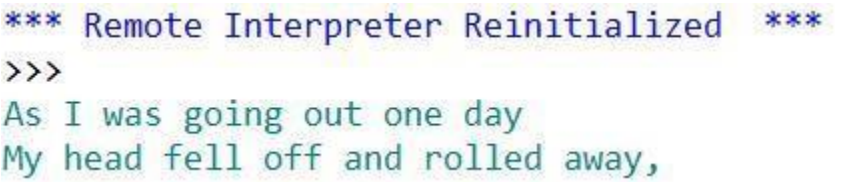
The flow of control then moves to line 2 as highlighted

FIGURE 1.14: Second Line Execution



Line 2 is executed and the output console now looks as follows:

FIGURE 1.15: Second Line Result



Program execution continues in a sequential manner until finally the last line (in this case line 4) is executed.

Once line 4 has been executed in this program the output console looks like this:

FIGURE 1.16: Overall Result

```
*** Remote Interpreter Reinitialized  ***
>>>
As I was going out one day
My head fell off and rolled away,
But when I saw that it was gone,
I picked it up and put it on.
```

Recall, that this was what we wanted the program to do (i.e. display the poem). We can verify that the program works by comparing the actual output to what we expected.

It is very important to realise that although the programs run very quickly by the computer. This gives the illusion that the output is displayed in one step. The reality however is that the output is built up on a line by line basis as each individual line of code is executed.

Challenge
Extend the 'display poem' program to show the following 2nd verse

And when I got into the street
A fellow cried look at your feet!
I looked at them both and sadly said
I've left them both asleep in bed

The final output window should look like this:

FIGURE 1.17: Display the Poem!

```
Python Interpreter
*** Remote Interpreter Reinitialized  ***
>>>
As I was going out one day
My head fell off and rolled away,
But when I saw that it was gone,
I picked it up and put it on.
And when I got into the street
A fellow cried look at your feet!
I looked at them both and sadly said
I've left them both asleep in bed
>>>
```

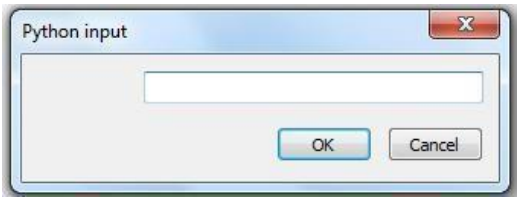
Experiment
Key in the following code and see what it does.

```
print("Knock knock.")
input()
print("Who's there?")
input()
print("Doctor.")
input()
print("Doctor who?")
input()
print("How did ya guess!")
```

Don't forget to save your program somewhere you will be able to find it later. Call it something relevant e.g. KnockKnock.py.

Notice that the `input()` command causes the following screen to appear in PyScripter.

FIGURE 1.18: Python's input()



You need to press enter (or click the OK button) to make this screen disappear. Program execution continues each time at the next line of code.

Take a careful look at the output console each time the above screen is displayed. What do you notice?

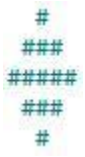
Describing what a program does
Question. Can you describe (on a line by line basis) what the following program does?

```
print("  #")
print(" ###")
print("#####")
print(" ###")
print("  #")
```

Answer. Line 1 causes two spaces followed by the "#" (often called the hash or pound character) to be displayed. Line 2 displays a space followed by three hash characters. Line 3 displays five hash characters. Line 4 does the same as line 2 and line 5 the same as line 1. The overall effect is to display the following pattern on the output

screen:

FIGURE 1.19: Diamond Asterisks



KEY POINT: You will learn Python faster if – every time you come across a program listing you haven't seen before - you take some time to study it and try to figure out what the program does.

Exercises 1

Draw the patterns that would be generated by the following programs. (Hint: You can key each program in and run them to see the pattern displayed.)

```
1. print("####")
2.
print("#     #")
3.
print("#     #")
4. print("####")
```

```
1.
print("####")
2. print("#
#")
3.
print("####")
4. print("#")
5. print("#")
```

Can you spot the syntax error(s) in each of the following code snippets?
Hint: you should start by typing the code into PyScripter, and you will see the syntax error highlighted when you run the code. Can you correct the errors?

PROGRAM 1

```
print["Knock knock."]
print("Who's there?")
print["Doctor.")
print "Doctor Who?"
print("How did ya guess!"]
```

_____

_____

_____

PROGRAM 2

```
Print(How did ya guess!)
prin(Doctor Who?)
print("Doctor.)
pint(Who's there?)
print("Knock knock.")
```

_____

_____

_____

PROGRAM 3

```
print("Knock knock.")
input
input
input
input
print("Sorry for delay!")
```

Write a program that displays the following:

```
Hello, my name is Sam!
```

Extend the previous program so that it displays the following:

```
Hello, my name is Sam!
Sam I am.
Do you like green eggs and ham?
```

Write a program that displays your own name and address

Re-arrange the lines of code below into a program that displays the pattern shown on the right. Note that you can use any line as often as you like but you won't need to use every line.


```
###### ###### ######
   ##  ##     ##
###### ###### ######
   ##  ##  ##     ##
###### ###### ######
```

7. Write programs to display the following (individual) patterns.

a)

```
######
#
#
#
######
```

b)

```
######
#
######
#
######
```

c)

```
######
#
######
#
#
```

d)

```
#      #
#      #
########
#      #
#      #
```

e)

```
   #
   #
   #
   #
   ########
```

f)

```
########
#      #
#      #
#      #
########
```

g)

```
########
   #
   #
   #
   #
```

h)

```
#      #
#      #
#      #
#      #
######
```

By using the code you just wrote, write a program that outputs the letters LOFT (one letter underneath the next).

The following three programs all result in the same output being displayed. What does this tell you about the `print` command?

PROGRAM 1

```
print("2 4 6 8")
```

PROGRAM 2

```
print("2 4", "6 8")
```

PROGRAM 3

```
print(2, 4, 6, 8)
```

Write programs to do the following:

display the numbers 1 to 5, in a horizontal line (*Hint:* use one `print()` statement)

display the numbers 1 to 5, one under the other (*Hint:* use five `print()` statements)

Write a program that displays the first 10 prime numbers.

Lesson 2

Functions (Part I)

INTRODUCTION

Let us develop some good habits before delving too deep into programming. To do this we need to learn about **functions**. Functions are the building blocks of computer programs.

A function can be thought of a sub-program that does a specific job for the programmer. Computer programs typically contain many functions. Functions are important and useful because they are re-usable. Every function definition is made up of a **header** and a **body**.

The header is always the first line – it always starts with the word **`def`** followed by some name chosen by the programmer followed by brackets and a colon.

The name of the function shown below is `displayPoem`. The body of this function contains four `print` statements. Notice that the statements appear further to the right than the header line. This is called **indentation**. In Python, the statements inside a function body must always be indented.

```
def displayPoem():
   print("One fine day in the middle of the night,")
   print("Two dead men got up to fight,")
   print("Back to back they faced each other,")
   print("Drew their swords and shot each other.")
```

The above is an example of a **function definition**.

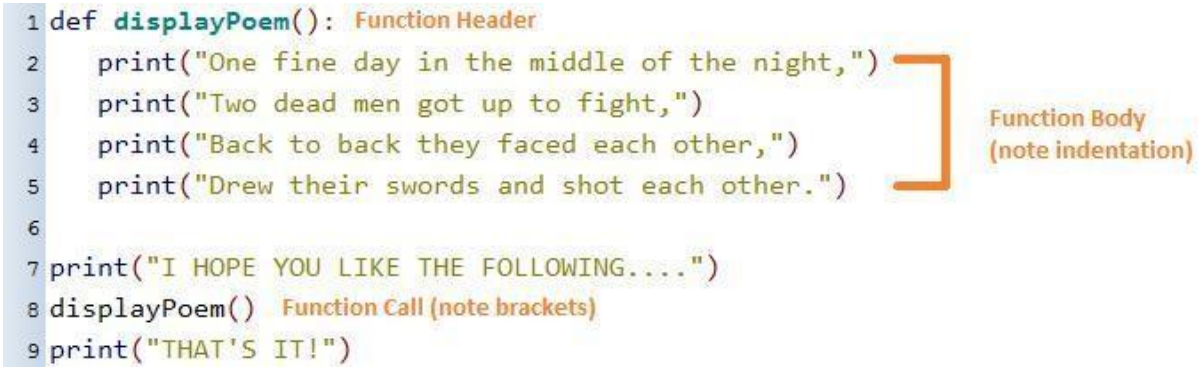
To execute the code inside a function body, a programmer must write a line of code to **call the function**. To call a function just type the name of the function followed by brackets.

```
displayPoem()
```

In a Python program the code to define a function must appear before the code to call it. When a call is made, the flow of control jumps to the first line of the function body and execution continues from that point until the last line of the function body has been executed. At this point the flow jumps back to the point from which the call to the function was made.

Let's look at the short program below.

FIGURE 2.1: displayPoem()

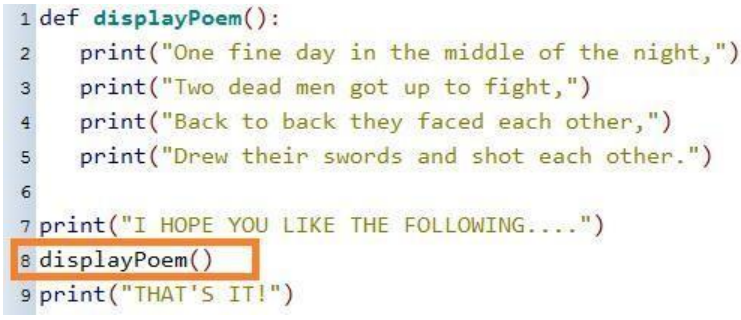


The screenshots on the next pages illustrate the flow of control as this simple program is executed. Whatever line is being executed by Python is highlighted in red in the screenshots.

Python will start to run this program from line 7.

Line 7 causes the string *I HOPE YOU LIKE THE FOLLOWING….* to be displayed. Execution
then moves to line 8.

FIGURE 2.2: Call the Function



Line 8 calls the function `displayPoem`. Note the use of brackets. The flow of control jumps to line 2.

FIGURE 2.3: Function's behavior (Part 1)

```
1 def displayPoem():
2     print("One fine day in the middle of the night,")
3     print("Two dead men got up to fight,")
4     print("Back to back they faced each other,")
5     print("Drew their swords and shot each other.")
6
7 print("I HOPE YOU LIKE THE FOLLOWING....")
8 displayPoem()
9 print("THAT'S IT!")
```

After line 2, line 3 is executed – this is illustrated below

FIGURE 2.4: Function's behavior (Part 2)

```
1 def displayPoem():
2     print("One fine day in the middle of the night,")
3     print("Two dead men got up to fight,")
4     print("Back to back they faced each other,")
5     print("Drew their swords and shot each other.")
6
7 print("I HOPE YOU LIKE THE FOLLOWING....")
8 displayPoem()
9 print("THAT'S IT!")
```

After executing line 3 the output display looks like:

FIGURE 2.5: Result of the function

```
I HOPE YOU LIKE THE FOLLOWING....
One fine day in the middle of the night,
Two dead men got up to fight,
```

Lines 4 and 5 are then executed in turn.

FIGURE 2.6: Other behavior of displayPoem()

```
def displayPoem():
    print("One fine day in the middle of the night,
    print("Two dead men got up to fight,")
    print("Back to back they faced each other,")
    print("Drew their swords and shot each other.")

print("I HOPE YOU LIKE THE FOLLOWING....")
displayPoem()
print("THAT'S IT!")
```

```
def displayPoem():
    print("One fine day in the middle of the night,
    print("Two dead men got up to fight,")
    print("Back to back they faced each other,")
    print("Drew their swords and shot each other.")

print("I HOPE YOU LIKE THE FOLLOWING....")
displayPoem()
print("THAT'S IT!")
```

The change in indentation tells Python that line 5 is the last line of the function. Once line 5 has been executed the flow of control *returns* to the point from where the original function was called i.e. line 9.

FIGURE 2.7: Print the result!

```
1 def displayPoem():
2     print("One fine day in the middle of the night,")
3     print("Two dead men got up to fight,")
4     print("Back to back they faced each other,")
5     print("Drew their swords and shot each other.")
6
7 print("I HOPE YOU LIKE THE FOLLOWING....")
8 displayPoem()
9 print("THAT'S IT!")
```

Line 9 is executed and the program terminates as there are no more lines left to execute.
The final message displayed is:

FIGURE 2.8: Look again at the result

```
I HOPE YOU LIKE THE FOLLOWING....
One fine day in the middle of the night,
Two dead men got up to fight,
Back to back they faced each other,
Drew their swords and shot each other.
THAT'S IT!
```

Test your understanding.
Design and write a function to display the output shown.

Way down south where banannas grow,
A grasshopper stepped on an elephant's toe.
The elephant said, with tears in its eyes,
'Pick on somebody your own size.'

LEARNING OBJECTIVES

1. **Understand:** Identify the key components of a function, including the function header and body, and describe the purpose of indentation in Python functions.
2. **Apply:** Write a Python function to display a specific set of print statements and call the function within a program to demonstrate its reusability.
3. **Analyze:** Evaluate the modularity of a given Python program and suggest improvements by refactoring the code to use functions more effectively.

LECTURE DISCUSSION

Lesson: Developing modular code using functions

Recall, from an earlier exercise you were asked to write a program to display the following '365' pattern.

FIGURE 2.9: 365 in asterisks

A initial solution to the problem is shown below.

FIGURE 2.10: Print the 365 in asterisks

```
1
2 print("#######  #######  #######")
3 print("      #  #        #")
4 print("#######  #######  #######")
5 print("      #  #     #        #")
6 print("#######  #######  #######")
7
```

Key the code in to make sure it does what it is meant to do. (Be careful to get the number of spaces correct on lines 3 and 5!)

The program is quite simple. Five calls to `print()` and voila – job done!

Let's look at an alternative solution – this time using a function called `draw365`.

FIGURE 2.11: Create function for 365

```
1
2 def draw365():
3     print("#######  #######  #######")
4     print("      #  #        #")
5     print("#######  #######  #######")
6     print("      #  #     #        #")
7     print("#######  #######  #######")
8     return
9
10 draw365()
11
```

This solution is better than the first because it is more **modular**. Recall, a function is a sub-program that does a specific task. In this case, the task is to display the '365' pattern, and so, by writing a function to do this we are making the system more modular.

A modular system is one in which there is a separate function defined for each separate task system performs. Writing modular code is considered good programming practice. It is achieved by breaking larger blocks of code into smaller blocks of related code through the use of functions. The initial solution is said to lack modularity.

The advantage of the second solution is that it just takes a single line of code (i.e. a call to the function `draw365`) to draw another '365' pattern. In effect, we are re-using existing code. We will return to the important topic of code reuse at a later stage.

Finally, it is worth noting that `print` is a special type of function called a **built in** function. This is because it is built into the Python language. In contrast, functions we define ourselves are called **user-defined functions**.

Exercises 1

1. Study the code below carefully and answer the questions that follow:

```
1  def displayName():
2      print("John Doe")
3
4  def displayAddress():
5      print("1 Main Street")
6      print("Newtown")
7      print("Navan")
8      print("Co. Meath")
9
10 displayName()
11 displayAddress()
```

How many functions are defined?

What are the names of the functions?

Write out the function header for the functions?

On what line number does program execution begin?

What line is executed after line 10?

What line is executed after line 2?

Write down the program output.

2. *Match the following sentences with their meaning*

The first word in the function header	Function
The sequence in which lines of code are executed	Function Body
The type of functions that come as part of Python	Def
This symbol must always appear at the very end of the function header	Function Call
A sub-program that does a specific task	Built-in
Functions	
A function header and a function body	Colon(:)
A special line that cause the flow of control to switch to the first line inside the function body	Flow of Control
The block of code after the function header	Function Definition

3. *Study the two programs shown and answer the question which follows.*

```
1  def displayName():
2      print("John Doe")
3
4  def displayAddress():
5      print("1 Main Street")
6      print("Newtown")
7      print("Navan")
8      print("Co. Meath")
9
10 displayName()
11 displayAddress()
```

PROGRAM 1

```
1  def displayName():
2      print("John Doe")
3
4  def displayAddress():
5      print("1 Main Street")
6      print("Newtown")
7      print("Navan")
8      print("Co. Meath")
9
10 def displayNameAndAddress():
11     displayName()
12     displayAddress()
13
14 displayNameAndAddress()
```

PROGRAM 2

What logical difference, if any, exists between the two listings? Explain your answer.

It is up to the programmer to decide what to call each function. It is important that functions are given meaningful names. Take a look at the scenarios below and suggest an appropriate function name for each scenario. A function that displays….

Scenario	Suggested function name
the lyrics of a song	_____
a person's medical history	_____
instructions on how to play a game	_____
a welcome message (e.g. ATM machine)	
a bank account balance	_____
a recipe for a cake	_____
a recipe for an apple tart	_____
the list of students is a particular class	_____

The code shown below defines a function to print a verse of a well known song called
"To the city of Chicago".

```
1 def printChorus():
2     print("To the city of chicago")
3     print("As the evening shadow falls")
4     print("There are people dreaming")
5     print("Of the Hills of Donegal")
6     print("")
7     return
```

Notes:

1. The statement `print("")` causes a single blank line of output to be displayed
2. The statement `return` causes the function to end. (Execution continues at the next line after the function call.)

Answer the following:

What is the name of the function?

Write a line of code to call the function

Try running the following code.

```
def humptyDumptyVerse1():
print("Humpty Dumpty sat on a wall,")
print("Humpty Dumpty had a great fall.")
print("All the king's horses and all the king's men")
print("Couldn't put Humpty together again")
```

Explain the syntax error you get. How can the error be fixed?

7. What happens when you run the code below? How can the problem be fixed?

```
def humptyDumptyVerse1():
   print("Humpty Dumpty sat on a wall,")
   print("Humpty Dumpty had a great fall.")
   print("All the king's horses and all the king's men")
   print("Couldn't put Humpty together again")
```

The program below displays a little known limerick written by Edward Lear entitled *There was an old man of Nepaul*.

Write a function called `nepaul` to display the limerick. Don't forget to call the function.

```
print("There was an old man of Nepaul")
print("From his horse had a terrible fall;")
print("But, though split quite in two,")
print("By some very strong glue,")
print("They mended that man of Nepaul.")
```

A lesson in design – Green Eggs and Ham!

In this lesson we introduce how functions can be used to support good design habits

Let us say we were asked to write a program to display the poem shown below (adapted from Dr Seuss's classic Green Eggs and Ham)

FIGURE 2.12: Result of Green Eggs and Ham!

```
I do not like green eggs and ham.
I do not like them Sam-I-am.

I do not like them here or there.
I do not like them anywhere.
I do not like them in a house
I do not like them with  a mouse

I do not like green eggs and ham.
I do not like them Sam-I-am.

I do not like them in a box
I do not like them with a fox
I will not eat them in the rain.
I will not eat them on a train

I do not like green eggs and ham.
I do not like them Sam-I-am.
```

An initial solution might look like this:

```
    print("I do not like green eggs and ham.")
    print("I do not like them Sam-I-am.")
    print()
    print("I do not like them here or there.")
    print("I do not like them anywhere.")
    print("I do not like them in a house")
    print("I do not like them with  a mouse")
    print()
    print("I do not like green eggs and ham.")
    print("I do not like them Sam-I-am.")
    print()
    print("I do not like them in a box")
    print("I do not like them with a fox")
    print("I will not eat them in the rain.")
    print("I will not eat them on a train")
    print()
    print("I do not like green eggs and ham.")
```

```
   print("I do not like them Sam-I-am.")
```

This code works but is considered poor design because:
- it contains **duplication** (lines 1,2 are repeated twice – at 9,10 and again at 17,18)
- it lacks **modularity** (i.e. it does not use functions)

To eliminate the duplication we write a function to display the chorus and call it when needed. This is done below.

```
   def showChorus():
      print("I do not like green eggs and ham.")
      print("I do not like them Sam-I-am.")
      return
5.
   showChorus()
   print()
   print("I do not like them here or there.")
   print("I do not like them anywhere.")
   print("I do not like them in a house")
   print("I do not like them with  a mouse")
   print()
   showChorus()
   print()
   print("I do not like them in a box")
   print("I do not like them with a fox")
   print("I will not eat them in the rain.")
   print("I will not eat them on a train")
   print()
   showChorus()
```

Hopefully, you can see that this program displays the same output. Execution starts at line 6.
The function showChorus is useful because it is re-used.

However, the above program still lacks modularity. A more modular solution is shown below:

```
   def showChorus():
      print("I do not like green eggs and ham.")
      print("I do not like them Sam-I-am.")
      return
5.
   def showVerse1():
      print()
      print("I do not like them here or there.")
      print("I do not like them anywhere.")
      print("I do not like them in a house")
      print("I do not like them with  a mouse")
      print()
      return
14.
   def showVerse2():
      print()
      print("I do not like them in a box")
```

```
    print("I do not like them with a fox")
    print("I will not eat them in the rain.")
    print("I will not eat them on a train")
    print()
    return
23.
showChorus()
showVerse1()
showChorus()
showVerse2()
showChorus()
```

The use of functions to do specific tasks make the program more modular. Modular code is the result of good design.

Exercises 2:

1. The purpose of this question is to give a better understanding of functions

a) Write a three line program (without using functions) that displays following output.

```
Hello, my name is Sam!
Sam I am.
Do you like green eggs and ham?
```

b) What does the program below do?

```
1 def displayMessage():
2     print("Hello, my name is Sam!")
3     print("Sam I am.")
4     print("Do you like green eggs and ham?")
5
6 displayMessage()
```

```
Hello, my name is Sam!
Sam I am.
Do you like green eggs and ham?
Hello, my name is Sam!
Sam I am.
Do you like green eggs and ham?
```

c) How would you modify both programs so that the output displayed is as follows?

Use your answer to describe *one* benefit of functions.

Design and write *two* functions to display the verses below. (Call them `verse1` and `verse2`).

The cabbage is a funny veg.
All crisp and, green and, brainy.
I sometimes wear one on my head
When it's cold and rainy.

I eat my peas with honey;
I've done it all my life.
It makes them taste quite funny,
But it keeps them on a knife

Now design and write a single function (call it `displayVerses`) that calls the functions you have just written. Call `displayVerses`. Explain why the program is *modular*.

3. Read the program shown below and answer the questions that follow.

```
1 print("To market, to market")
2 print("to buy a fat pig.")
3 print("Home again, home again")
4 print("Jiggity jig!")
5 print("")
6 print("To market, to market")
7 print("to buy a fat hog.")
8 print("Home again, home again")
9 print("Jiggity jog!")
```

What output does the program display?

How could the program be made more modular?

What lines are duplicated? How could this duplication be removed?

Fill in the blanks below so that the program will display the same output (as the program in the previous question)

```
1  def toMarket():
2      print("To market, to market")
3
4  def homeAgain():
5      print("Home again, home again")
6
7  toMarket()
8  print("to buy a fat pig.")
9  homeAgain()
10 print("                ")
11 print("")
12
13
14
15 print("Jiggity jog!")
```

5. Fill in the blanks below so that the program is modular and contains no duplicate lines.

```
def toMarket():
    print("                    ")

def homeAgain():
    print("                    ")

def verse1():
    toMarket()
    print("to buy a fat pig.")
    homeAgain()
    print("Jiggity jig!")

def verse2():
    toMarket()
    print("to buy a fat hog.")
    homeAgain()
    print("Jiggity jog!")

def displayPoem():
    
    print("")
    


displayPoem()
```

Design and write a modular program to display the following 2 verses of the well known poem Humpty Dumpty. Your solution should not have any duplication.

Humpty Dumpty sat on a wall
Humpty Dumpty had a great fall.
All the king's horses and all the king's men
Couldn't put Humpty together again.

Humpty Dumpty sat on a wall
Humpty Dumpty had a great fall.
Four-score Men and Four-score more,
Could not make Humpty Dumpty as he was before.

The program below generates the lyrics of the song "To the city of Chicago". Use the space on the right-hand side below to write all the output generated by the program (shown on the left)

Note
The # symbol tells Python to ignore the text that comes after it. It is called a **comment**.

OUTPUT

PROGRAM LISTING

```
# Function to print the song chorus
def printChorus():
    print("To the City of Chicago,")
    print("As the evening shadows fall,")
    print("There are people dreaming,")
    print("Of the hills of Donegal.")
    return

# Function to print the 1st verse
def printVerse1():
    print("Eighteen forty seven,")
    print("Was the year it all began,")
    print("Deadly Pains of hunger,")
    print("Drove a million from the land,")
    print("They journeyed not for glory,")
    print("Their motive wasn't greed,")
    print("Just a voyage of survival,")
    print("Accross the stormy sea.")
    return

# Function to print the 2nd verse
def printVerse2():
    print("Some of them knew fortune,")
    print("And some them knew fame,")
    print("More of them knew hardship,")
    print("And died upon the plain,")
    print("They spread throughout the nation,")
    print("Rode the railroad cars,")
    print("Brought their songs and music,")
    print("To ease their lonely hearts.")
    return

# Program execution starts here
printChorus()
printVerse1()
printChorus()
printVerse2()
printChorus()

```

Explain the main benefit of the function `printChorus` in the above program.

What do you think the purpose of program *comments* are?

8. Assume you are given a program with the function definition below,

```
def printChorus():
    print("To the city of chicago")
    print("As the evening shadow falls")
    print("There are people dreaming")
    print("Of the Hills of Donegal")
    print("")
    return
```

Now, use the space provided to the right below to write the code necessary to produce the outputs shown on the left.

	Desired Output	Code (write your code here)
a)	TO THE CITY OF CHICAGO To the city of chicago As the evening shadow falls There are people dreaming Of the Hills of Donegal	
b)	TO THE CITY OF CHICAGO To the city of chicago As the evening shadow falls There are people dreaming Of the Hills of Donegal	
c)	TO THE CITY OF CHICAGO Christy Moore To the city of chicago As the evening shadow falls There are people dreaming Of the Hills of Donegal	
d)	Christy Moore To the city of chicago As the evening shadow falls There are people dreaming Of the Hills of Donegal Anyone for the last few Choc Ices, now Lisdoon, Lisdoon, Lisdoon, Lisdoonvarna!	

*Study the code listing below and see if you can figure out what it does. Use the space on the right hand side to write the **expected output**.*

PROGRAM LISTING	OUTPUT

```
1  def foo():
2      print("Starting foo()")
3      print("Leaving foo()")
4
5  def bar():
6      print("Starting bar()")
7      foo()
8      print("Leaving bar()")
9
10 def foobar():
11     print("Starting foobar()")
12     bar()
13     print("Leaving foobar()")
14
15
16 foo()
17 bar()
18 foobar()
19
```

Now key the program in and run it. Compare the expected output with the **actual output**. Is the actual output the same as the expected output? What does your answer mean? If the actual and expected output were different what would it mean?

How would the output of the program be altered if lines 7 and 12 were removed? Try it.

What do you think it would be a bad idea to insert a call to the function `bar()` inside the function `foo()`? Don't try this!

Make the changes necessary so that following outputs are generated. Answer each part separately and in turn.

(i)

```
Starting bar()
Leaving bar()
Starting foo()
Leaving foo()
Starting foobar()
Leaving foobar()
```

(ii)

```
Starting foobar()
Starting foo()
Leaving foo()
Leaving foobar()
Starting foo()
Leaving foo()
Starting bar()
Leaving bar()
```

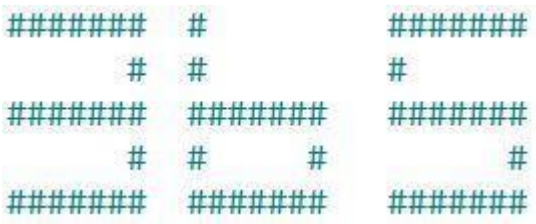
(iii)

```
Starting foo()
Starting bar()
Leaving bar()
Leaving foo()
Starting bar()
Leaving bar()
Starting foobar()
Starting bar()
Leaving bar()
Leaving foobar()
```

A study in design – displaying a 365 pattern

Imagine you were asked to write a program to display the following '365' pattern.

FIGURE 2.12: Display of '365' pattern



We will look at three different solutions.

FIGURE 2.13: 365 Solution 1

```
1
2 print("#######  #######  #######")
3 print("      #  #        #")
4 print("#######  #######  #######")
5 print("      #  #     #        #")
6 print("#######  #######  #######")
7
```

365 Solution 1

The first implementation is the simplest. Five calls the `print()` function and voila! However this solution lacks modularity. If (for some reason) we need to have the '365' pattern displayed more than once then we would need to duplicate these lines elsewhere in the program. By now we should know that duplication of code is bad practice and should be avoided. The best way to avoid this situation is to use a function.

FIGURE 2.14: 365 Solution 2

```
1
2 def draw365():
3     print("#######  #######  #######")
4     print("      #  #        #")
5     print("#######  #######  #######")
6     print("      #  #     #        #")
7     print("#######  #######  #######")
8     return
9
10 draw365()
11
```

365 Solution 2

The above implementation has the advantage that the functionality to display the pattern is now contained in a user defined function called, `draw365`. Anytime, the programmer needs to have the '365' pattern displayed he/she just needs to call this function.

At this stage there would appear to be little room for improvement on this solution. However, if you look closely you should notice that three of the lines (lines 3, 5, and 7) are identical. These lines all do the same thing i.e. display a line of hash tags. This is code duplication and, as before can be avoided through clever use of functions.

FIGURE 2.15: 365 Solution 3

```
1
2  def displayLine():
3      print("#######  #######  #######")
4      return
5
6  def draw365():
7      displayLine()
8      print("      #  #        #")
9      displayLine()
10     print("      #  #     #        #")
11     displayLine()
12     return
13
14 draw365()
15
```

365 Solution 3

Our third and final implementation includes a new user-defined function called `displayLine`. This simple little function just displays a horizontal line of # symbols and is called three times inside the other user-defined function `draw365`.

Although, this solution does not seem to be as obvious or simple as the earlier implementations, it does incorporate better design techniques. Better design techniques lead to better code because the resulting systems are easier (and cheaper) to maintain. Such systems are said to be *robust* and *extensible*.

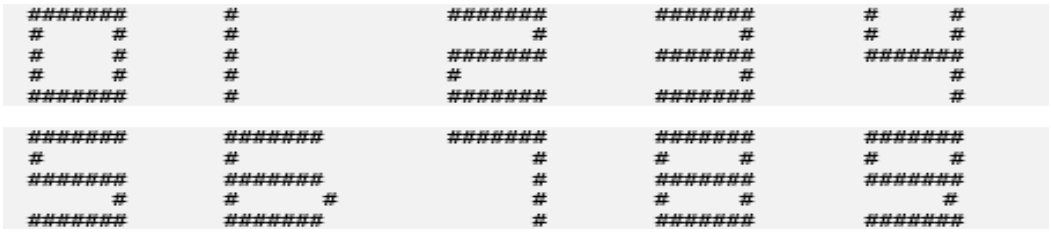
Worked Example (0-9, digital digits)

Design and implement a program to display any of the digits zero through nine shown below. We need to look for common patterns.

Every digit has five rows of hash tags.

We notice that zero and eight are almost identical (the only difference being the middle line).

Every digit (except 1) uses a sequence of seven hash tags (#######) at some place or other. For example, this pattern appears on the first and fifth line in the zero. It occurs on the first, third and fifth line in both two and three.



Other recurring patterns are :

is used in 0, 4, 8 and 9
appears in 2, 3, 4, 5, 6, 7, and 9 #
occurs in 1, 2, 5, and 6

The next step in the design is to put names on the patterns.

Pattern	Our Name
###### #	HashTagLine
# #	LeadingAndTrailingHashTag
#	TrailingHashTag
#	LeadingHashTag

FIGURE 2.16: Names using hashtags

We now write our own user defined functions to display these patterns.

```
1
2  def drawHashTagLine():
3      print("#######")
4      return
5
6  def drawLeadingAndTrailingHashTag():
7      print("#     #")
8      return
9
10 def drawLeadingHashTag():
11     print("#")
12     return
13
14 def drawTrailingHashTag():
15     print("      #")
16     return
17
```

FIGURE 2.17: Display patterns!

Our next task is to use the above functions to draw the actual digits. We define our own user defined functions to draw the digits themselves. As an example the following code draws the digit zero when called.

FIGURE 2.18: drawZero()

```
18 def drawZero():
19     drawHashTagLine()
20     drawLeadingAndTrailingHashTag()
21     drawLeadingAndTrailingHashTag()
22     drawLeadingAndTrailingHashTag()
23     drawHashTagLine()
24     return
25
```

Finally, to display zero as output we just call the function `drawZero()` as follows:

```
26 drawZero()
```

FIGURE 2.19: call drawZero()

Exercise 3:

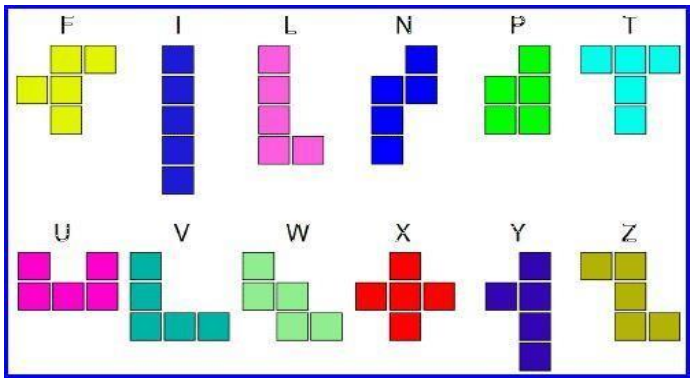
Key in the above code (all 26 lines) and run the program. What does it do?
Now, write additional functions to display the other 9 digits.
(*Hint*: You will need to define and call one function per digit. Do them one at a time. Each function will contain 5 lines followed by return.)

Chapter Exercises

Study the shapes below and see if you can spot any re-use of patterns. (Ignore the colours.)

For example shapes labelled F and Z have the following in common:

3 rows of blocks
The first line of shape F is the exact same as the last line of shape Z
The second line of shape F is the exact same as the first line of shape Z



Imagine each block was represented by a '#' character. Now key in and run the code below.

```
def hash():
print("#")

def blankHash():
print(" #")

def hashHash():
print("##")

def blankHashHash():
print(" ##")

blankHashHash()
hashHash()
blankHash()
```

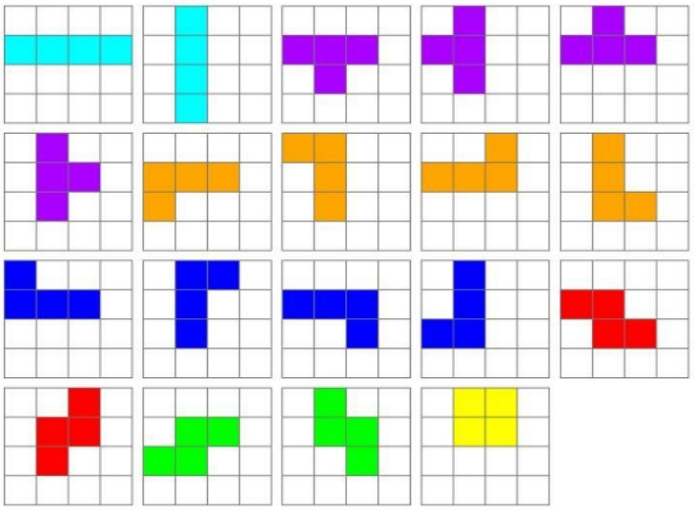
Hopefully, you see that the output displayed is meant to be shape labelled F. How could the code shown be made more modular? (*Hint:* put lines 13-15 in a separate function.)

Use the four functions provided to draw shapes N, P, W, Y and Z.

Is there any shape you cannot draw? If so, write the code!

2. Use the code on the left hand side to draw as many of these shapes as possible.

```
1  # Tetris blocks
2
3  def draw3Down():
4      print("*")
5      print("*")
6      print("*")
7
8  def draw3Across():
9      print("***")
10
11 def draw2AcrossL():
12     print("**")
13
14 def draw2AcrossR():
15     print(" **")
16
17 def draw1Left():
18     print("*")
19
20 def draw1Right():
21     print("  *")
22
23 def draw1Middle():
24     print(" *")
25
```



3. Write programs to display the following (individual) patterns.

i)
```
#######
#
#
#
#######
```

j)
```
#######
#
#######
#
#######
```

k)
```
 #######
#
 #######
#
#
```

l)
```
#       #
#       #
 #######
#       #
#       #
```

m)
```
#
#
#
#
#######
```

n)
```
#######
#       #
#       #
#       #
#######
```

o)
```
#######
    #
    #
    #
    #
```

p)
```
#       #
#       #
#       #
#       #
 #######
```

By using the code you just wrote, write a program that outputs the letters LOFT (one letter underneath the next).

Can you make any more letters?

The code on the left hand side below displays the shapes displayed on the right hand side.

```
 1 print("  ______")
 2 print(" /      \\")
 3 print("/        \\")
 4 print("\\        /")
 5 print(" \\______/")
 6 print("")
 7 print("\\        /")
 8 print(" \\______/")
 9 print("+--------+")
10 print()
11 print("  ______")
12 print(" /      \\")
13 print("/        \\")
14 print("|  STOP  |")
15 print("\\        /")
16 print(" \\______/")
17 print("")
18 print("  ______")
19 print(" /      \\")
20 print("/        \\")
21 print("+--------+");
22
```

Program to display shapes.

```
  ______
 /      \
/        \
\        /
 \______/

\        /
 \______/
+--------+

  ______
 /      \
/        \
|  STOP  |
\        /
 \______/

  ______
 /      \
/        \
+--------+
```

Program output

Read the notes below and then type the above program in. Make sure to save (call it **Shapes1.py**) and test as you go.

Notes:

You need to be careful with the amount of leading spaces you use on lines 1, 2, 5 and 6

The character used in the print statement on line 1 is an underscore

In order to print a single backslash (‘\’) character you must precede it with another backslash inside the string (i.e. ‘\\’). This is known as an *escape sequence*.

Once you have the above program working improve the design as follows:

a) Make the program more modular.
Hint: You will need to structure the program so that it has four functions as follows:
a function to display the egg shape
a function to display the teacup shape
a function to display the stop sign
a function to display the hat

Call this program **Shapes2.py**.

b) Remove duplication.
Hint: You will need to examine the shapes very closely and look for patterns that occur more than once. Write a separate function to display each re-usable part. Your functions to display the actual shapes will call the functions that display the parts of the shape. Call this program **Shapes3.py**.

Write a full modular program to draw the following patterns (used by the game ‘hangman’)

	
Diagram 1	**Diagram 2**
	
Diagram 3	**Diagram 4**
	
Diagram 5	**Diagram 6**
	
Diagram 7	

Use your program to display the patterns in the sequence shown.

Summary / Key Points

A function is a piece of re-usable code that performs a specific task (as such they do not normally contain too many lines)

Functions are the building blocks of programs. A modular program is usually made up of many functions. If used properly, programmers can use functions to avoid code duplication.

When you write more than one function in your program use blank lines to separate them from each other. This makes your program more attractive and easier to read.

The code to define a function must appear before the code that calls it.

A function definition is made up of a function header (aka signature) and function body.

The function header must start with the Python reserved word, **`def`** followed by the name of the function, followed by brackets and finally a colon

Function names cannot contain spaces or begin with a number

Function names must be unique i.e. no two functions can have the same name

Function names should be meaningful i.e. they should in some way describe what the function does

As a matter of style I usually use a lower case letter for the first letter in a function name. Any full words contained in the function name should start with a capital letter

In Python, the lines of code in the function body must be indented.

A function is called (invoked) in a program simply by using its name (followed by brackets)

When a function is called the flow of control jumps to the first line of the function. Execution continues by running the function body. Once the body has been executed, the flow of control is returned to the line where the function was called in the first place

Functions can call other functions
Some functions are **built-in** as part of Python (e.g. `print` and `input`) and some functions are defined by the programmer. **User-defined functions** can be thought of as special commands developed by programmers for use in their programs.
There's a lot more to learn about functions but that's enough for the momen

Lesson 3

Variables, Assignments and Expressions

INTRODUCTION

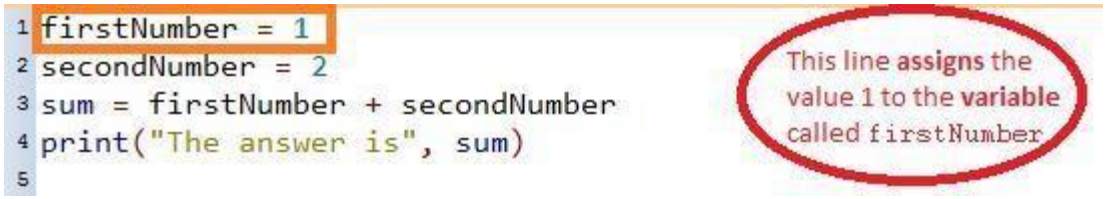
Question: Can you guess what this short program does?

```
firstNumber = 1
secondNumber = 2
sum = firstNumber + secondNumber
print("The answer is", sum)
```

Answer: The program adds the numbers 1 and 2 and displays the result in a message.

We will now *walk through* the program - line by line. We start with line 1.

FIGURE 3.1: firstNumber - Variable



The variable `firstNumber` is declared

The name of the variable is chosen by the programmer (you!)

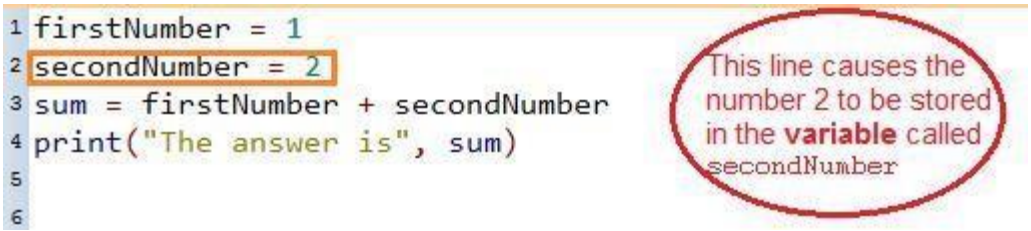
Variables are used to remember data – they are memory locations

= is the Python **assignment operator**

The value on the right hand side is stored in the variable on the left hand side

Let's move on to line 2.

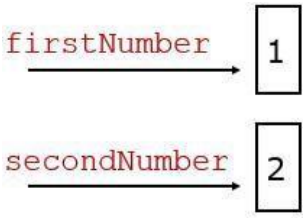
FIGURE 3.2: secondNumber - Variable



The variable `secondNumber` is declared and initialised to 2. After

executing line 2 the computer's memory looks like this:

FIGURE 3.3: Initialize firstNumber and secondNumber

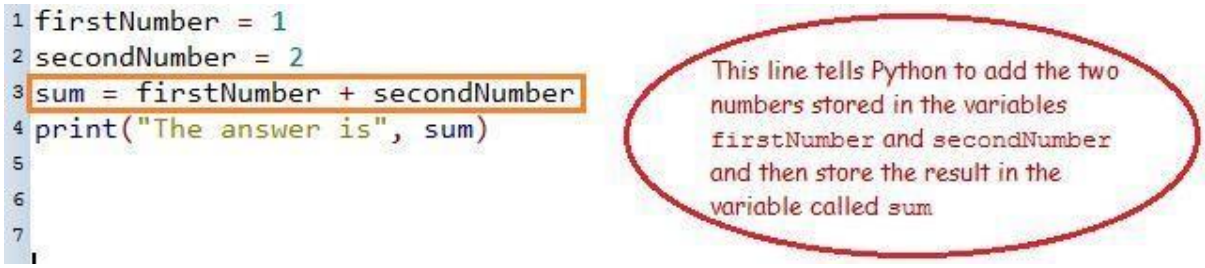


At this stage the programmer has declared and initialised two variables - `firstNumber` and `secondNumber`.

Both variables are now know to Python and therefore can be used in subsequent lines of the program.

We now look at line 3.

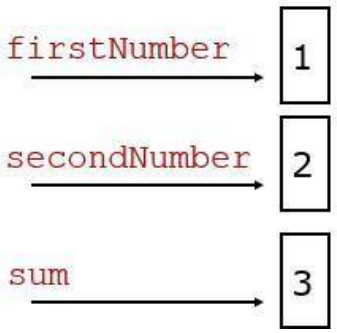
FIGURE 3.4: Add them both!



Python always evaluates the right hand side of an assignment statement first
In this case Python evaluates `firstNumber + secondNumber` and gets 3.
The result of the evaluation is stored in the variable named on the left hand side
So, 3 is stored in `sum`.

Python's memory for the program now looks like this:

FIGURE 3.5: Include the sum



The right hand part of an assignment is called an **expression**. Some expressions are literal values (as in lines 1 and 2) and some expressions are arithmetic (as in line 3).

In line 3 `firstNumber + secondNumber` is an arithmetic expression. Python can deal with it because both variables are already declared and + is a legal operator. Other arithmetic operators are shown below.

Operator	Operation
	Addition
	Subtraction
	Multiplication
	Division

Finally, Python reaches line 4.

FIGURE 3.6: Print with Arithmetic Operators

```
print("The answer is", sum)
```

This line is a print statement and it causes the message, *The answer is 3* to be displayed.

Notice that the variable `sum` is not included between the quotes and that the value of the variable (i.e. 3) is displayed, in place of the variable name, in the output.

Review

The program below adds the numbers 1 and 2 and displays the result in a message.

```
firstNumber = 1
secondNumber = 2
sum = firstNumber + secondNumber
print("The answer is", sum)
```

The program is useful – not because it adds 1 and 2 (we already know how to do that!) – but, because it teaches us the concepts of variables, assignments and expressions.

The program uses 3 variables – `firstNumber, secondNumber` and `sum`
Lines 1, 2 and 3 are all assignment statements. Remember the expression is on the right hand side and the variable is on the left.
Lines 1 and 2 use *literal expressions*, whereas line 3 uses an *arithmetic expression* (i.e. addition)

Line 4 is a `print` statement

Finally, it is worth noting that the following four programs do exactly the same thing. In other words, these programs are *logically equivalent*.

(FIGURE 3.7 – FIGURE 3.10)

```
1 firstNumber = 1
2 secondNumber = 2
3 sum = firstNumber + secondNumber
4 print("The answer is", sum)
```

Figure 7: PROGRAM 1

```
1 n1 = 1
2 n2 = 2
3 result = n1 + n2
4 print("The answer is", result)
```

Figure8: *PROGRAM 2*

```
1 sum = 1 + 2
2 print("The answer is", sum)
```

Figure9:PROGRAM 3

```
1 print("The answer is", 1+2)
```

Figure10:PROGRAM 4

Look and Learn!
Study the two examples below carefully.

FIGURE 3.11:
print(message)

1.
```
1 message = "Hello World!"
2 print(message)
```

Line 1 declares a variable called `message` and assigns the string *Hello World!* to it. Line 2 displays the contents of the variable `message` on the screen.

FIGURE 3.12: Display the time

2.
```
1 days = 2
2 hours = 24
3 mins = 60
4 totalMins=(days*hours*mins)
5 print("There are ", totalMins, "minutes in 2 days")
```

This program computes and displays the total number of minutes in 2 days

The second example uses 4 variables – `days`, `hours`, `mins` and `totalMins` – and 4 assignment statements. Line 5 is a `print` statement.

Rules for naming variables

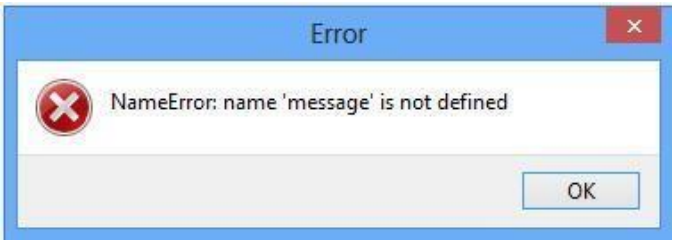
It is up to the programmer to decide when a variable is needed and also what to name it. Certain basic rules for naming variables apply. (The same rules apply to naming functions.)

A variable cannot be a word already reserved by Python (e.g. "`print`" "`def`", etc.)
Variable names must contain only letters, digits, and the underscore character, _.
Variable names cannot have a digit for the first character.
Spaces or dots are not allowed in a variable name

Variables must be declared before they can be used in an expression.

The following syntax error appears when Python comes across a variable it does not understand. (In this case, the name of the unknown variable is `'message'`.)

FIGURE 3.13: NameError notice!



Remember that Python is case-sensitive. For example, the variable names 'ISBN' and 'isbn' are different to Python even though both names are made up of the same letters (in the same order). Forgetting that variable names are case sensitive is a very common cause of error to the novice programmer.

Programmer Tips

Choose meaningful names for your variables. A meaningful name is one that tells something about what the variable is used for.

Capitalise interior words in multi-word variable names, e.g. `lastName`, `firstName`, `stockCount`, `barCode`, and `payRate` are all good variable names.

Review Exercise
Which of the following are legal variable names?

```
student.Number
x
1x
x1
input
number
21
h21
PPSN
ppsn
person name
address
date_of_birth
2+4
```

Exercises 1
Answer the following using pen and paper – no computer necessary!

It is up to the programmer to decide when a variable is needed and what to call it. It is important that variables are given meaningful names. Take a look at the scenarios below and suggest an appropriate variable name for each scenario. A variable that stores….

Scenario	Suggested variable name
ix.a person's name	personName (pName, name)
a person's date of birth	
an employee's salary	
a student mark/grade	
a bank account balance	
a user option (on a menu system)	
someone's password	
the number of times a user has attempted to login to a system	
the number of lives left a player has on a game	
a player's score	

A variable is used to store data. As a programmer you need to understand what data your program needs to process and what variables you need to use.
Identify some data/variables that might be needed by the systems listed below

System	Data used in system/Variables
ATM system	PIN, option, account number, balance, amount requested
Airline Reservation System	
College Application System	
Point of Sale (Retail) System	
Facebook	
Amazon	
Calculator Application	

Choose any variable you identified and explain a situation that might necessitate the value for the variable to change?

Read carefully the following block of code and answer the questions which follow:

```
answer=1+2
print(answer)
value1=answer+3
value2=1+2+3
print(value1, value2)
```

How many variables are used?
What are the names of the variables?
What lines are the assignment statements on?
What output does line 2 display?
What output does line 5 display?

Write a line code to do each of the following:

create a variable called `name`*, initialised to your own full name*

display the contents of the variable

```
name = "Joe Bloggs"
```

```
print("The value in the variable called name is ",name)
```

Write a line code to do each of the following:

create a variable called `firstName`*, initialised to your own first name*

create a variable called `lastName`*, initialised to your own surname*

display the contents of the variables in a meaningful output message

---

Write a line code to do each of the following:

create a variable called `result`*, initialised to 50*

display the value of `result`

Write a line code to do each of the following:

create a variable called `number`*, initialised to 0*
assign the value 100 to `number`
display the value of `number`

Read carefully the following block of code and answer the questions which follow:

```
1. goals = 0
2. goals = goals + 1
3. print("The value of goals is", goals)
```

How many assignment statements are there?
What is the value of `goals` *after line 2 is executed?*
What message is displayed?

Write a line code to do each of the following:

create a variable called `livesLeft`*, initialised to 3*
subtract 1 from `livesLeft` *and assign the result to* `livesLeft`
display the value of `livesLeft`

Read carefully the following block of code and answer the questions which follow:

```
1. a=10
2. b=5
3. temp=a
4. a=b
5. b=temp
```

How many variables are there? (What are their names?)
What are their initial values?
What are the values of `a`*,* `b` *and* `temp` *at the end of the program?*
Explain – in one sentence - what the program does?
What is the purpose of the variable `temp`*?*

Write code to declare variables as follows:

`op1`, *initialised to 20*
`op2`, *initialised to 5*
`sum` *initialised to the sum of* `op1` *and* `op2`
`difference` *initialised to the difference between* `op1` *and* `op2`
`product` *initialised to the product of* `op1` *and* `op2`
`quotient` *initialised to the quotient of* `op1` *and* `op2`

Now write the code to display the values of `sum, difference, product` *and* `quotient`

Imagine a small banking application. What effect do you think the following fragment of code has on the value held in the variable `accountBalance`*?*

```
accountBalance=1000
withdrawalAmount=600
accountBalance=accountBalance-withdrawalAmount
```

Let us say a banking application contains a variable called `accountBalance` *which has a value of €1000. Write a statement (i.e. a line of code) to do the following:*

Increase the balance by €200
Increase the balance by a further €100

LEARNING OBJECTIVES

1. **Remembering:** Define key terms such as variable, assignment, expression, and operator.
2. **Understanding:** Describe the process of variable assignment and the role of the assignment operator.
3. **Analyzing:** Differentiate between valid and invalid variable names according to Python naming rules.

LECTURE DISCUSSION

Lesson: Basic Arithmetic

In this lesson we will learn more about how programs evaluate expressions. The first rule of thumb is that Python evaluates expressions using the normal rules of precedence. (Recall BIRDMAS?). Consider the following code snippet.

```
x = 2+3*4
print(x)
y = (2+3)*4
print(y)
```

Line 2 displays 14 because multiplication is carried out before addition. Line 4 displays 20 because the brackets forces the addition to be carried out before the multiplication.

Variable Substitution

Given two variables, **`x=2`**, and **`y=8`** what do the Python expressions shown below evaluate to?

```
x+y
x-y
5*y
5*x+y
3*(x+y)
x*y+4
y/x+1
y/x+y
(x*y)+(y/x)
```

Simply substitute 2 in for **`x`** and 8 in for **`y`**. Once we do this and follow the normal precedence rules of arithmetic we can be confident of arriving at the correct answer.

Solution

This is just a simple addition, $2 + 8$. The answer is 10.
This is a subtraction, $2 - 8$. We just need to be careful about the sign. The answer is -6.
5 times 8. The answer is 40.
This line contains two operations – multiplication and addition. Since the multiplication has a higher precedence than addition we multiply 5 by 2 first. This gives 10 which we can now add to 8. The answer is 18.
The brackets here forces the addition to be done first. We get 10. This is then multiplied by 3 to give an answer of 30.

This line is evaluated from left to right. 2 is multiplied by 8 to give 16. This is then added to 4 to give an answer of 20.
Here 2 to be divided into 8 first. This gives 4 which is then added with 1. The answer is 5.
Again 8 is divided by 2 to give 4, which is then added with 8. The answer is 12.
The expressions inside the brackets are evaluated first. The multiplication evaluates to 16 and the division evaluates to 4. Both results are added to give an answer of 20.

FIGURE 3.14: Look and Learn!

Look and Learn!

```
1 minutes=60
2 texts=389
3 cost=20+(0.25*minutes)+(0.1*texts)
4 print("The total monthly charge is", cost)
5
```

This code calculates a phone bill based on a charge of 25 cents per minute for calls and 10 cents per minute for texts.
An initial charge of €20 is also added.

Evaluating Formulae – Temperature Conversion

Consider the problem – write a program to convert 20° Centigrade into Fahrenheit. The formula to convert from degrees Centigrade into Fahrenheit is,

$$= \frac{9}{5} + 32$$

where, stands for the temperature in Centigrade and stands for the converted value in Fahrenheit.

We can now design our solution around this formula using the following pseudo-code:

Initialise the value of variable c to 20 (this is the value to be converted)
Apply the conversion formula – this can be broken down as follows:
Divide 9 by 5
Multiply the result by the value of 'c' Add 32
The result is then assigned to the variable 'f'
Display the answer

The Python implementation looks like this:

```
c=20
f=9/5*c+32
print("There are ", f, "degrees Fahrenheit in", c, "degrees Centigrade")
```

As an exercise modify the above program so that it converts 100° Centigrade into Fahrenheit.

Note: It is relatively straightforward to write a program to perform a calculation *once we know a formula*. Very often the task of formulating the formula is more difficult than the actual task of writing a program to use it.

Incrementing and Decrementing

These terms mean increasing or decreasing the value of a variable by a certain amount – usually 1.

For example, the following line has the effect of increasing the value stored in variable `score` by one.

```
score = score+1
```

Therefore, if the value of `score` before the line was executed was 6, it would be 7 afterwards. Notice that the variable appears on both sides of the assignment statement.

Similarly, the following line reduces the value stored in the variable `livesLeft` by 1.

```
livesLeft = livesLeft-1
```

Exercises 2

1. Explain what each of the following five code fragments do:

CODE FRAGMENT 1

```
print(2+3)
```

CODE FRAGMENT 2

```
print ("2+3
  =",2+3)
```

CODE FRAGMENT 3

```
print       ("2+3
=",23)
```

CODE FRAGMENT 4

```
a = 2
b = 3
print(a,"+",b,"=",a+
b)
```

CODE FRAGMENT 5

```
num1 = 2
num2 = 3
print(num1,"+",num2,"=",num1+nu
m2)
```

Given the variables **x=2,** and **y=6** what do the Python expressions shown below evaluate to?

```
x+y
x-y
5*y
5*x+y
3*(x+y)
x*y+4
y/x+1
y/x+y
(x*y)+(y/x)
```

Write lines of code to carry out the following calculations.

74*64
81/10
2*3.14*5
10*50/5
10*(50/5)

In all cases you should store your answer in an appropriately named variable and then display the value of the variable in a meaningful message on a separate line. So for example the answer to part a) would be:

```
answer=74*64
print("74*64=",answer)
```

Write a single line of code that adds the numbers 62 and 47 and displays the result.

Write a small program that does the following:
Adds the numbers 62 and 47 and stores the answer in a variable
Display the content of the variable

The average (arithmetic mean) of two numbers is calculated by adding them up and dividing by 2. Write a program to calculate and display the average of 62 and 47.

Given the following code fragment that initialises three variables - `x`, `y` and `z` – insert a line of code (at the end) that calculates and prints out their average.

```
x=27
y=15
z=18
```

4.

What do you think the following programs would output:

```
x=3*4
y=10/2
z=6-1
sum=x+y+z
print(x, y, z, sum)
```

What (if anything) is wrong with the following? (Answer on a line by line basis.)

```
a) x=1+2
b) 3=1+2
c) 10/2=5
d)
sum=a+b+c
e) print(a, b, c,
sum) f) a=8*2
g)
b=8(+2)
h) c=4*a
i) c=4a
```

The area of a rectangle can be computed by multiplying its width by its height. Given the following two lines of code to initialise these two variables write a third line that computes the rectangle's area. (Write a 4th line to display the answer.)

```
   width=7
   height=5
3.
```

The perimeter of a rectangle can be computed by adding twice the width to twice its height. Given the following two lines of code which initialise these two variables, write a third line that computes the rectangle's perimeter. (Write a 4th line to display the answer.)

```
   width=7
   height=5
3.
```

Given that a person's pay is calculated by multiplying the number of hours they work by their hourly rate, write a program to calculate how much a person on €13.50 an hour would earn for working a 40 hour week. Use appropriate variable names and display the result.

Extend the program just developed to deduct 20% PAYE, and 5% PRSI. Again, use appropriate variable names and display the result (i.e. net wage).
Given the following formula to convert Fahrenheit ( ) to Centigrade ( ) write a program to convert $100°F$ into $°C$ and display the result.

$$=(-32)\times\frac{5}{9}$$

Read the following code carefully (the values are in cents) and answer the questions that follow:

```
   fifties=4
   twenties=5
   tens =7
   total=fifties*.5 + twenties*.2 +tens*.1
   print(total)
```

How many variables does the program use?
What is the purpose of each variable?
What does the code do?

Let us say I have 18 fifty cent coins, 14 twenty cent coins and 15 ten cents. Write a program to calculate the total number of euros I have.

Now modify the above program to calculate the number of euros given that I have a five euro note, 22 fifty cent coins, 17 twenty cent coins, 25 ten cents and 13, 2 cent pieces.

Write a program to calculate and display the total from the bill below. Hint: Just re-arrange the lines of code shown below:

5 x mars bars @ €1 each
4 x cans of coke @ €1.50 each
3 x bags of crisps @ 80 cents each
2 x cups of tea @ €2 euro each
1 x slice pan @ €3.50 each

```
costOfCoke = 4 * 1.5
costOfCrisps = qtyCrisps * unitCostOfCrisps
print("The total cost is", total)
qtyCrisps = 3
costOfBread = 3.50
qtyMars = 5
total = costOfMars+costOfCoke+costOfCrisps+costOfTea+costOfBread
costOfTea = 2 * 2
unitCostOfCrisps = 0.8
costOfMars = qtyMars * 1
```

And finally, can you write programs to solve the following problems?

The life expectancy for a man is 78 years and the life expectancy for a woman is 82 years. Write a program to compute and display the average life expectancy.

Jim was born in 1997. Write a program to display what year it will be when he turns 19.

Jack was born in 2002. Write a program to display what age he will be in 2064

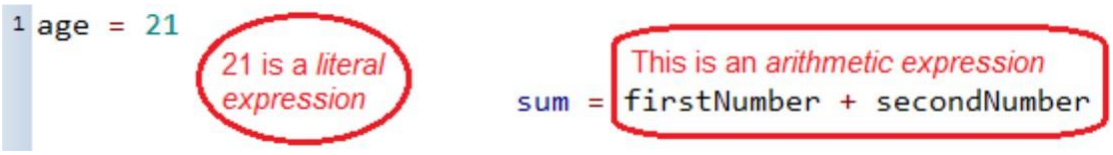
The year is 2053. Mary has just celebrated her 87th birthday. Write a program to display
Mary's year of birth.

The sum of the first n numbers can be calculated using the formula (+)/ .
Write a program that computes the sum of the first 100 numbers.

Lesson: Input variables and the `input` command

So far variables have been assigned values resulting from either *literal expressions* or *arithmetic expressions*.

FIGURE 3.15: Arithmetic Expression



This lesson examines how variables can also be assigned values from the end-user. This is called **user input**. The Python `input` command allows us to do just that.

The `input` command allows a user to enter a value into a running program and have that value stored in a variable.

Consider the following short program that asks a user to enter their name and then prints out a greeting message using whatever name the user enters.

```
1 personName = input("Please enter your name")
2 print("Nice to meet you", personName)
```

Line 1 causes the message box shown below to be displayed as a *prompt* to the end-user.
Notice that the string used in the input command appears as a prompt in the message box.

FIGURE 3.16: Enter your name…

Once the user enters a value and presses <OK> the value entered is assigned to the variable `personName`. So, for example, if the user enters *Joe*, the value *Joe* will be assigned to the variable `personName`.

FIGURE 3.17: Joe!

Line 2 is very straightforward – it just prints a message *Nice to meet* you followed by the person's name e.g. *Nice to meet you Joe*.

Notice how the value stored in the variable is substituted for the variable name used in the `print` command.

Take your time to understand this as there is a lot going on here!

KEY POINT: Use the **`input`** command to assign text entered by a user to a variable

Exercises 3

Describe what do you think the following program does?

```
colour=input("Please enter your favourite colour")
print("Your favourite colour is", colour)
```

Key in the following code. What does it do?

```
firstName=input("What is your name?")
colour=input("What is your favourite colour?")
print("Hi", firstName, "Your favourite colour is",
colour)
```

Modify the code in the previous question so that it asks the user for their surname as well as their first name.

Complete the following:

Write a line of code that asks a user how many brothers they have. Store the value entered in a variable called `brothers`.

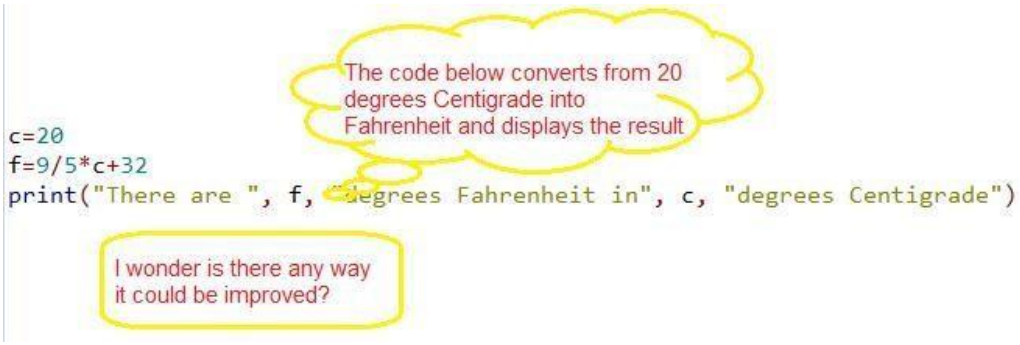
Now write a line of code that asks a user how many sisters they have. Store the value entered in a variable called `sisters`

Finally, write a line that uses the values entered in parts a) and b) to output a message like, *You have 2 brothers and 3 sisters*

Lesson: Reading numbers from the end-user

Recall from earlier an example program to convert from 20℃ to Fahrenheit

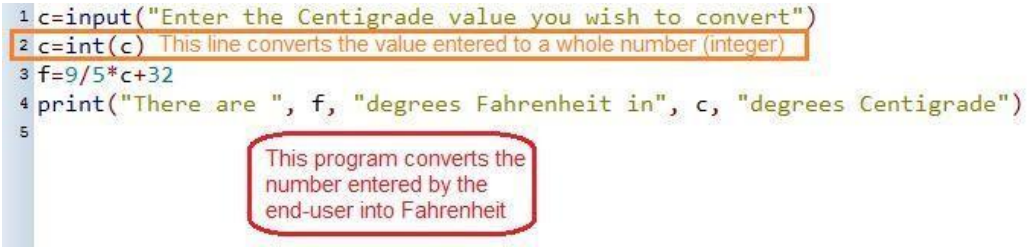
FIGURE 3.18: Convert the temperature!



Imagine if our program could ask the user to enter *any* Centigrade value that they would like to convert. The user could enter *any* number and the program would output the Fahrenheit equivalent.

Our solution makes use of the `input` command as follows:

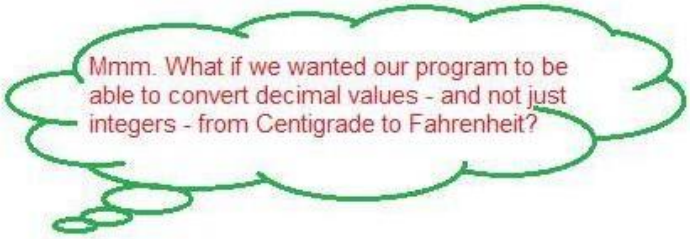
FIGURE 3.19: input()



The above solution is known as a *general solution* because it would work for any number.
General solutions are much more useful than solutions that only work for specific values.

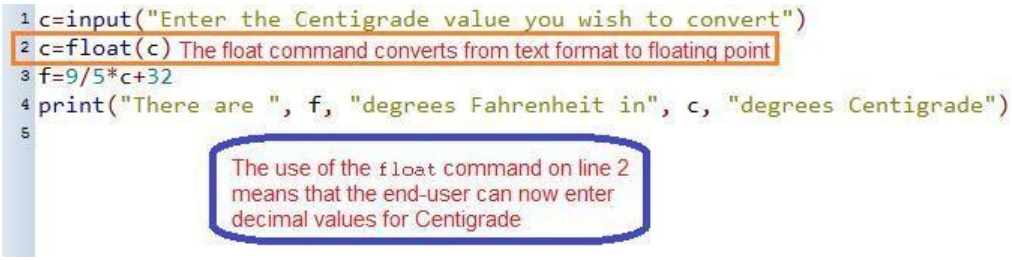
Line 2 is important because it converts the value entered by the end-user from text into a whole number. In Python, whole numbers are called **integers**.

FIGURE 3.20: Statement about temperature



A slightly better solution would be to make one change so that the user can enter numbers with decimal points. In Python, these are called **floating point numbers**.

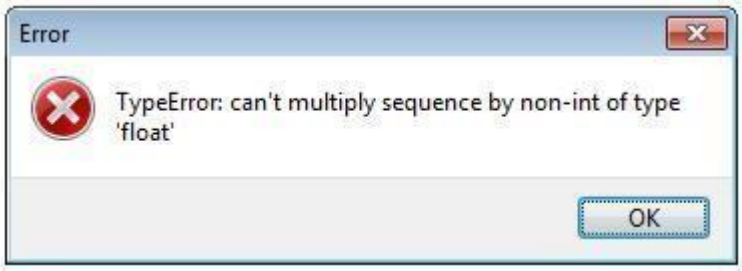
FIGURE 3.21: float()



The two commands `int` and `float` are important because they tell Python to treat the value entered as a number. This means that they can be used in arithmetic expressions.

The following syntax error is displayed if you attempt to use a text value in an arithmetic expressions.

FIGURE 3.22: TypeError Notice! (Arithmetic Expressions)



This is called a type error. To correct this error you will need to use the `int` or `float` commands to convert the text to a numeric type.

The concept of *datatype* is very important in computer programming. The basic idea is that programmers need to be aware of the type of data their program is dealing

with. The most common types are – strings (text), integers (int), floating point numbers (float) and Boolean (true/false).

A note on testing

Testing is an important aspect of developing software. Simply put testing means to check or verify that the program does what it is meant to do. Before you commence testing your program you should always devise a number of test cases. Each test case is made up of an input and an expected output. When the test case is run the actual output should be recorded. If there is a difference between the expected and actual output then your program contains an error (or bug) that you need to fix. A comprehensive test will ensure that every line of code is executed. It will also take 'abnormal' scenarios into consideration

The table below provides a good basis for testing the Centigrade to Fahrenheit functionality in the above program. Each row in the table is known as a test case.

Sample Input (°C)	Expected Output (°F)	Actual Output
0	32	
32	89.6	
1000	1832	
-10	14	
-40	-40	
-45	-49	
-1000	-1768	

As an exercise you should key in the temperature conversion program above and fill in the Actual Output column on the table shown here.

Note that the values in the sample input column are arbitrarily chosen. The expected output values are calculated by hand/using a calculator or come from some other source e.g. internet. The values in the actual output column should be recorded by running the program (see exercise). Any deviations between expected and actual output is an indication of an error (bug) and usually necessitates some change to the source code. Once the change(s) have been made the all test cases should be run again. When the program passes all test cases it is said to be *unit tested*.

Look and Learn!

Simple interest is calculated using the following formula

FIGURE 3.23: Interest calculation

$$= \times \times$$

```
# Simple Interest calculation
amount = 1000*0.1*5

# Display the answer
print("The interest amount is", amount)
```

Calculate and display the simple interest earned on €1000 at 10% over 5 years

A nicer solution would make use of variables….

FIGURE 3.24: Declare 3 variables

```
# Declare 3 variables
principal = 1000
rate = 0.1
time = 5

# Simple Interest calculation
amount = principal*rate*time

# Display the answer
print("The interest amount is", amount)
```

OK. This is better but.....
Wouldn't it be nice if we could let the user decide what values to use for principal, rate and time?

Values are no longer hard-coded - the bells and whistle solution!

FIGURE 3.25: Ask for the principal!

```
# Ask the user for the principal
principal=input("Enter principal")
principal=float(principal)

# Ask the user for the interest rate
rate=input("Enter rate")
rate=float(rate)

# Ask the user for the length of time
time=input("Enter time in years")
time=float(time)

# Simple Interest calculation
amount = principal*rate*time

# Display the answer
print("The interest amount is", amount)
```

Extend the above simple interest calculation program to calculate and display the new principle. (*Hint:* You will need to add the interest earned to the original principal.)

How might you test the above program? (see exercises 4, question 7)

Experiment!

One of the best ways to learn is just to experiment. Don't be afraid to make mistakes.

1. Can you understand the following?

FIGURE 3.26: 'x' and 'y' variables

```
x=5
y=x
x=10
```

A simple trick to remember a value. The value in x is saved in variable y before it is changed (overwritten)

After executing the above three lines…

what is the value of x?

what is the value of y?

The 4 programs below all use the formula $= 2$ to compute the length ( ) of the circumference of a circle of radius . The value of is entered by the user. (The programs assume $= 3.14$.)

Study the 3 programs carefully – type them in and run them if you have time.

FIGURE 3.27: 'r' and 'l' variables

```
r = input("Enter the radius")
r = float(r)
l = 2 * 3.14 * r
print("The circumference length is", l, "units")
```

FIGURE 3.28: Work with pi!

```
pi = 3.14
r = input("Enter the radius")
r = float(r)
l = 2 * pi * r
print("The circumference length is", l, "units")
```

```
pi = 3.14
r = input("Enter the radius")
r = float(r)
print("The circumference length is", 2*pi*r, "units")
```

```
pi = 3.14
r = float(input("Enter the radius"))
print("The circumference length is", 2*pi*r, "units")
```

What are the main differences?

What would happen if the `float` command was not used?

Exercises 4

Write a program that displays the sum of two numbers entered by the user. The solution is broken down as follows:

prompts the user to enter a number and store its value in a variable (e.g. `number1`)
convert `number1` to an integer
prompts the user to enter another number and store its value
convert `number2` to an integer
computes the sum of the two integers
displays the answer in an appropriate message

Hint: You need to re-arrange the lines of code displayed below into the correct order (but be warned there is one extra line given that you don't need)

```
number2 = input("Enter a value:")
sum = number1 + number2
print("The answer is", sum)
number1 = int(number1)
print("The answer is sum")
number2 = int(number2)
number1 = input("Enter a value:")
```

Modify the program you just wrote to compute and display the difference, product and quotient of the two numbers entered.

The average (arithmetic mean) of two numbers is calculated by adding them up and dividing by 2. Write a program to ask a user to enter two numbers, compute their average and display the result.

Given two types of scores – goals and points – where a goal counts for 3 points, write a program that prompts a user to enter (in two steps)
the number of goals scored and
the number of points scored.
The program uses the data input to calculate the total points scored. (For example, if a user enters 4 goals and 10 points the program will output 22 points)

The area of a circle is given by the formula $= ^2$. Write a program that prompts a user to input a value for a radius (r), and then calculates circle area. The program should also print out the result. (Assume $= 3.14$.)

How would the program in the previous exercise have to be changed if the requirement was to ask for the diameter (as opposed to the radius)?

7. Key in the simple interest program shown earlier.
Test your program using the sample input in the table below. Use a calculator to work out the expected output. Use your program to tell you the actual output. Complete the table

Sample Input			Expected Output		Actual Output	
principal (€)	**rate (%)**	**time (years)**	**INTEREST**	**NEW PRINCIAL**	**INTEREST**	**NEW PRINCIAL**
1000	0.1	1	100.00	1100.00		
1000	0.1	5	500.00	1500.00		
1000	0.1	10				
1000	0.05	10				
5000	0.1	10				

8. A person's basic gross wages is calculated based on an hourly rate for a standard 40 hour week. In addition, overtime is paid at a rate of 'time and a half' for the first 10 hours and 'double time' thereafter. Deductions include PAYE (20%), PRSI (5%) and USC (2%). Write a program that prompts the end-user (e.g. a wages clerk) to enter the number of hours worked and then outputs the following information:

a) the employee's basic gross earnings (based on 40 hours)
b) the employee's 'time and a half' overtime earnings (or zero, if none) c) the employee's 'double time' overtime earnings (or zero, if none) d) the employee's total gross pay
e) PAYE deductions
f) PRSI deductions
g) USC deductions
h) the employee's total deductions
i) the employee's net (take home) pay.

9. Two points in a plane are specified using the coordinates (x1,y1) and (x2,y2). Write a program that uses the formula below to calculate (and display) the slope of a line through
two points entered by the user.

$$\frac{}{} = \frac{2-1}{2-1}$$

Why is testing such an important aspect of developing software? As an exercise devise and implement test cases for the previous program. Use the table below to assist you.

Sample Input				Expected Output	Actual Output
X1	Y1	X2	Y2		

Appendix

Summary of Python Operators

Operator	Name	Explanation	Examples
+	Plus	Adds the two objects	`3 + 5` gives 8. `'a' + 'b'` gives `'ab'`.
-	Minus	Either gives a negative number or gives the subtraction of one number from the other	`-5.2` gives a negative number. `50 - 24` gives 26.
*	Multiply	Gives the multiplication of the two numbers or returns the string repeated that many times.	`2 * 3` gives 6. `'la' * 3` gives `'lalala'`.
**	Power	Returns x to the power of y	`3 ** 4` gives 81 (i.e. `3 * 3 * 3 * 3`)
/	Divide	Divide x by y	`4 / 3` gives 1.3333333333333333.
//	Floor Division	Returns the floor of the quotient	`4 // 3` gives 1.
%	Modulo	Returns the remainder of the division	`8 % 3` gives 2. `-25.5 % 2.25` gives 1.5.

Common Built-in Functions

Function Name	Description
`print`	Displays text on the output console
`input`	Prompts the user to enter a value. The data entered is taken as a string
`int(x)`	Converts from a string to an integer
`float(x)`	Converts from a string to a floating point number
`round(x, n)`	Rounds a floating point number, `x`, to a specified number of decimal places given by `n`.
`pow(x,y)`	Calculates `x` to the power of `y`

Lesson 4

Functions (Part II)

LEARNING OBJECTIVES

1. **Remembering**: Recall the syntax and structure of a function in Python, including the components of a function definition and function call.
2. **Understanding**: Explain the purpose and benefits of using functions, including the concept of reusability and modularity in programming.
3. **Applying**: Write simple functions with parameters and arguments to perform specific tasks, and use these functions in a program.

LECTURE DISCUSSION

Revision of Functions

Let's recap on what we know about functions. Functions are the building blocks of programs. Programs are typically made up of many functions. Each function can be thought of as a sub-program designed to perform a specific a task.

Functions are useful because they are re-usable. This means they can save programmers from having to repeat the same lines of code every time they need a specific task carried out. Therefore, functions can be used to avoid duplication of code.

A function definition is made up of a header and a body.

The header is one line which starts with the keyword, `def` followed by the function name, followed by brackets, followed by a colon. The name of the function is chosen by the programmer. The rules for naming functions are the same as those for naming variables.

The function body usually contains several lines of code (i.e. statements). Remember that the code inside the function body must be indented.

From what we know so far, the general form of a function definition looks like this:

```
def functionName():
    FUNCTION BODY
```

Once a function has been defined it just takes one line of code to use it. This is the function call.

The general form of a function call look like this. (Note, the brackets are needed but `def` and colon are not).

```
functionName()
```

The code to define a function must appear before the code to call it in a Python program.

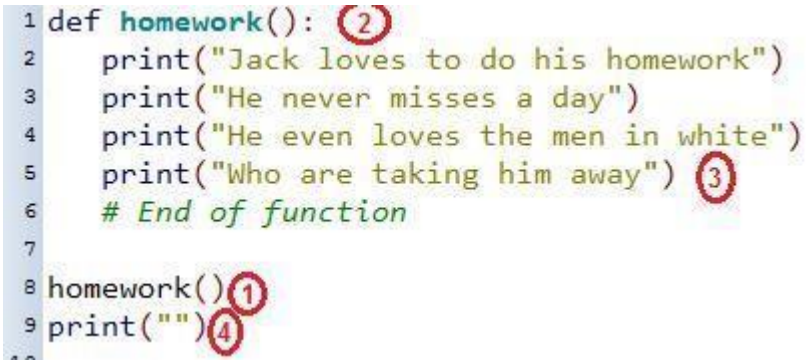
The semantics of a function call are explained below:

When a function is called

the flow of control jumps to the first line of the function (2) and execution continues until the function has ended

The flow of control then returns to the point where the call was made from (4).

FIGURE 4.1: Semantics of function call

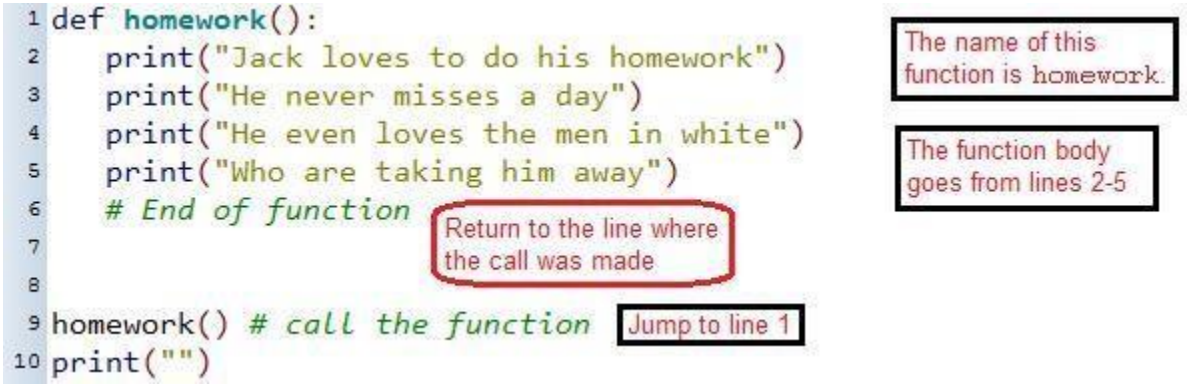


Functions make it possible to develop solutions piece by piece. Large scale software systems are developed by breaking big problems down into smaller problems. Each function is written to do a specific task. Such systems are said to be *modular*.

Example

The example below shows a definition of a function called `homework`.

FIGURE 4.2: def homework()



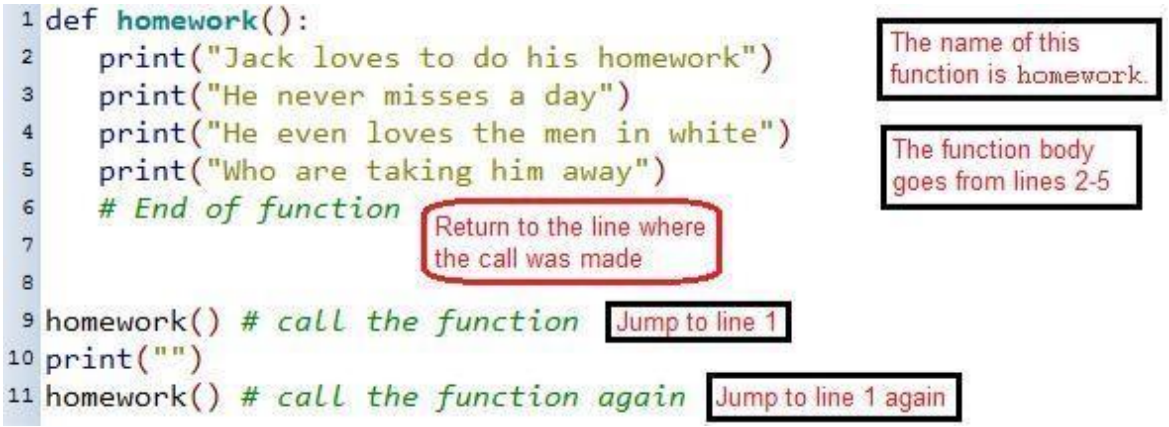
The program begins running from line 9 which calls the function and the following output is displayed:

```
Jack loves to do his homework
He never misses a day
He even loves the men in white
Who are taking him away
```

Once all the lines in the function body have been executed the program continues to run at line 10. Line 10 tells Python to display a blank line.

The function can be re-used simply by adding another call – see line 11 below.

FIGURE 4.3: Re-using functions



The output displayed now becomes:

```
Jack loves to do his homework
He never misses a day
He even loves the men in white
Who are taking him away

Jack loves to do his homework
He never misses a day
He even loves the men in white
Who are taking him away
```

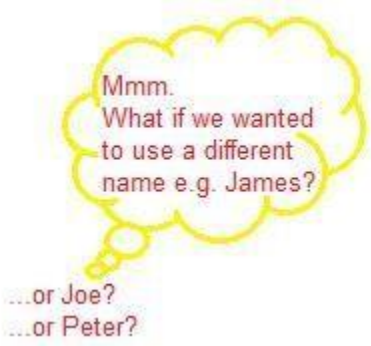


FIGURE 4.4: About Jack…

Function parameters

Notice that the same name is always displayed in previous example. Wouldn't it be nice if we could display the verse using different names?

Parameters allow programmers to pass information into a function. Take a look at this.

FIGURE 4.5: Function Parameters

```
1 def homework2(personName):
2     print(personName, "loves to do his homework")
3     print("He never misses a day")
4     print("He even loves the men in white")
5     print("Who are taking him away")
6     # End of function
7
8
9 homework2("James") # call the function
```

Notice that the name of the function is now `homework2`.

The output displayed is:

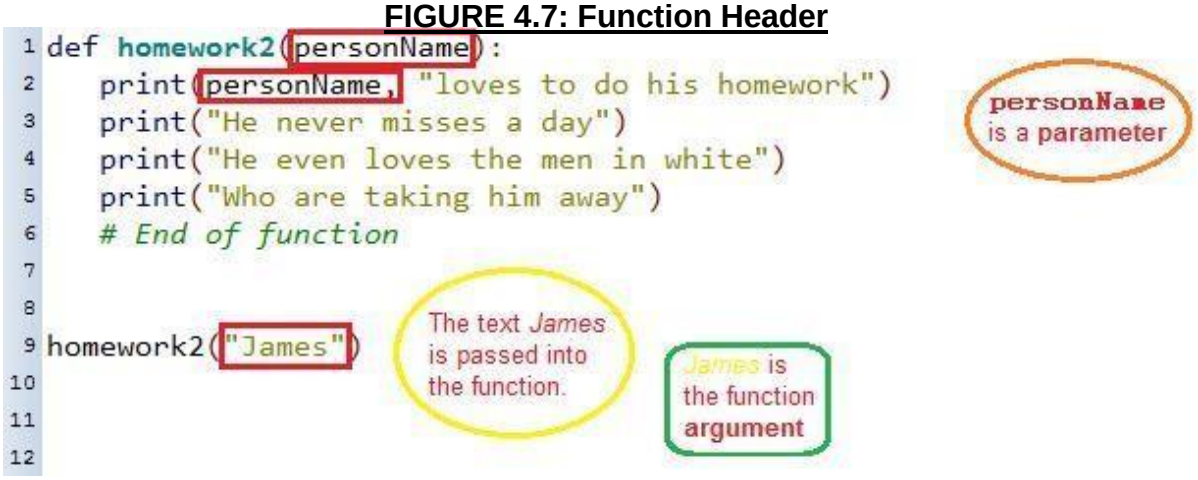
```
James loves to do his homework
He never misses a day
He even loves the men in white
Who are taking him away
```

Notice the poem is now about James (and not Jack)!

FIGURE 4.6: James and not Jack!

Let's take a closer look at the program:

FIGURE 4.7: Function Header



Notice `personName` between brackets in the function header? This is a function **parameter**.
A parameter is a special kind of variable used by functions.

Notice also the text *James* between brackets in the function call (line 9)? This is a function **argument**. Arguments are *passed into* functions.

The text *James* is passed into the function in the above example.

KEY POINT: A function parameter is a variable which gets it value from the argument passed in. When a function is called value of the argument is assigned to the parameter.

The parameter `personName` is assigned the value *James* when the function is called.

The value of `personName` is then used in the print statement on line 2. A parameter can be used anywhere inside the function body in the same way as you would use a variable.

The advantage of using parameters and arguments is that they make functions much more flexible. The behaviour of a function can be altered by passing different arguments into it.

This is demonstrated below where the three calls on the left hand side result in the output displayed on the right.

FIGURE 4.8: Passing different arguments

```
9  homework2("James")
10 print("")
11 homework2("Joe")
12 print("")
13 homework2("Fred")
```

```
James loves to do his homework
He never misses a day
He even loves the men in white
Who are taking him away

Joe loves to do his homework
He never misses a day
He even loves the men in white
Who are taking him away

Fred loves to do his homework
He never misses a day
He even loves the men in white
Who are taking him away
```

The three arguments used are the strings *James*, *Joe* and *Fred*.

At runtime each argument is received by the parameter, `personName` and the name is used in the output.

Test your understanding: Change the above code so that the output displayed would be as follows:

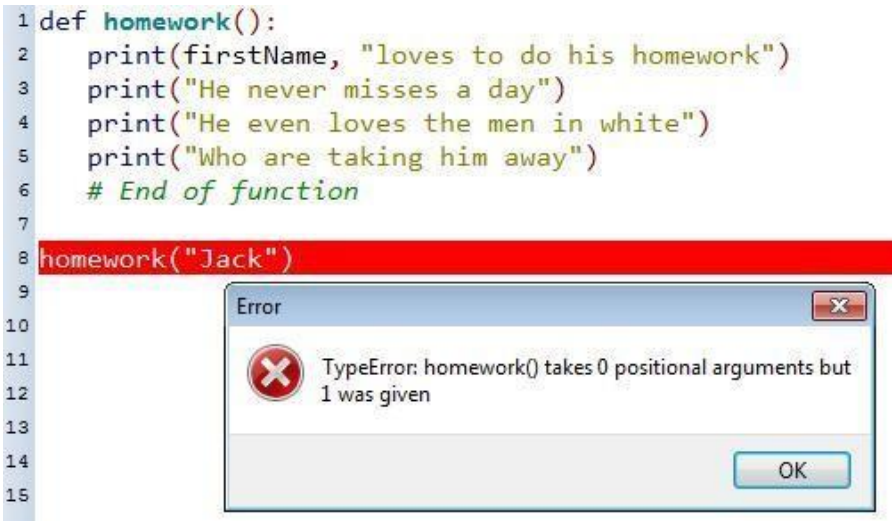
```
Pat loves to do his homework
He never misses a day
He even loves the men in white
Who are taking him away

Peter loves to do his homework
He never misses a day
He even loves the men in white
Who are taking him away

Paul loves to do his homework
He never misses a day
He even loves the men in white
Who are taking him away
```

Write your answer here:

Syntax Check

Programmers need to take care to match parameters and arguments. Take a close look at the two error screens shown below.

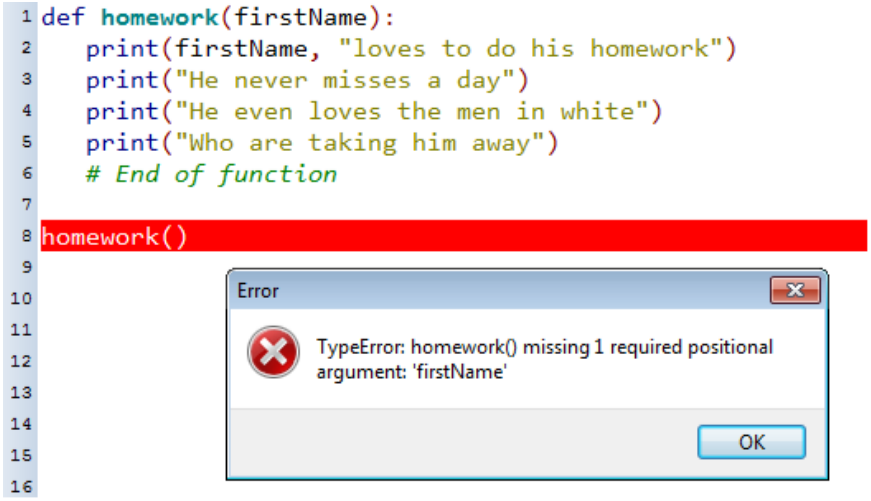
FIGURE 4.9: Syntax Check



The error is displayed because line 8 is attempting to call a function (i.e. `homework`) using *Jack* as an argument. However, there is no parameter specified inside brackets in line 1 of the function definition. This tells Python not to expect any arguments when the function is called.

Now look at the following. The parameter named on line 1 (i.e. `firstName`) tells Python to expect a value to be passed when the function is called. However, the function call – line 8 – does not specify an argument. Hence the error.

FIGURE 4.10: Error notice for homework()



KEY POINT: Python always expects the same number of arguments to be passed into the function as the number of parameters specified in the function definition.

Tips:

Experiment. Deliberately cause syntax errors.

Read each syntax error message carefully before fixing it.

Exercises 1

1. Study the program below and answer the questions which follow.

```
2 def singHappyBirthday(person):
3     print("Happy birthday to you.")
4     print("Happy birthday to you.")
5     print("Happy birthday dear", person)
6     print("Happy birthday to you.")
7     return
8
9 singHappyBirthday("Josephine")
```

What is the name of the parameter? _____

What is the name of the argument? _____

What output does the program display?
(Key in the code and run it if you need to). _____

Explain the main benefit of using a parameter
in the function `singHappyBirthday` _____

Explain the use of `person` in line 5. _____

Given the above program, write one line of code that would display the following output:

```
Happy birthday to you.
Happy birthday to you.
Happy birthday dear John
Happy birthday to you.
```

Write a single line of code that calls the function using *Mary* an argument

2. Study the two function definitions shown below and answer the questions that follow:

```
1  def hisHomework(maleName):
2      print(maleName, "loves to do his homework")
3      print("He never misses a day")
4      print("He even loves the men in white")
5      print("Who are taking him away")
6      # End of function
7
8  def herHomework(femaleName):
9      print(femaleName, "loves to do her homework")
10     print("She never misses a day")
11     print("She even loves the men in white")
12     print("Who are taking her away")
13     # End of function
```

What are the names of the two functions?

What are the names of the parameters?

Outline the main difference between the two functions

Explain lines 2 and 9

The listing does not show any arguments. Why not?

The three column below show three different function calls – each using different arguments. What are the arguments? Use the space underneath each call to write the output that would be displayed from that call.

`hisHomework("Jack")`	`herHomework("Jill")`	`hisHomework("Mary")`

What output does the program shown below display?

```
def displayMessage(msg):
    print("The message is:", msg)

displayMessage("I am Sam")
displayMessage("Sam I am")
displayMessage("I do not like Green Eggs and Ham")
```

List any parameters/arguments used.

Now study the following. What difference(s) do you notice? (Try to use words like parameter and argument in your answer). What output does this program display?

```
def displayMessage(str):
    print("The message is:", str)

displayMessage("I am Sam")
displayMessage("Sam I am")
displayMessage("I do not like Green Eggs and Ham")
```

Finally what output does this program display?

```
def displayMessage(str):
    print("The message is:", str)
    print(str)

displayMessage("I am Sam")
displayMessage("Sam I am")
displayMessage("I do not like Green Eggs and Ham")
```

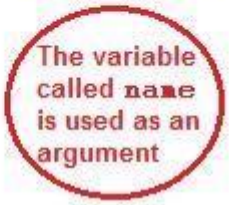
Lesson: Using variables as arguments

Variables can also be passed into a function.

Look at the code below carefully – the variable `name` is the **argument** and `firstName` is the **parameter**.

FIGURE 4.11: Used as an argument

```
1  def homework(firstName):
2      print(firstName, "loves to do his homework")
3      print("He never misses a day")
4      print("He even loves the men in white")
5      print("Who are taking him away")
6      # End of function
7
8  name = "Jack"
9  homework(name)
10
```



When the function is called (line 10), the string *Jack* is passed into the function - `firstName` is assigned the value *Jack*.

The program would display the exact same output if the call was:

```
homework("Jack")
```

The code below uses the `input` command to prompt the user for a name.

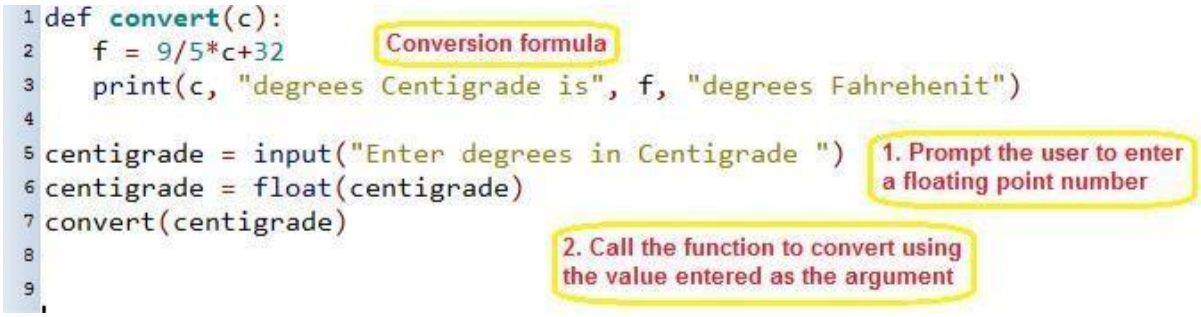
FIGURE 4.12: Using input command

```
1  def homework(firstName):
2      print(firstName, "loves to do his homework")
3      print("He never misses a day")
4      print("He even loves the men in white")
5      print("Who are taking him away")
6      # End of function
7
8  name = input("Please enter a name")
9  homework(name)
```

Look and Learn!

This code converts whatever number the user enters for Centigrade into the Fahrenheit equivalent – `centigrade` is the argument and `c` is the parameter.

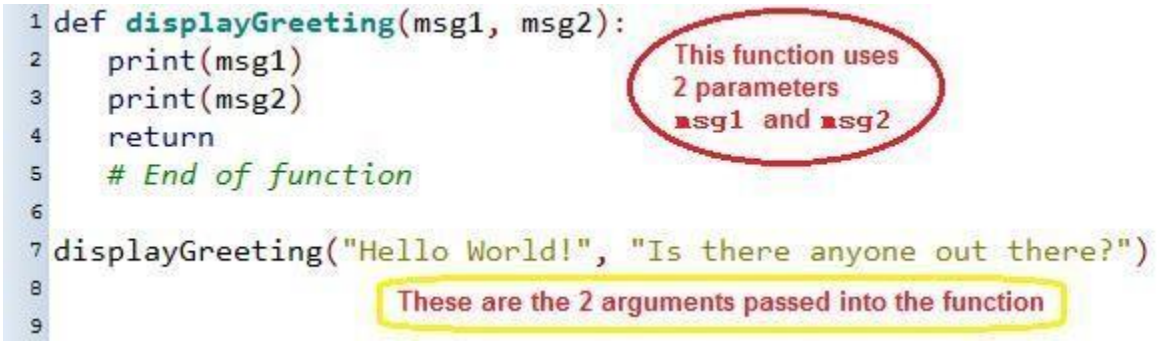
FIGURE 4.13: Conversion formula



Multiple parameters

A function can have more than just one parameter. Take a look at the following:

FIGURE 4.14: Two parameters



When the function `displayGreeting` is called (line 7) Python does two assignments:

the value of the first argument (i.e. *Hello World!*) is assigned to the parameter `msg1`.

the value of the second argument (i.e. *Is there anyone out there?*) is assigned to the parameter `msg2`.

The program causes the following output to be displayed:

```
Hello World
Is there anyone out there?
```

Parameters are received into a function in the same order as the arguments provided. Therefore, if the arguments were switched around like this,

```
displayGreeting("Is there anyone out there?", "Hello World!")
```

the function would cause the text below to be displayed.

```
Is there anyone out there?
Hello World!
```

Notice the use of a comma to separate parameters (and arguments) from one another?

Test your understanding: What output do you think this program would display? What are the arguments and what are the parameters?

```
def displayGreetingV3(msg1, msg2):
    print(msg1)
    print(msg2)
    return
    # End of function


name = "HAL"
displayGreetingV3("Hi", name)
```

Write your answer below:

Class Activity

Study the following code carefully.

```
1  def add(n1, n2):
2      sum = n1 + n2
3      print ("The answer is", sum)
4      return
5      # End of function
6
7
8  add(2, 3)
9
10
```

A function that displays the sum of 2 numbers

Call the function passing in 2 and 3 as the arguments

What do you think the program does?

Is this program flexible (robust)? How might the program be improved?

Now study the code below – in particular lines 7, 8 and 9.

```
1  def add(n1, n2):
2      sum = n1 + n2
3      print ("The answer is", sum)
4      return
5      # End of function
6
7  x = input("Enter first number")
8  x = int(x)
9  add(x, 3)
10
```

The value of x is passed to the function as the first argument

Key the code in and run it. State in *one sentence* what the program does?

Modify the above program so that it adds *any* two numbers entered by the end-user. (*Hint:*
You will need to add code to prompt the user to enter a second number – call it `y`.)

Finally, implement definitions for the following functions.

Function (prototype)	Description
`def subtract(n1, n2):`	A function to display the difference between `n1` and `n2`
`def multiply(x, y):`	A function to display the product of `x` and `y`
`def quotient(n1, n2):`	A function to display the result when `n1` is divided by `n2`

Exercises 2

1. Can you figure out what is wrong with each of the following pieces of code? Type each program in and try running it. Can you fix the problem? Use the space provided to record each problem and your solution.

a)

```
1 def displayMessage():
2     print(str)
3
4 displayMessage("I am Sam")
```

b)

```
1 def displayMessage(str):
2     print(str)
3
4 displayMessage()
```

c)

```
1 def displayMessage(str):
2     print(str)
3
4 displayMessage("I am Sam", "Sam I am")
```

d)

```
1 def displayMessage(str):
2     print(message)
3
4 displayMessage("I am Sam")
```

e)

```
1 def displayMessage(str):
2     print(str)
3
4 displayMessage(I am Sam)
```

f)

```
1 def displayMessage(str1, str2):
2     print(str1)
3     print(str2)
4
5 displayMessage("I am Sam")
```

2. The function below is called `displayGreetingV1`.

This function uses one parameter – `name`.
The value passed into `name` is displayed in a greeting message.

```
1 def displayGreetingV1(name):
2     print("Hello", name, "Pleased to meet you.")
3     return
4     # End of function
```

For example, the call,

```
6 displayGreetingV1("Joe")
```

would result in this message being displayed,

Hello Joe Pleased to meet you.

Now answer the following questions:

Write a line of code to call the function using the name “Fred” as an argument

Modify the function so that it displays the text *Pleased to meet you*. on a separate line.

Write a line of code to initialise a variable `n` (for name) to the string *Joe*. Now call the function using `n` as the argument.

Write code that prompts a user to enter a name and then use the name entered as the argument to the above function.

What does the following function do? (Be careful!)

```
1 def add(n1, n2):
2     answer = n1 * n2
3     print(answer)
4     # End of function
```

Study the following function definition and state what you would expect to happen when each of the calls displayed on the left hand side below is run

```
1 def add(n1, n2):
2     answer = n1 + n2
3     print(answer)
4     # End of function
```

Function Call	Expected output
`add(2,4)`	
`add(1+1,4)`	
`a = 3` `add(a,4)`	
`b = 5` `add(2,b)`	
`a = 3 b = 5` `add(a,b)`	
`a = 3 b = 5` `add(b,a)`	
`a = 3 b = 5` `add(a+b,a)`	
`a = 3 b = 5` `add(a,a+b)`	

5. Write your own function to display the following message – name the function `iAmSam`

```
I am Sam
Sam I am
```

Modify the function you just wrote to use a parameter (call it `name`). Whatever name is passed in as the argument to the function should be used instead of *Sam* in the output message.

Define your own function to accept three parameters – your first name (`name`), the county you come from (`county`) and your favourite colour (`colour`). Test your function by calling it in a program. The function should print out a message like:

Hi *Joe* from *Cavan*. Your favourite colour is *red*.

8. Study the following two function definitions and answer the question that follows.

```
2 def displayGreetingV2(msg):
3     print(msg)
4     return
5     # End of function
```

```
2 def displayGreetingV3(msg1, msg2):
3     print(msg1)
4     print(msg2)
5     return
6     # End of function
7
```

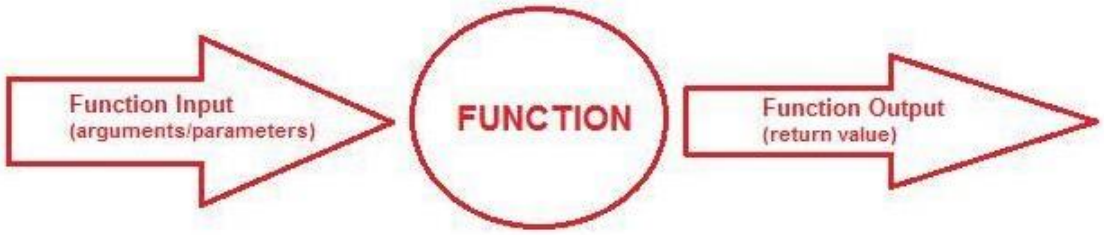
State what you would expect to happen when each of the code blocks displayed on the left hand side below is run.

Code Block	Expected output
`displayGreetingV2("Good evening, Dave")`	
`str = "Good evening, Dave"` `  displayGreetingV2(str)`	
`name = "Dave"` `  displayGreetingV2("Good evening", name)`	
`displayGreetingV3("Good evening", "Dave")`	
`name = "Dave"` `  displayGreetingV3("Good evening", name)`	
`str = "Good evening, Dave"` `  displayGreetingV3(str)`	
`sum = 2+3` `  displayGreetingV2(sum)`	
`sum = "2+3"` `  displayGreetingV2(sum)`	
`sum = "2+3"` `  displayGreetingV3(sum, 2+3)`	
`displayGreetingV3("2+3", "equals", "5")`	

Lesson: Function Return Values

Functions can be thought of little machines that accept **inputs** and produce an **output**. The inputs are *parameters* and the output is the called the *return value*.

FIGURE 4.15: Return Value



Have you ever changed money from one currency to another?

Consider the conversion of pounds sterling into euro. The bank would need to know the amount in sterling and the exchange rate. These are the inputs. The inputs are used to calculate an output. In this case the output would be the amount in euros.

FIGURE 4.16: Convert Sterling to Euro

The bank might use the following Python program to do the conversion.

We already know that data can be passed into functions using parameters. The function `convert` receives 2 parameters – `sterling` and `rate`. The arguments passed into the function are `pounds` and 1.26 respectively. (The program is based on a conversion rate of £1 = €1.26).

FIGURE 4.17: Value returned by convert

```
1  def convert(sterling, rate):
2      euro = sterling * rate
3      return (euro)
4
5
6  pounds = input("Enter sterling amount")
7  pounds = float(pounds)
8
9  amountInEuro = convert(pounds, 1.26)
10
11 print("The converted amount is", amountInEuro, "euros")
```

The value returned by convert is assigned to the variable amountInEuro

IMPORTANT: The `return` statement on line 3 is used to pass the value of `euro` back to the caller.

When the function ends the value in `euro` is assigned to the variable `amountInEuro`.

KEY POINTS:

Data can be passed in and out of functions using parameters and the return statement. Parameters are used by functions to receive input (from the caller) The return statement is used by functions to send output (back to the caller) The syntax of the return statement is `return <expression>`

Look and learn!

The function below converts miles to kilometres (based on 1 = 1.6)

The input to the function is parameter `miles`. The output of the function is `kms`.

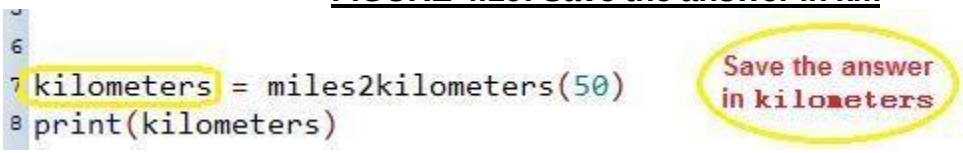
FIGURE 4.18: Pass the answer back to the caller

```
1  def miles2kilometers(miles):
2      kms = miles * 1.6
3      return kms
4      # End of function
5
```

Pass the answer back to the caller

We now look at the code the call function in order to calculate the number of kilometres in 50 miles.

FIGURE 4.19: Save the answer in km



The output of the function is `kms`. This value is assigned to the variable `kilometres`.

Once a function ends its variables and parameters are all destroyed.

Test your understanding:
The function shown below (`kilometers2miles`) accepts a value in kilometres as input and then outputs the equivalent in miles (based on $1 = 0.62$)

Write a line of code to call this function to convert 80 kilometers into miles and display the result.

```
def kilometers2miles(kms):
    m = kms * 0.62
    return m
    # End of function
```

Now write a line of code to call the first function, `miles2kilometers`. Use `miles` (returned by your call to `kilometers2miles`) as the argument. What is your result?

Look and learn!
The function below (`add`) returns the sum of whatever two numbers are passed into it.

The function uses two parameters – `n1` and `n2`.

The function is called on line 9. The arguments passed into `add` are 2 and 3. The value returned by the function (i.e. `sum`) is assigned to the variable `answer`.

FIGURE 4.20: Simple function adding 2 numbers

```
2  # A simple function to add two numbers
3  def add(n1, n2):
4      sum = n1+n2
5      return sum
6      # End of function
7
8  # Call the function
9  answer = add(2,3)
10 print(answer)
```

The assignment on line 9 means the calling code has a handle on the result of the function and can use after the function is executed.

In this case line 10 displays the result of the addition to the user.

The three code snippets shown below all do the same thing. You should study them carefully before trying them out for yourself.

FIGURE 4.21: Three code snippets

```
a=2
answer = add(a,3)
print(answer)
```

```
a=2
b=3
answer = add(a,b)
print(answer)
```

```
a=2
answer = add(a,a+1)
print(answer)
```

Test your understanding:

Take a look at the two function definitions shown below:

FIGURE 4.22: Function 1 and 2…

```
1 def add(n1, n2):
2     sum = n1 + n2
3     return sum
4     # End of function
```

FUNCTION 1

```
1 def add(n1, n2):
2     sum = n1 + n2
3     print("The answer is", sum)
4     return
5     # End of function
```

FUNCTION 2

Which function is better in your opinion? Explain.

Write a line of code to call the first function – save your answer in a variable e.g. `answer`. Now try to do the same for the second function. What problem do you encounter?

Exercises 3

Imagine you were asked to write a separate function to carry out each of the tasks listed below.

For each of the tasks suggest a name for the function and state what you think the inputs and the outputs should be.

Convert from degrees Celsius to Fahrenheit

Suggested name	
Input(s)	
Output	

Convert euros to pounds sterling (based on a certain exchange rate)

Suggested name	
Input(s)	
Output	

Calculate a person's net pay based on their gross pay and their deductions

Suggested name	
Input(s)	
Output	

Calculate the area of a rectangle

Suggested name	
Input(s)	
Output	

Calculate simple interest earned on an amount of money

Suggested name	
Input(s)	
Output	

Count the number of words in a document

Suggested name	
Input(s)	
Output	

Query a person's account balance

Suggested name	
Input(s)	
Output	

Login to a system e.g. Facebook

Suggested name	
Input(s)	
Output	

2. The function below can be used to calculate the area of a rectangle.

```
2 def calculateRectangleArea(length, breadth):
3     area = length*breadth
4     return area
```

Examples of two different ways the function can be called are shown below (A and B)

A.

```
6 rectangleArea = calculateRectangleArea(7, 8)
```

B.

```
6  length = input("Please enter the length of the rectangle")
7  length = int(length)
8  breadth = input("Please enter the breadth of the rectangle")
9  breadth = int(breadth)
10 rectangleArea = calculateRectangleArea(length, breadth)
```

Which code is more robust/flexible in your opinion – A or B? Explain.

_____

_____

_____

3. Explain the syntax error in the code shown below. How could this error be fixed?

```
1 def add(n1, n2):
2     answer = n1 + n2
3     return answer
4     # End of function
5
6 result = add(8, 3)
7 print(answer)
```

_____

_____

_____

_____

4. What is wrong with the following? Suggest a solution.

```
1 def add(n1, n2):
2     answer = n1 + n2
3     return
4     # End of function
5
6 result = add(8, 3)
7 print(result)
```

_____

_____

_____

_____

5. What, if anything, is wrong with the following? Suggest a solution if one is needed.

```
1 def add(n1, n2):
2     answer = n1 + n2
3     return answer
4     # End of function
5
6 add(8, 3)
```

Complete the program below so that it can add *any* two numbers entered by the end-user.

```
1 def add(n1, n2):
2     answer = n1 + n2
3     return answer
4     # End of function
5
6 x = input("Enter first number:")
7 x = int(x)
8 y =
9 y =
10
11 result = add( , )
12 print(result)
```

Implement definitions for the following functions (in the same program). Write code to test your functions.

Function (prototype)	Description
`def subtract(n1, n2):`	A function to return the difference between `n1` and `n2`
`def multiply(x, y):`	A function to return the product of `x` and `y`
`def quotient(n1, n2):`	A function to return the result when `n1` is divided by `n2`

8. The function `square` squares the number passed in. Modify the code so that it cubes.

```
def square(x):
    answer = x * x
    return answer
    # End of function

number = input("Enter a number to square:")
number = int(number)

nrSquared = square(number)
print("The square of", number, "is", nrSquared)
```

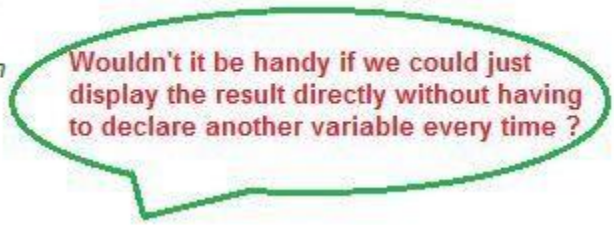
Function Composition

Now that we understand return values, function composition will be easy!

Take a look at the following code to add two numbers and display the result.

FIGURE 4.23: Wouldn't it be handy?

```
def add(n1, n2):
    answer = n1 + n2
    return (answer)
    # End of function

result = add(2,3)
print(result)
```



The problem is that we have to declare the variable `result` to 'catch' the value returned by the function.

Well actually this isn't really necessary. Look at this!

```
print( add(2,3) )
```

FIGURE 4.24: Print the result!

The call to `add` is used as an argument to the `print` function. In fact, the value returned by `add` is passed into print. The effect is to display 5.

Function composition means using one function as an argument to another function.

When a function is used an argument to another function the *inner* function is always evaluated first. (Python starts at the innermost function and works its way out).

The call below displays 10. Can you figure out why?

FIGURE 4.25: Print the enhanced syntax

```
print( add( add(2,3), 5) )
```

Look and Learn!

We know the `input` command returns text data. In order to work with numbers, the data entered must first be converted into numeric format – either integer or floating point.

To convert this data to integer format we use `int` and to convert the data into decimal format we use `float`. Normally this is done in two lines as follows:

FIGURE 4.26: Convert text to integer and float

```
nbr = input("Please enter an integer:")
nbr = int(nbr)
```

Converting text to a integer

```
decimal = input("Please enter a decimal number:")
decimal = float(decimal)
```

Converting text to a float

Function composition can be used to perform these conversions as follows:

FIGURE 4.27: Function composition

```
nbr = int( input("Please enter an integer:") )
decimal = float(input("Please enter a decimal number:"))
```

Lesson: The math library

Finding square roots

The square root of 16 is 4 because $4^2 = 16$. The square root symbol is $\sqrt{\ }$. So, $\sqrt{16} = 4$.

The square root of any number, is the number which must be squared to get .

The Python library `math` contains a host of ready-made functions that can be used to make calculations. One of these functions is called `sqrt`. This function returns the square root of its argument.

Before a program can use any library function, it must first import the library using the `import` statement (see line 1 below)

The listings below show three different ways to find and display the square root of 2. Try them out.

FIGURE 4.28: Import Math module

```
1 import math
2
3 # Print the root of 2
4 answer = math.sqrt(2)
5 print(answer)
```

The numeric literal 2 is used as an argument to the sqrt function

```
1 import math
2
3 # Print the root of 2
4 n=2
5 answer = math.sqrt(n)
6 print(answer)
```

The variable n is used as an argument to the sqrt function

```
1 import math
2
3 # Print the root of 2
4 print( math.sqrt(2) )
```

The sqrt function is used as an argument to print

The syntax for making a call to any Python library function is as follows:

```
<library-name>.<function-call>
```

For example, to use the `sqrt` function from the Python math library to calculate the square root of 2 we would write:

```
math.sqrt(2)
```

Test your knowledge – square root challenge!

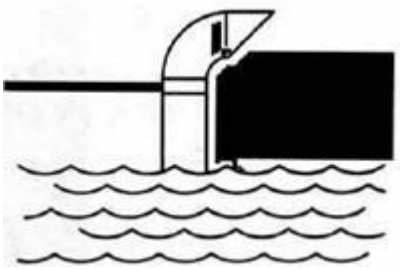
One particularly interesting application of the square root function is its use to estimate how far a person can see when looking through a periscope in a submarine.

The actual distance is given by the formula:

$$= 1.5\sqrt{h}$$

where, is the distance in miles and h is the height in feet of the periscope above the surface of the water.

FIGURE 4.29: Square Root!



Let us say that a submarine captain is looking through a periscope that is 3 feet above the water. Write a program to calculate how far the captain is able to see. Include a function definition to perform the calculation in your solution.

Look and Learn!
Try running this code.

FIGURE 4.30: The pow function

```
1 import math
2
3 answer = math.pow(2, 3)
4 print ( answer )
5
```

The pow function returns a number raised to a power

The program uses the `pow` function from the `math` library to calculate 2 to the power of 3.
The result is assigned to the variable `answer` and is then displayed.

The need for `answer` can be avoided if we use function composition.

FIGURE 4.31: Passing pow function as an argument

```
1 import math
2
3 print( math.pow(2, 3) )
4
```

Function composition is when one function is passed as an argument to another function

The program does exactly the same thing. In this case, the `pow` function is passed as an argument to the print function.

We can define our own power function using the power operator.

FIGURE 4.32: Power operator

```
1 def power(a, b):
2     return a ** b
3
4 print( power(2, 3) )
```

The Python operator for power is **

See if you can figure these out for yourself.

1. **FIGURE 4.33: Random Number**

```
1 import random
2
3 randomNumber = random.randint(1, 6)
4 print(randomNumber)
```

Generate a random number between 1 and ?

2. **FIGURE 4.34: Import the os**

```
1 import os
2
3 cwd = os.getcwd()
4 print("Current working directory is", cwd)
5
6 print("The name of the current program is")
7 print(__file__)
```

Lesson: Encapsulating code in functions

The process of taking code and putting it into functions is called **encapsulation**.

Programmers ask the following questions when encapsulating code using functions:
What does the code do? This would be a good name for the function.

What are the inputs? The function should have a parameter for each input. What is the output? This will be the return value of the function

For example, we can encapsulate the code below using a function.

```
# This program converts from kilometres to miles
kms = input("Enter a value in kilometres ")
kms = float(kms)
mls = kms*0.62
print(kms, "kilometres is approximately", mls, "miles")
```

We ask the three questions:

What does the code do? The code converts kilometres to miles. It uses a conversion of one kilometre to approximately 0.62 miles. A good name for the function would be `convertKms2Miles`

What are the inputs? The only input is the value in kilometres to convert. What is the output? The output is the converted value in miles.

We can now write our function.

```
   # This program converts from kilometres to miles
   def convertKms2Miles(k):
   m = k*0.62
   return m

# Program execution starts from here
   kms = input("Enter a value in kilometres ")
   kms = float(kms)
   mls = convertKms2Miles(kms)
   print(kms, "kilometres is approximately", mls, "miles")
```

kEY POINT: Deciding what function parameters and return values to use is an essential programmer skill.

Lesson: Design using parameters and return values

The code below displays the contents of Met Eireann's weather forecast web page in HTML format. Don't worry about the technical details for the moment.

FIGURE 4.35: Import the urllib.request

```
1 import urllib.request
2
3 page = urllib.request.urlopen("http://www.met.ie/forecasts")
4 text = page.read()
5 print (text)
```

Let's put the code into a function and call it.

FIGURE 4.36: Putting code into a function

```
1 import urllib.request
2
3 def getWeather():
4     page = urllib.request.urlopen("http://www.met.ie/forecasts")
5     text = page.read()
6     print (text)
7
8 getWeather()
```

Ok. So this works. The code is more modular because it has a function but it still just displays the same old weather page every time it's called. Boring!

FIGURE 4.37: Conversation

The solution below passes in the web page URL as an argument and the function returns the page to the caller. This is the bells and whistle solution!

FIGURE 4.38: Web page URL as an argument

```
import urllib.request

def getWebPage( url ):
    page = urllib.request.urlopen(url)
    text = page.read()
    return (text)

weatherPage = getWebPage( "http://www.met.ie/forecasts" )
# do something with weatherPage

sportsPage = getWebPage( "http://www.irishtimes.com/sport" )
# do something with sportsPage
```

Chapter Exercises

Encapsulate the following pieces of code using functions.

a)

```
print("Choose one of the following.....")
print("(1) Balance Enquiry")
print("(2) Cash Lodgement")
print("(3) Cash Withdrawal")
print("")
print("(0) Exit")
print("")

choice=input("Enter your choice : ")
choice=int(choice)
```

Function

What does the code do? :

What are the input(s) if any

What is the output?

b)

```
c = input("Enter degrees in Centigrade ")
c = float(c)
f = 9/5*c+32
f=round(f, 2)
print(c, "degrees Centigrade is", f, "degrees Fahrenheit")
```

Function

What does the code do? :

What are the input(s) if any

What is the output?

c)

```
import math

r = input("Enter a number:")
r = float(r)
a = math.pi*(r**2)
a = round(a,2)
print("The answer is", a)
```

Function

What does the code do? :

What are the input(s) if any

What is the output?

Write a program that prompts the user to enter the dimension required to calculate the area of a square and then use this value to calculate its area. The result should be displayed in a meaningful message. Use a function to calculate the area.

Write a function to return the sum of three integers.

The program shown here calculates the compound interest earned on €1000 invested for 10 years at a rate of 5%.

```
# Compound Interest Calculator

# Perform the calculation
amount = 1000*(1+0.05)**10

# Display the answer
print("The compounded amount is", amount)

```

The formula used is:

$= (1 +$ $)$

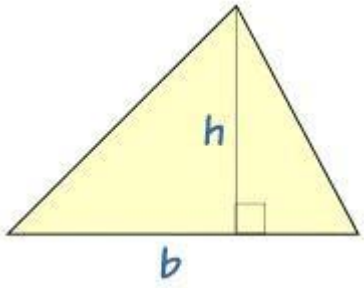
where, is the initial principal, the interest rate and the time in years.

Modify the program so that it asks the user for the required inputs and does the calculation in a function.

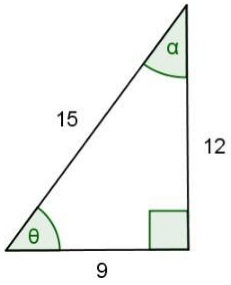
Write a program that prompts a user to enter the radius of a circle and uses the value entered to calculate and display its area. The calculation should be performed in a function. (Note: $= {}^{2}$)

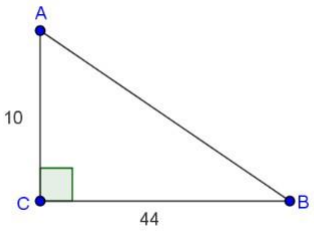
Define a function to calculate the area of a triangle using the formula:

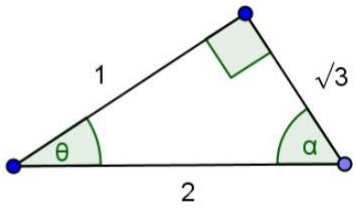
$$= {}_{2} \times \frac{1}{} \times h$$



Use this function to calculate the areas of the triangles displayed below:



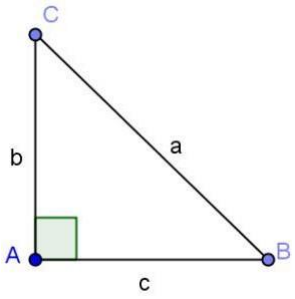




The Theorem of Pythagoras states that in a right angled triangle the square of the hypotenuse is equal to the sum of the squares of the other two sides.

In the triangle below, is the length hypotenuse while and are the lengths of the other two sides. Mathematically, Pythagoras's Theorem can be written as

$$^2=^2+^2$$

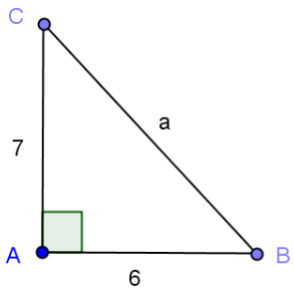


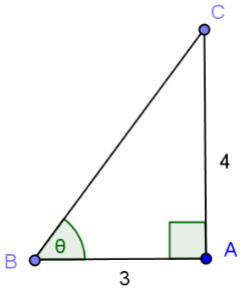
Pythagoras's Theorem is very useful when we know the lengths of two sides of a right angled triangle and we need to find the length of the third side. The following equation which derives from the above formula can be used to calculate the length of the hypotenuse of a right angled triangle.

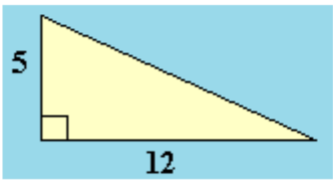
$$=\sqrt{^2+^2}$$

Define a function called `hypotenuse` that calculates and returns the length of the hypotenuse of a right angled triangle when the lengths of the other two sides are given.

Then use the function to find the lengths of the hypotenuses of the triangles displayed below:



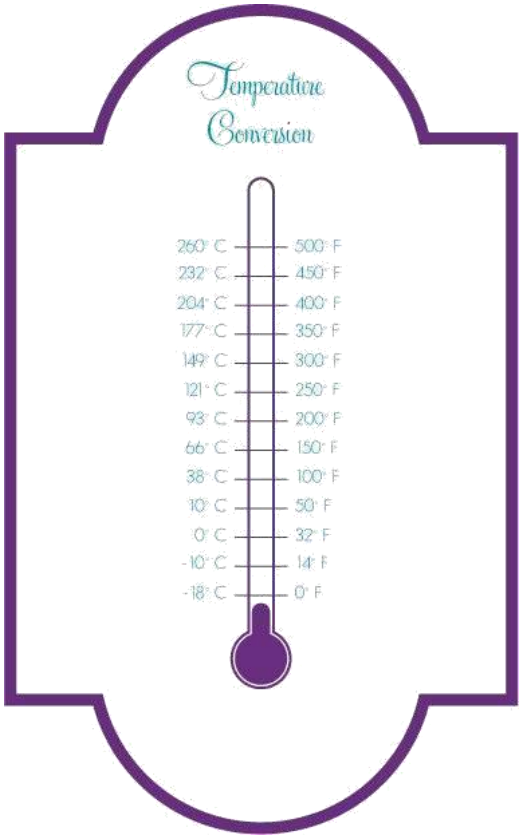




Write a program that prompts a user to enter a number of days and then proceeds to compute the number of minutes in that number of days.

Extend the program just written to prompt for a number of hours as well as a number of days.

The diagram on the right hand side illustrates the Fahrenheit values for a selection of Celsius values.



Given the formulae below to convert between the two scales write a program that can be used to verify the accuracy of the values shown.

$$=_5 \frac{9}{+32}$$

$$=(-32)\times\frac{5}{9}$$

The program will contain two functions as follows:

```
def F2C(f):
```

```
def C2F(c):
```

11. Write a program to carry out the following currency conversions:

From	To
Euro	Pounds Sterling
Pounds Sterling	US Dollar

The program should define two functions – one for each conversion. Each function should receive the amount to be converted as a parameter and return the converted amount. You will need to look up the proper exchange rates yourself (try xe.com).

When you are satisfied that your currency converter works properly enhance it by writing a small function that converts from Euro to US Dollar. For this task you should rely *only* on the functions you've just implemented.

Verify that your program works.

Chapter 5

Introduction to Conditions & Decisions (Building Logic into Python Programs)

LEARNING OBJECTIVES

1. Objective 1.
2. Objective 2.
3. Objective 3.

LECTURE DISCUSSION

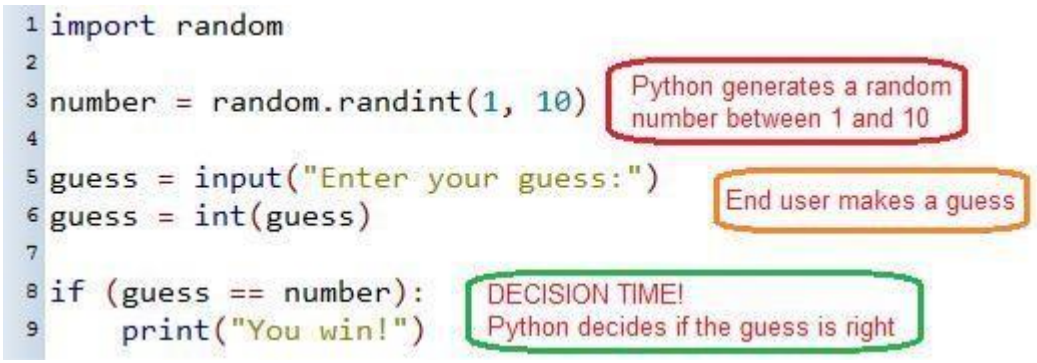
DECISIONS, DECISIONS – PYTHON CONDITIONS

In this section you will learn how to use Python to make decisions.

Let's start off with a simple game program. The computer picks a random number between 1 and 10 and the user tries to guess it. If the guess is right the message *You win!* is displayed.

Here's the code for the game. Study it carefully – there's a fair bit going on.

Figure 5.1: Guess the number



Line 3 is an assignment statement. A random number between 1 and 10 is generated and stored in the variable `number`.

Lines 5 and 6 should be easy to understand. They are both assignment statements. Line 5 prompts the user to enter a number and stores this number in the variable called `guess`. Line 6 converts the value of `guess` to an integer (from a string).

Line 8 checks whether the user's guess is the same is the computer's number. This check is made using an **if-statement**. If statements are Python's way of making decisions.

An if-statement is composed of the reserved word `if`, followed by a **condition**, followed by a colon. All conditions evaluate to either `True` or `False`.

In this case the condition is **`guess == number`**. *Line 9 of the program will be executed only if the two numbers are the same*. Notice the use of double equals to compare two variables.

The execution of line 9 is conditional i.e. it depends on whether the condition associated with it (**`guess == number`**) evaluates to `True` or `False`. Notice how this line is indented.

If the two numbers are not the same the condition will evaluate to `False` and Python will not run line 9.

Now type the above program in, and try it out.

Quick Question
Why might it be useful to insert line 4 as follows into the above program? (try it and see!)

```
4 print(number)
```

So there we have it. Computer programs make decisions using if statements. The syntax and semantics (meaning) of the Python single if-statement is shown below.

```
if <condition>:
   <conditional-code>
```

If the condition evaluates to True then conditional code inside the if statement will be executed.

If the condition evaluates to False then the conditional code is skipped and execution continues from the next line of code after the if-statement.

Note the use of the colon at the end of the line and also the fact that the conditional code must be indented.

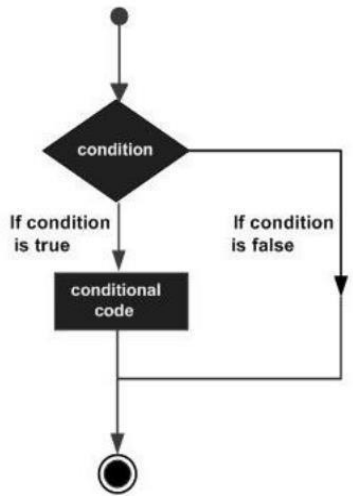
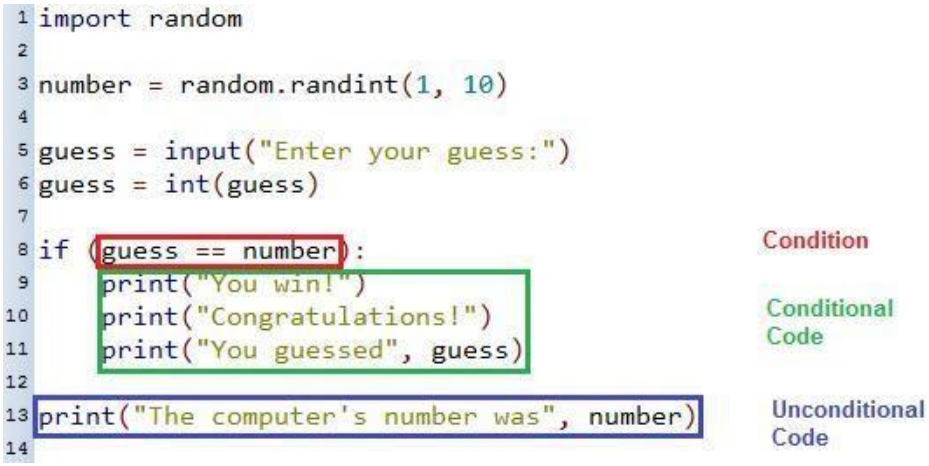


Figure 5.2: ***Flow chart illustration of if-statement***

Let's look at a slightly modified version of our game program:

Figure 5.3: Modified code of the game



Try it out! The condition is still **guess == number**.

This time however, three messages are displayed if the guess is correct. Notice that lines 9-11 are all indented to the same level. These lines are associated with the condition on line 8. They make up what is called the *conditional code* for this condition.

Indentation returns to normal on line 13. Line 13 is therefore executed *unconditionally*. This means that the final message displaying the computer's number will be displayed regardless of what the user entered

Quick Question
How might a random number generator be used to simulate a coin toss?

Exercises 1

1. Identify the *condition* in each of the following code blocks. On what lines are the conditional-code? For each program state the output generated and what exactly the program does?

Code	Answers
(i) `1 result = 100` `2 if (result == 100):` `3  print("Full marks!")`	The condition is: result == 100 The conditional code is on line(s): 3 The program output is: *Full marks!*
State what the above program does	The program tests the variable result and, if it is equal to 100, it displays the message *Full marks!*
(ii) `1 mark = 100` `2 if (mark == 100):` `3  print("Full marks!")` `4  print("Well done!")` `5  print("Perfect score!")`	The condition is: The conditional code is on line(s): The program output is:
State what the above program does	
(iii) `1 result = 50` `2 if (result == 100):` `3  print("Full marks!")`	The condition is: The conditional code is on line(s): The program output is:
State what the above program does	

(iv)

```
1 temperature = 100
2 if (temperature == 100):
3     print("WARNING")
4     print("Boiling point!")
```

The Condition is: _____

The conditional code is on line(s): _____

The program output is: _____

State what the above program does _____

_____

2. Insert the 2 lines of code in the space provided necessary to prompt the user to enter a student result and then convert the value entered to an integer.

```
1
2
3
4
5  if (result == 100):
6      print("Full marks!")
7      print("Well done!")
8      print("Perfect score!")
9
10 print("The result was", result)
11
```

Under what condition are lines 6-8 executed? _____

Under what circumstances is line 10 executed? _____

3. What is wrong with the following? Suggest *two* solutions to the problem.

```
1 mark = input("Enter student result")
2 mark = int(mark)
3
4 if (result == 100):
5     print("Full marks!")
6     print("Well done!")
7     print("Perfect score!")
8
9 print("The result was", result)
```

_____

_____

_____

_____

_____

_____

Explain why the conditional code (lines 3 and 4) in the listing below never gets executed.

```
1 livesLeft = 3
2 if (livesLeft == 0):
3    print("Game over!")
4    print("You have no lives left")
```

Study the code below and explain what happens when the user enters:

```
1 number = input("Enter your guess:")
2 number = int(number)
3
4 if (number == 0):
5     print("You entered zero")
```

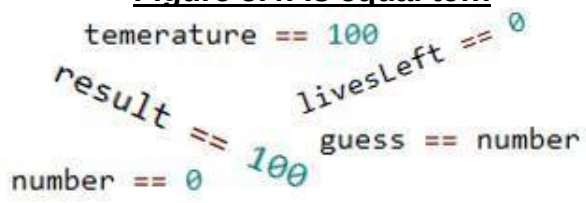
0

Any number other than zero

3. Complete the sentence: When Python evaluates a condition the result is either _____or _____.

Relational Operators
Let's look at some of the conditions we have encountered so far.

Figure 5.4: Is equal to…



Question: What do these conditions have in common?
Answer: They all compare two values and the answer is always either True or False depending on these values at the time of execution.

The values in the above examples are all compared for equality. Conditions such as these are tests for equality because the == operator is used.

== is just one of Python's six relational operators as outlined below.

Table 5.1: Relational Operators

Operator	Meaning
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
==	Equals
! =	Not equals

Relational operators are used to compare two values. They are the basic building blocks for conditions – also known as **Boolean expressions**. The first value is compared to the second value. The type of comparison depends on the operator used. The result will always be either True or False.

EXAMPLES

This example illustrates the use of relational operators to compare two values. Python's result is shown in the right hand column.

Table 5.2: Example 1

Condition	Result
2 > 5	False
5 > 2	True
6 >= 6	True
6 >= 5	True
3 < 1	False
0 > 1	False
1 < 0	False
1 == 0	False
4 == 4	True
4 <= 4	True
3 ! = 4	True
3 <= 4	True

Conditions nearly always involve the use of at least one variable. For the purpose of this example let us assume x = 1, y = 0 and z = −1.

Table 5.3: Example 2

Condition	Result
$5 > x$	True
$x > y$	True
$x <= y$	False
$y <= 0$	True
$z > y$	False
$x == z$	False
$0 == y$	True
$x ! = y$	True

Test your understanding

1. What do these expressions evaluate to? (*Hint*: True or False)
   a) 7 == 7 _____
   b) 7 ! = 7 _____
   c) 7 >= 6 _____
   d) 2 < 3 _____
   e) 3 < 2 _____
2. Assuming x = 1, and y = 0 what would Python e valuate the conditions listed in here to?
   a) x > 5 _____
   b) 5 > x _____

c) y <= 0 _____

d) 0 == y _____

e) x == y _____

f) x ! = y _____

g) x > y _____

h) y > x _____

i) x <= y _____

j) x >= y _____

3. Arithmetic expressions can also be used as part of a condition. Some of the expressions listed below are valid conditions. What do they evaluate to?

a) 2 + 3 == 5 _____

b) 5 = 2 + 3 _____

c) 6 ! = 1 + 2 _____

d) 4 <= 8 + 9 _____

e) 6 + 3 > 6 * 3 _____

f) 18 − 10 = 2 ** 3 _____

Class Activity

See if you can figure out what the short program below does?

```
1 result = input("Enter the student mark")
2 result = int(result)
3 if (result == 100):
4    print("Full marks!")
5    print("Well done!")
6    print("Perfect score!")
7
8 result = input("Enter the next student mark")
9 result = int(result)
10 if (result >= 50):
11    print("Pass")
12
13 print()
```

Remember, conditional code is always indented.

The conditional block from lines 4-6 will be executed only of the condition result==100 evaluates to True. If Python finds this condition is False the flow of control jumps straight to line 8.

Try running the above program using the inputs shown in the table below. Record the output displayed, along with an explanation in the space provided below

1st result entered	2nd result entered	Output Generated	Explanation
100	100		
100	49		
80	50		
50	20		

KEY POINT
The association of a block of conditional code with an if statement ends as soon as the level of indentation returned back to the original level.

Syntax check

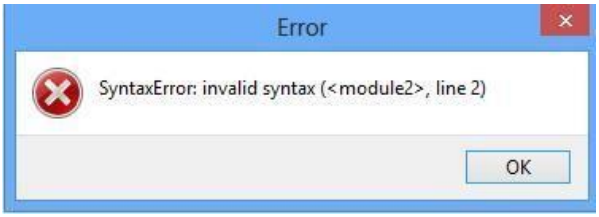
Question: What is wrong with the following code?

Figure 5.4: What is wrong in this code?

```
1 age=21
2 if (age > 18)
3     print("Adult")
```

Answer: The colon is missing from the end of line 2. Attempting to run the above code will result in the following syntax error message box being displayed.

Figure 5.5: SyntaxError1 notice!



One of the skills you need to focus on as a novice programmer is the ability to deal with the unexpected. In this case, you should teach yourself to understand Python error messages such as the one displayed and become competent in dealing with them. To correct the syntax error above just add a colon to the end of line 2.

KEY POINT: Python syntax requires that a colon is used to mark the end of a condition in an `if` statement.

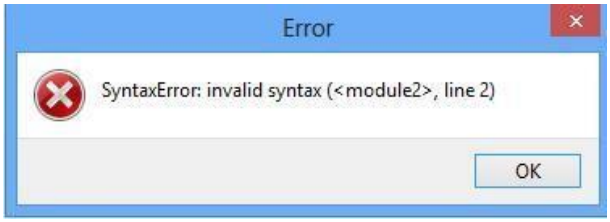
Question: What is wrong with the following code?

Figure 5.6: What is wrong in this code? (Part 2)

```
1 x = 0
2 if (x = 0):
3     print("Zero")
```

Answer: On line 2 an assignment statement i.e. `(x=0)` is being used where a Boolean expression is expected. The code produces the following error – notice it is the exact same error produced when we forgot the colon above.

Figure 5.7: SyntaxError2 Notice!



To correct the syntax error above just change line 2 to be: **if (x == 0):**

KEY POINT: It is a common programming error to use the assignment operator (=) when the intention is to test for equality using the relational equals operator (==)

Exercises 2 (single option if)

1. What output would be generated by the following snippets of code?

	Python Code Snippet	Output
(i)	`temperature = 12` `if (temperature > 20):` `    print("Lovely day")`	
(ii)	`livesLeft = 0` `if (livesLeft > 0):` `    print("Continue game ...")`	
(iii)	`message = "Game Over!"` `livesLeft = 3` `if (livesLeft != 0):` `    message = "Continue game ..."` `print(message)`	
(iv)	`sunshine = True` `if (sunshine):` `    print("Lovely day")` `    print("Wear sunglasses")`	
(v)	`sunshine = False` `if (sunshine):` `    print("Lovely day")` `print("Wear sunglasses")`	
(vi)	`payRate = 10` `if (payRate < 8.65):` `    print("Below minimum wage")` `    payRate = 8.65` `print("Hourly pay is", payRate)`	
(vii)	`payRate = 5` `if (payRate < 8.65):` `    print("Below minimum wage")` `    payRate = 8.65` `print("Hourly pay is", payRate)`	

2. Study the code below carefully and answer the questions which follow:

```
 3
 4 print("If Test")
 5
 6 if (x == y):
 7     print("x is equal to y")
 8
 9 if (x > y):
10     print("x is greater than y")
11
12 if (x < y):
13     print("x is less than y")
14
15 if (y > x):
16     print("y is greater than x")
17
18 if (y < x):
19     print("y is less than x")
20
21 if (x >= y):
22     print("x is greater than or equal to y")
23
24 if (y >= x):
25     print("y is greater than or equal to x")
26
27 print("Last Line")
```

The following table shows three different scenarios for initialising and before the code above (i.e. at lines 1 and 2). What output would be generated for each scenario?

	Initialisation	Output
(i)	`1 x = 3` `2 y = 2`	
(ii)	`1 x = 2` `2 y = 2`	
	`1 x = 2` `2 y = 3`	

Write a program to read in two numbers (e.g. x and y) and display the message *x is not equal to y* as appropriate.

Write a program that prompts the user to enter a student percentage mark (as an integer). If the value entered is **not 50 or greater** display the message *Unfortunately, you have not been successful on this occasion. Try again.*

Double Option if statement (if-else)
Recall, in our number guessing game program, the message *You win!* is displayed if the user guesses correctly.

Let's say we want to display the message *You loose!* if the user's guess is wrong.

Only one of the messages is displayed – **never both**. We use an `if-else` statement as shown below.

Figure 5.8: If-Else statement

```
1  import random
2
3  number = random.randint(1, 10)
4
5  guess = input("Enter your guess:")
6  guess = int(guess)
7
8  if (guess == number):
9      print("You win!")      Executed only if guess == number is True
10 else:
11     print("You loose!")    Executed only if guess == number is False
12
13 print("The computer's number was", number)
```

Python evaluates the condition, **`guess == number`**, and, depending on the outcome either line 9 or line 11 will be executed next. Never both. Line 13 is not part of the if-else statement. It is always executed.

Notice:

4. the `else` keyword appears on the same level of indentation as the `if`
5. the use of the colon after `else`
6. the code attached to the `else` is indented

The else statement is always used as an alternative to the if-statement. Both scenarios are said to be *mutually exclusive*.

Test your understanding.
What output would the following code display?

```
1 result = 72
2 if (result >= 50):
3     print("Well done!")
4 else:
5     print("Unsuccessful")
```

Output: _____

```
1 result = 45
2 if (result >= 50):
3     print("Well done!")
4 else:
5     print("Unsuccessful")
```

Output: _____

What do you think this program would display if the value of `result` was 50?

The syntax and semantics of the Python double option if-else statement is now explained

```
if <condition>:
<IF-CODE>
 else:
   <ELSE-CODE>
```

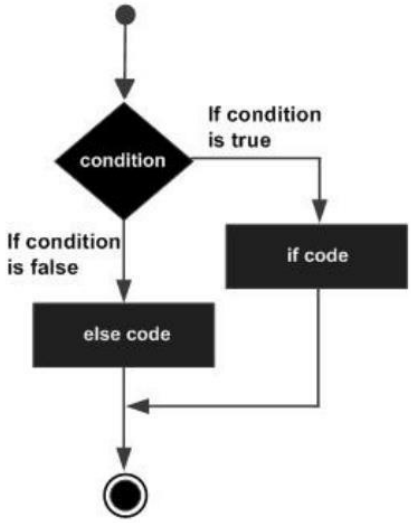


Figure 5.9: Flow chart illustration of if-else statement

Either the IF CODE or the ELSE CODE is executed but not both.

If the condition evaluates to `True` the block of code associated with the if-statement (i.e. the if-code block) is executed.

Otherwise, the block of code associated with the else statement (i.e. the else-code block) is executed. (In other words, the else block is executed only when the condition is evaluated to `False`.)

Once either block has been executed the flow of control continues at the next line (i.e. the line after the else-code).

Line 7 will always be executed in both scenarios shown below.

```
1 result = 72
2 if (result >= 50):
3    print("Well done!")
4 else:
5    print("Unsuccessful")
6
7 print("Finished")
```

The output of this program is:

Well done!
Finished

Line 7 is executed after line 3

```
1 result = 45
2 if (result >= 50):
3    print("Well done!")
4 else:
5    print("Unsuccessful")
6
7 print("Finished")
```

The output of this case is:

Unsuccessful
Finished

Line 7 is executed after line 5

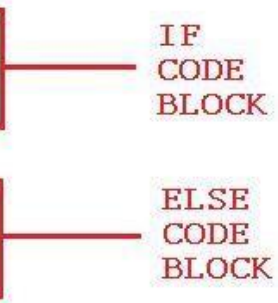
Test your knowledge
Write a program to prompt the user to enter a number and display the word POSITIVE if the number is greater than zero and NEGATIVE if the number is less than zero.

Ignore the possibility of zero

Class Activity
Both if and else code blocks can contain more than one line of code.

Take a look at the following.

```
result = input("Enter student mark")
result = int(result)

if (result >= 50):
    print("Well done!")
    print("Your options are now ....")
    print("1. Graduate and apply for a job")
    print("2. Emigrate")
else:
    print("Unsuccessful")
    print("Options are ...")
    print("1. Repeat course or go on the dole")
    print("2. Emigrate")

```



The behaviour of this program at runtime depends on the value entered by the user for result. Now complete the following:

result	**38**	**51**
Program output:	*Unsuccessful* *Options are …* *1. Repeat course or go on the dole* *Emigrate*	_____ _____ _____ _____

Can you spot any difference between this listing and the one above (at the top of the page)?

```
result = input("Enter student mark")
result = int(result)

if (result >= 50):
    print("Well done!")
    print("Your options are now ....")
    print("1. Graduate and apply for a job")
else:
    print("Unsuccessful")
    print("Options are ...")
    print("1. Repeat course or go on the dole")

print("2. Emigrate")
```

Explain why both listings are *logically equivalent* (i.e. do exactly the same thing).

_____

In your opinion which solution is better? Why?

_____

_____

Exercises 3 (if-else statement)

Pass or Fail. A student pass mark is 40%. Anything less is considered a Fail. Write separate conditions that satisfy these statements in both code blocks shown.

```
1 mark = input("Enter mark")
2 mark = int(mark)
3 if (          ):
4     print("Fail")
5 else:
6     print("Pass")
```

```
1 mark = input("Enter mark")
2 mark = int(mark)
3 if (          ):
4     print("Pass")
5 else:
6     print("Fail")
```

The code below displays a simple menu system, prompts the user to make a choice and then, based on that choice, calls one of two functions.

The requirement is that if the user enters a 1 the function convertCent2Fahr will be called. Otherwise, the function convertFahr2Cent will be called. (Assume definitions exist for both functions).

a) Insert the correct condition into the red box so that this requirement is satisfied.

```
print("Menu Options ....")
print("-----------------")
print("1. Convert Centigrade to Fahrenheit")
print("2. Convert Fahrenheit to Centigrade")
choice = input("Enter your choice (1 or 2)")
choice = int(choice)

if (          ):
    convertCent2Fahr()
else:
    convertFahr2Cent()
```

What changes would need to be made to the rest of the code if the condition in the red box above was: choice == 2?

_____

_____

_____

_____

What do you think would happen if the user entered any number other than 1 or 2 as their choice?

Imagine if the code below was executed towards the end of a game. Study the code and answer the questions which follow.

```
if (score >= highScore):
   print("Well done!")
   print("Do you want to play again?")
else:
   print("Hard luck!")
   print("Do you want to play again?")
```

What output would be generated for the following values of score and highScore?

score	highScore	Output
1001000		
1000	100	
1000	1000	

Re-write the code so that the duplicated line, print("Do you want to play again? ") appears only once. Both solutions should be logically equivalent.

```
if (score >= highScore):
   print("Well done!")
   print("Do you want to play again?")
else:
   print("Hard luck!")
   print("Do you want to play again?")
```

Remove the duplicated instruction to display the message GAME OVER from the following:

7

```
if (score >= highScore):
   print("GAME OVER")
   print("Well done!")
else:
   print("GAME OVER")
   print("Hard luck!")

print("Do you want to play again?")
```

8. a) Explain in your own words what the following short program does. (Note `n1` and `n2` are two randomly generated numbers between 1 and 10).

```
1  import random
2
3  n1 = random.randint(1, 10)
4  n2 = random.randint(1, 10)
5
6  print("What is", n1, "+", n2)
7  userAnswer = input("Enter your answer")
8  userAnswer = int(userAnswer)
9
10 if (userAnswer == n1+n2):
11     print("Correct")
12 else:
13     print("Wrong")
14     print("The right answer is", n1+n2)
```

What change would need to be made so that line 14 gets executed regardless of what the user enters?

The arithmetic expression **`n1+n2`** appears twice in the above program – once on line 10 and again on line 14. Are both instances of this same expression *always* executed at runtime? Explain.

Without changing the logic of the program, suggest a change so that the expression `n1+n2` only occurs once in the program. (*Hint:* use a variable).

Let us say the condition on line 10 was changed to **`userAnswer != n1+n2`**. What other changes would need to be made so that the logic of the program remained unaltered.

Write a program that generates two random numbers and ask the user to enter their product. If the user is right the program should display *Correct*. Otherwise, the program should display *Incorrect*.

6. The program shown prompts its user to enter two numbers. It then displays a message stating which one is bigger.

a) Explain how the program figures out which number is bigger.

```
1  n1 = input("Enter the first number")
2  n1 = int(n1)
3
4  n2 = input("Enter the second number")
5  n2 = int(n2)
6
7  if (n1 > n2):
8     print(n1, "is bigger than", n2)
9  else:
10    print(n2, "is bigger than", n1)
```

What do you think would happen if the values entered for n1 and n2 were the same?

Assume that n1 and n2 are guaranteed to be different.

Now complete the print statements in both listings below so that the messages will make sense when they are displayed at runtime.

```
1  n1 = input("Enter the first number")
2  n1 = int(n1)
3
4  n2 = input("Enter the second number")
5  n2 = int(n2)
6
7  if (n2 > n1):
8     print(      , "is bigger than",      )
9  else:
10    print(      , "is bigger than",      )
```

```
1  n1 = input("Enter the first number")
2  n1 = int(n1)
3
4  n2 = input("Enter the second number")
5  n2 = int(n2)
6
7  if (n1 < n2):
8     print(      , "is bigger than",      )
9  else:
10    print(      , "is bigger than",      )
```

Can you think of any other test that could be used to determine that one number is greater than another? (*Hint:* What is the result when you subtract a larger number from a smaller number?).

Implement your solution in Python.

a) Explain the use of the variable bigger in the code below. Why is there no need for an else statement?

```
1 n1 = input("Enter the first number")
2 n1 = int(n1)
3
4 n2 = input("Enter the second number")
5 n2 = int(n2)
6
7 bigger = n2
8 if (n1 > n2):
9    bigger = n1
10
11 print("The bigger number is", bigger)
```

9. Let us say that line 7 in the above listing was changed to be:

```
7 bigger = n1
```

How would this affect the logic of the if-statement?

c) What minor improvement do you notice in the code below?

```
1  n1 = input("Enter the first number")
2  n1 = int(n1)
3
4  n2 = input("Enter the second number")
5  n2 = int(n2)
6
7  smaller = n1
8  bigger = n2
9  if (n1 > n2):
10     smaller = n2
11     bigger = n1
12
13 print(bigger, "is bigger than", smaller)
```

d) Insert the correct condition in the red box so that the logic makes sense.

```
smaller = n1
bigger = n2
if (          ):
    smaller = n1
    bigger = n2

print(bigger, "is bigger than", smaller)
```

Even or Odd. An even number is an integer which is "evenly divisible" by two. This means that if the integer is divided by 2, it yields no remainder.

The Python condition to test for the *evenness* of a number is **`n%2 == 0`**.

Write a program that prompts the user to enter a number and display the message *Even* if the number entered is even and *Odd* otherwise.

(Use the space on the right).

b) Explain how the following program works.

```
1  def isEven(number):
2      if (number%2 == 0):
3          return True
4      else:
5          return False
6
7  # Program starts execution from  here
8  n = input("Enter the first number")
9  n = int(n)
10 if isEven(n):
11     print(n, "is Even")
12 else:
13     print(n, "is Odd")
```

The function `isEven` is an example of a **Boolean function**. Boolean functions are functions that return either `True` or `False`. They usually perform some sort of test. By convention the name of a Boolean function starts with the prefix **`is`** e.g. `isEven` is a Boolean function that tests whether a number is even or not.

Implement the following Boolean functions. Test your code with numbers entered by the user.

`def isOdd(a):`	A function that returns True if a is odd. False otherwise.
`def isEquals(a, b):`	A function to take two parameters and return True if They are both equal; False otherwise.
`def isLessThan(a, b):`	A function to that returns True if a less than b. Falseotherwise
`def isGtREqual2(a, b):`	A function that returns True a is great than or equal to b. False otherwise.

Write a program that accepts a single number from a user and display the word 'fizz', if the number entered is a multiple of 10. Otherwise, 'buzz'. (*Hint*: A number is a multiple of 10 if the remainder after dividing by 10 is zero).

9. Explain what the code below does.

```
1  def mystery(a, b):
2      if (a > b):
3          return True
4      else:
5          return False
6
7  n1 = 28
8  n2 = 47
9  if (mystery(n1, n2)):
10     print("Message 1")
11 else:
12     print("Message 2")
```

What would the output be if n1 and n2 were both initialised to zero (instead of 28 and 47)

What kind of function is mystery?

What might be a more appropriate name for the function `mystery`?

Modify the code so that:
it uses a more appropriate function name
it generates random numbers between 1 and 100 for `n1` and `n2`
it displays more meaningful messages.
(Use the space on the right).

Write a program that uses a random number generator to simulate tossing a coin.

Define a function (call it `abs`) that accepts an integer from the user and returns its absolute value. Write code to test your function.

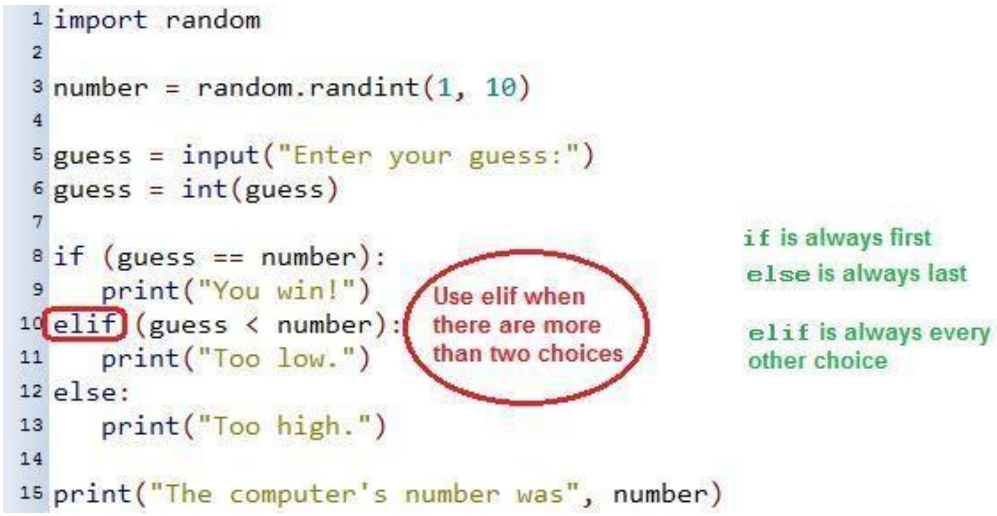
Multiple Option (if-elif statement)
Let's return to our number guessing game for the last time.

This time instead of displaying message *You loose!* if the user's guess is wrong we change our requirement so that the program displays the message
10. *Too High* if the user's guess is greater than the computer's number and
11. *Too Low* if the guess is less.

The program should still display the message *You win!* if the user's guesses is right.

Because there are now *three* possibilities we need to use a multiple option `if-elif` statement in our solution.

Figure 5.10: Three possibilities (If-Else Statement)



Only **one** of the three lines - 9, 11 and 13 - will be executed in any given run of the program. If line 9 is executed the next line to be executed will be 15. If line 11 is executed the next line to be executed will be 15. If line 13 is executed the next line to be executed will be 15.

In other words, once a block of code attached to a `True` condition is executed the flow of control jumps to the next line of code after the `if-elif` statement.

When executing an `if-elif` statement Python executes the block of code attached to the first condition which evaluates to `True`. If none of the conditions are found to be `True` then Python runs the code attached to the `else` statement (if one exists).

Test your understanding.
What output would the program above display for the following values of number and guess?

number	guess	Output
8	4	_____
7	7	_____
1	5	_____

It is worth noting that the same logic can be achieved by re-arranging the order of the conditions in the `if-elif` statement.

For example, these two blocks of code are logically equivalent:

Figure 5.11: Re-arrange if-else statement orders

```
if (guess == number):
   print("You win!")
elif (guess < number):
   print("Too low.")
else:
   print("Too high.")
```

```
if (guess < number):
   print("Too low.")
elif (guess > number):
   print("Too high.")
else:
   print("You win!")
```

Forming your own conditions

Hopefully by this stage you have realised that computer programs make decisions using one of the following types of **if statements**.

single option (the basic if statement)
double option (if-else statement)
multiple-option (if-elif statement)

As a programmer you will need to be able to recognise when and where you build decision statements into your programs. Examples of scenarios that require the use of decisions are:

a social network site (system) will only grant a user access if the password entered was correct for the given username

an ATM system should not permit a user request to withdraw cash unless that user had sufficient funds in their account to meet that request

a payroll program would contain a block of code to be used only if the number of hours worked was over 50 (i.e. overtime calculation)

a flight booking system might offer discounted fares for passengers who are over 55.

a CAO points calculator will only add 100 to a student's points tally if the grade achieved was an A1; 90 for and A2; 85 for a B1; 80 for a B2; 75 for a B3 etc.

As a first step towards creating your own conditions, it is a good idea to formulate the problem using pseudo-code.

Table 5.4: Using pseudo-code

Pseudo-code	Python Boolean Condition
if (password is correct) *grant access*	(passwordEntered == actualPassword)
if (amount requested is less than balance) *process cash withdrawal*	(amount < balance)
if (hours worked exceeds 40) *computer overtime calculation*	(hoursWorked > 40)
if (passenger age is 55 or more) *calculate discount*	(age >= 55)

Test your knowledge

a) Match the strings – *You win!, Too low. Too high* – to the appropriate condition.

```
8  if (guess > number):
9      print(                )
10 elif (guess == number):
11     print(                )
12 else:
13     print(                )
14
```

```
8  if (guess > number):
9      print(                )
10 elif (guess < number):
11     print(                )
12 else:
13     print(                )
14
```

b) Formulate conditions so that the correct messages will be displayed.

```
8  if (                ):
9      print("You win!")
10 elif (                ):
11     print("Too high.")
12 else:
13     print("Too low.")
```

c) Study the code below carefully and complete the table on the right (i.e. Output column).

```
1  import random
2
3  number = random.randint(1, 10)
4
5  guess = input("Enter your guess:")
6  guess = int(guess)
7
8  if (guess < number):
9      print("Too low.")
10 elif (guess > number):
11     print("Too high.")
12
13 print("The number was", number)
```

number	guess	Output
8	4	
2	3	
3	3	
5	1	
7	7	

Write a program to accept a single number from a user and print the word POSITIVE if the number is greater than zero and NEGATIVE if the number is less than zero and ZERO if the number entered is zero.

Syntax and Semantics
The syntax and semantics of the Python multiple option if-elif statement is as follows:

```
if <condition 1>:
   <CODE-BLOCK 1>
elif <condition 2>:
   <CODE-BLOCK 2>
elif <condition 3>:
   <CODE-BLOCK 3>
            …
elif <condition N>:
     <CODE-BLOCK N>
else:
     <ELSE-CODE-BLOCK>
```

Notes:

The first condition always appears in an if statement

There can be as many elif statements as you want Each elif statement must include a condition

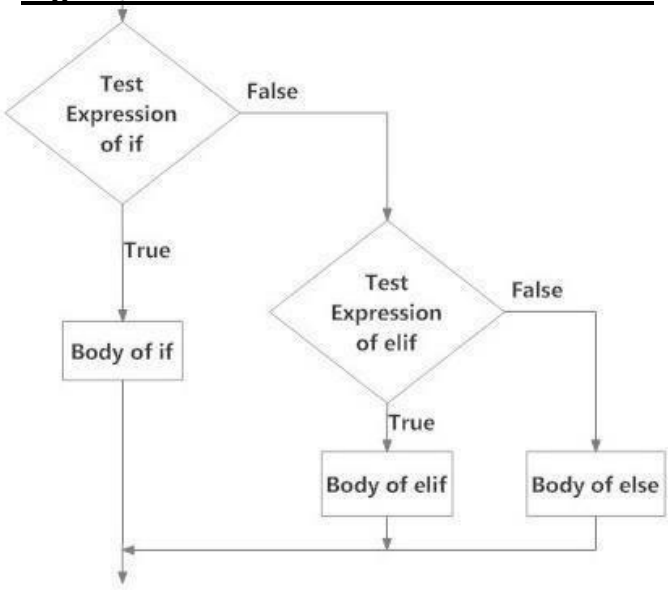
The use of a final else statement is optional

The if, elif and else keywords all appear at the same level of indentation. A colon must be used at the end of the lines containing the if, elif and else statements

Each condition is evaluated in sequence. If a condition evaluates to True then the associated code block is executed. Once the code block is executed the flow of control continues from the next line after the if-elif statement. If none of the conditions evaluate to True then the final code block (i.e. the code block associated with the else keyword) is executed.

The above logic is depicted by the following flowchart.

Figure 5.12: Flowchart of if-else statement



Exercises 4 (if-elif statement)

12. a) What do you think would happen if the user input any letter other than A as their choice?

(Assume that functions add and subtract are defined elsewhere).

```
print("Menu Options ....")
print("-----------------")
print("A - Add")
print("S - Subtract")
choice = input("Enter your choice (A or S)")

if (choice == "A"):
    add()
else:
    subtract()
```

b) Explain how the code below is an improvement on the previous listing.

```
print("Menu Options ....")
print("-----------------")
print("A - Add")
print("S - Subtract")
choice = input("Enter your choice (A or S)")

if (choice == "A"):
    add()
elif (choice == "S"):
    subtract()
else:
   print("ERROR: Choice must be A or S")
```

Let's say we extend the menu as shown below. Fill in the blanks.

```
print("Menu Options ....")
print("-----------------")
print("A - Add")
print("S - Subtract")
print("M - Multiply")
print("D - Divide")
choice = input("Enter choice (A, S, M or D)")

if (choice == "A"):
    add()
elif (choice == "S"):
    subtract()
elif (                ):
    multiply()
                        
    divide()
else:
   print("ERROR: Choice must be A, S, M or D")
```

Given the following definition for the function `add` implement `subtract` function.

Function definition for `add`

```
def add():
    n1 = input("Enter 1st number")
    n1 = int(n1)
    n2 = input("Enter 2nd number")
    n2 = int(n2)
    n3 = n1+n2
    print(n1, "+", n2, "=", n3)
    return
```

Function definition for `subtract`

e) Fill in the blank boxes below to implement a solution without using functions.

```
if (choice == "A"):
    x = int(input("Enter 1st number"))
    y = int(input("Enter 1st number"))
    print(x, "+", y, "=", x+y)
elif (choice == "S"):
    
elif (choice == "M"):
    
elif (choice == "D"):
    
else:
   print("ERROR: Choice must be A, S, M or D")
```

2. (Just for fun!) Key in the following – exactly as it is written.

```
print("""You open your eyes, trying to recall how you got here..\n
      You remember...nothing! \n
      Your vision becomes focused with strange colours. \n
      Suddenly a frightening sound comes from a bizarre creature,\n
      It stands in front of you as large as a mountain!\n
      Has it seen you?!?
      """)

choice = input("Type a to attack, b to back away or c to creep past.")
#The user now fills the variable choice with a, b or c

if choice == ("a"):
    print("You see an odd looking sword on the floor, you grab it and rush the creature swinging wildly ")

elif choice == ("b"):
    print("You back away quietly like the coward you are")
#elif gives you another option in the middle

else:
    print("Carefully, you sneak past and you are almost there when the creature spots you and...")
```

3. Study the two programs shown until you understand what they both do.

PROGRAM 1

```
1  n1 = input("Enter the first number")
2  n1 = int(n1)
3
4  n2 = input("Enter the second number")
5  n2 = int(n2)
6
7  if (n1 > n2):
8     print(n1, "is bigger than", n2)
9  else:
10    print(n2, "is bigger than", n1)
```

PROGRAM 2

```
1  n1 = input("Enter the first number")
2  n1 = int(n1)
3
4  n2 = input("Enter the second number")
5  n2 = int(n2)
6
7  if (n1 > n2):
8     print(n1, "is greater than", n2)
9  elif (n1 == n2):
10    print(n1, "is equal to", n2)
11 elif (n1 < n2):
12    print(n1, "is less than", n2
```

Explain why program 2 is superior to program 1.

Explain why line 11 in program 2 could be replaced by `else:`

Now study the two listings shown below.

```
n1 = input("Enter the first number")
n1 = int(n1)

n2 = input("Enter the second
number")
n2 = int(n2)

if (n1 > n2):
   print(n1, "is greater than", n2)
elif (n1 < n2):
   print(n1, "is less than", n2)
else:
   print(n1, "is equal to", n2)
```

```
n1 = input("Enter the first number")
n1 = int(n1)

n2 = input("Enter the second number")
n2 = int(n2)

if (n2 > n1):
   print(n1, "is less than", n2)
elif (n1 == n2):
   print(n1, "is equal to", n2)
else:
   print(n1, "is greater than", n2)
```

Explain why both these programs are logically the same as program 2 shown above

4. Study the two programs shown until you understand what they both do.

PROGRAM 1

```
x = input("Enter a number from 1-5")
x = int(x)

if (x == 1):
   print("One")
elif (x == 2):
   print("Two")
elif (x == 3):
   print("Three")
elif (x == 4):
   print("Four")
elif (x == 5):
   print("Five")
```

PROGRAM 2

```
x = input("Enter a number from
1-5")
x = int(x)

if (x == 1):
   print("One")
elif (x == 2):
   print("Two")
elif (x == 3):
   print("Three")
elif (x == 4):
   print("Four")
else:
   print("Five")
```

What do you think the purpose of the programs are?

Which program do you think is better and why?

What changes would you need to make to both programs in order to accommodate the following line of code.

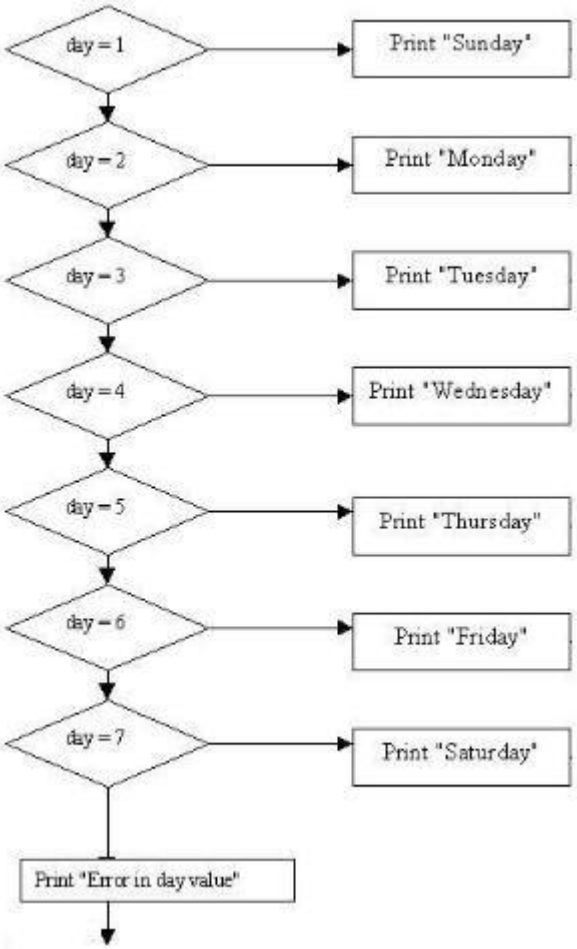
```
print("Invalid number entered - you must enter a number between 1 and 5.")
```

Write a program to accept a number from 1 to 10 from the user and display its ordinal value. For example, if the input was 5 the output would be 5th.

Outline how this program could be designed to display the ordinal value of *any* number entered

Write a program to accept a day number (e.g. 0-6 for Sun-Sat) and output the corresponding day name as a string. For example, if the user entered 0 the program would output *Sunday*; if the user entered 3 the program would output *Wednesday* and so on.

The program flowchart is illustrated below.



Write a program to prompt a user to enter a month number and then display the name of the month. For example, if the user enters 5 the program will display *May*.

14. The intention of the program below is to
15. display a student grade based on a
16. percentage mark entered by the user.
17.
18. The table on the right illustrates how marks are mapped to grades.

Mark (%)	Grade
85 or over	A
70 or over	B
55 or over	C
40 or over	D
Over 0	Fail

19. Study the code carefully and answer the questions which follow.

20.

```
1  mark = input("Enter a student percentage mark")
2  mark = int(mark)
3
4  if (mark >= 0):
5     print("Fail")
6  elif (mark >= 40):
7     print("D")
8  elif (mark >= 55):
9     print("C")
10 elif (mark >= 70):
11    print("B")
12 elif (mark >= 85):
13    print("A")
```

21. Why will the program not do what it is intended to do?

_____

_____

_____

22. Suggest a solution to the problem. Implement your solution.

_____

_____

_____

Modify the (fixed) code from the previous question so that it uses a variable called grade and just one print statement at the end that displays the value of grade.

Write a program to prompt a user to enter a mark and, based on the information provided in the table, display the appropriate points value.

Mark (%)	Grade
90 or over	100
75-89	85
60-74	70
45-59	55
40-45	40
Less than 40	0

The following program calculates gross pay based on number of hours worked and the hourly rate.

The end-user is prompted to enter the number of hours worked and the hourly rate of pay. Time and a half applies to all hour worked between 40 and 50 hours and double time applies to every hour worked in excess of 50.

```
1  def calcGrossPay(hoursWorked, hourlyRate):
2      basicPay      = 0.0
3      timeAndHalfOT = 0.0
4      doubleTimeOT  = 0.0
5
6      if (hoursWorked <= 40):
7          basicPay = hourlyRate*hoursWorked
8      elif (hoursWorked <= 50):
9          basicPay = hourlyRate*40
10         overtime = hoursWorked - 40
11         timeAndHalfOT = (1.5*hourlyRate)*overtime
12     else:
13         basicPay = hourlyRate*40
14         timeAndHalfOT = (1.5*hourlyRate)*10
15         overtime = hoursWorked - 50
16         doubleTimeOT = (2*hourlyRate)*overtime
17
18     return (basicPay + timeAndHalfOT + doubleTimeOT)
19
20 # Program execution starts here
21 hours = input("Enter hours worked:")
22 hours = int(hours)
23
24 rate = input("Enter hourly rate of pay:")
25 rate = float(rate)
26
27 grossPay = calcGrossPay(hours, rate)
28 grossPay = round(grossPay)
29
30 print("Gross Pay is:", grossPay)
```

`overtime` is the number of hours over 40 worked

`overtime` is the number of hours over 50 worked

Now complete the table below for the sample inputs provided. You should use a calculator to complete the expected output column. The Actual Output column can be completed by running the program.

Sample Input		Expected Output	Actual Output
Hours Worked	**Hourly Rate (€)**	**Gross Pay (€)**	**Gross Pay (€)**
20	€10		
40	€10		
50	€10		
65	€10		

Chapter Exercises (Decisions)

Write a program that prompts a user to enter their age. If the value entered is 18 or over the program should display the message *Adult*; otherwise *Child*.

Write a program to read in an integer and display the message *Valid month* if the integer is between 1 and 12 inclusive. If the number entered is not between 1 and 12 the program should display the message *Sorry. The value you entered is an invalid month*.

Write a program to display the sign (i.e. POSITIVE or NEGATIVE) that would result from the product of two integers (without calculating the product – you should verify your program works by having the program perform the actual calculation).

Write a program that initialises a variable called accountBalance to be 1000. The program should then simulate a cash withdrawal request by prompting a user to enter a withdrawal amount. If the amount requested is less than the balance the program should print the message Sorry. Insufficient funds. The program should end with the message *Do you require another transaction (Y/N)?*

Write a program that examines two integer variables and exchanges their values if the first one is greater than the second one

Write a program that reads in and displays the largest of three integers

Let use assume for the purpose of this exercise that passwords can only be numbers (integers). Write a program that initialises a variable called password to the value 9999. The program should then prompt a user to enter a number (i.e. password). If the value entered is the same as 9999 the program should display the message *User authenticated*. Otherwise, the program should display the message *WARNING: Intruder alert!*

Write a program that asks a user whether s/he wants to add (0), subtract (1), multiply (2) or divide (3). The program should then ask for two numbers to be input and, based on the code entered, perform the appropriate operation. (See iteration exercise for continuation.)

Write a program that prompts a user to enter a percentage mark (between 0 and 100) and depending on the mark entered display the appropriate grade as follows: Distinctions (80 or greater), Merit (between 65 and 80), Pass (between 50 and 65) and Unsuccessful (less than 50).

Write a program to prompt a user to enter a month number and then display the number of days in that month. (Assume February has 28 days.) For example, if the user enters 3, the program will display the message - *This month has 31 days*.

Write a program to prompt a user to enter a month number and then display the number of days in that month using the name of the month in the output. (Assume February has 28 days.) For example, if the user enters 3, the program will display the message – *March has 31 days*.

Write a program that reads a date as three integers (day, month and year) from the keyboard. The program should output the message *Valid* if the date is valid; *Invalid* otherwise. A valid date is any date spanning from 01/01/00 up to the current date. (Ignore the possibility of leap years.)

Write a programme that prompts a user to enter a year and display whether the year entered was (or will be) a leap year. A year is defined to be leap if it is divisible by 4 but not by 100. If a year is divisible by 4 and by 100, it is not a leap year unless it is also divisible by 400. (For example, 1800 and 1900 are *not* leap years while 1600 and 2000 *are*.)

Thus, years such as 1996, 1992, 1988 and so on are leap years because they are divisible by 4 but not by 100. For century years, the 400 rule is important. Thus, century years 1900, 1800 and 1700 while all still divisible by 4 are also exactly divisible by 100. As they are not further divisible by 400, they are not leap years.

Ordinal numbers are the words representing the rank of a number with respect to some order, in particular order or position (i.e. *first*, *second*, *third*, etc.). Ordinal numbers are alternatively written in English with numerals and letter suffixes: 1st, 2nd or 2d, 3rd or 3d, 4th, 11th, 21st, 101st, 477th. Write a program that accepts a number as input and prints out its ordinal value by concatenating an appropriate suffix to the number that was inputted. The following table lists some example inputs and outputs:

Input	Output
1	1st
2	2nd
3	3rd
12	12th
21	21st

Write a program that prompts the user to enter a time (e.g. HH:MM – two separate inputs may be needed) and display the time in words e.g. 9:40 “twenty to ten”.

Write a program that reads two integers (day and month) and prints the corresponding Zodiac sign based on the data provided.

SIGN	FROM	TO
Capricorn	December 22	January 19
Aquarius	January 20	February 17
Pisces	February 18	March 19
Aries	March 20	April 19
Taurus	April 20	May 20
Gemini	May 21	June 20
Cancer	June 21	July 22
Leo	July 23	August 22
Virgo	August 23	September 22
Libra	September 23	October 22
Scorpio	October 23	November 21
Sagittarius	November 22	December 21

Chapter 6

Introduction to Loops

LEARNING OBJECTIVES

1. Objective 1.
2. Objective 2.
3. Objective 3.

LECTURE DISCUSSION
REPETITION, REPETITION – PYTHON LOOPS

We can build real power into our programs by using loops. Loops are the topic of this section.

Let's say – for some strange reason – that we wanted to display the string *"Hello World"* 10 times.

Although the code shown below does the job, it looks a bit strange (or at least it should do!) with the same line being repeated 10 times. There must be a better way.

Figure 6.1: Print("Hello World")

Thankfully, a better solution is possible using a loop.

A **loop** is a block of code that gets executed more than once.

We can see from the above that we would like the line `print("Hello World")` to be executed 10 times. Before coding a loop we must first recognise that a loop is needed. This is the difficult bit. The easy bit is actually writing the loop in Python.

There are two types of loops in Python – while loops and for loops. For the moment we will stick with while loops.

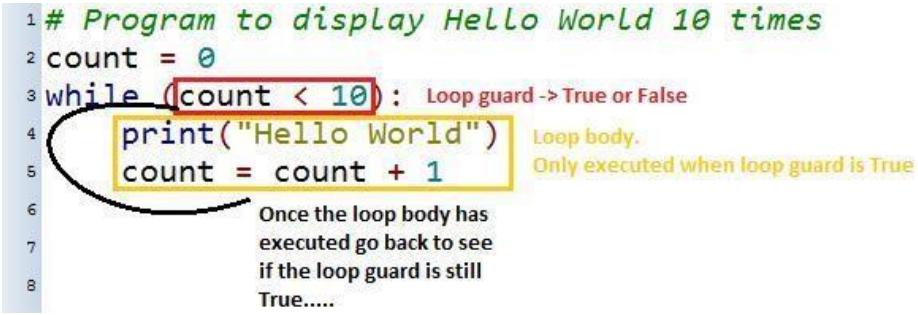
The program below uses a while loop to display the string *"Hello World"* 10 times.

Figure 6.2: "Hello World" in While Loop

Study the program carefully and see if you can figure out how it works.

Let's take a closer look at the code.

Figure 6.3: How While Loop works…



]

The working of the loop hinges on the variable count. As long as the value of count is less than 10 the program will display the text Hello World The value of count is increased by one on each loop iteration.

Here's an explanation of the above code in some more detail.

Line 2 initialises a variable called count to zero. The loop will end when count reaches 1
Line 3 is the start of the while loop.
count < 10 is a Boolean expression known as the **loop guard**
(the use of brackets around the loop guard is optional)
if the value of count is less than 10 the loop guard evaluates to True
if the loop guard evaluates to True the loop body is executed
line 4 displays the text *Hello World*
line 5 increases the value of count by 1
after line 5 execution loops back to line 3

In this example count is known as the **loop variable**. The loop variable is important because it controls the number of times the loop will be executed. Here the value of count will eventually reach 10 at which point the loop guard will evaluate to False and the loop body will not be executed.

A loop works by testing the value of the loop variable on each cycle.

When the above code is executed the following output is displayed:

```
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
```

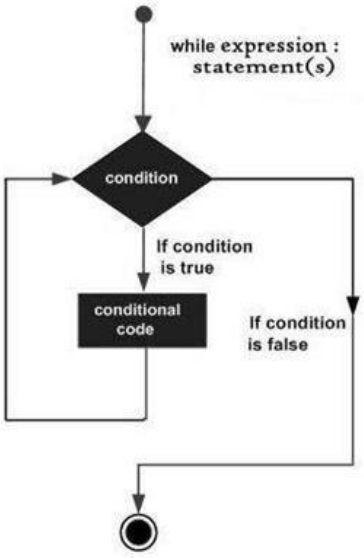
The while loop - syntax and semantics

The syntax of the while loop is as follows:

```
while <expression>:
<statement(s)>
```

This syntax is illustrated in the following flow diagram.

Figure 6.4: Flow Diagram of While Loops



In Python, while loops start with the keyword while.

This is followed by a Boolean expression i.e. an expression which evaluates to either True or False. If the expression evaluates to True the statement(s) are executed. If the expression evaluates to False then the statement(s) are skipped.

Because the expression guards entry into the loop it is called the **loop guard**. Every loop must have a loop guard. *Think of a loop guard as a green-red signal system. Green means enter the loop; Red means bypass the loop.*

Note the use of colon (:) at the end of the line

The statement(s) section of a loop is called the **loop body**. The loop body is *conditional code* and can consist of one or more lines. Each line of code in a loop body must be indented.

Once the last line of the loop body has been executed the flow of control loops back to the loop guard. This is called an **iteration**. The loop guard is re-evaluated and as before the loop body will be executed if the result is True; otherwise the loop body will be skipped and execution will continue from the next line after the loop body

*Remember! The **loop body** is executed as long as the **loop guard** remains true.*

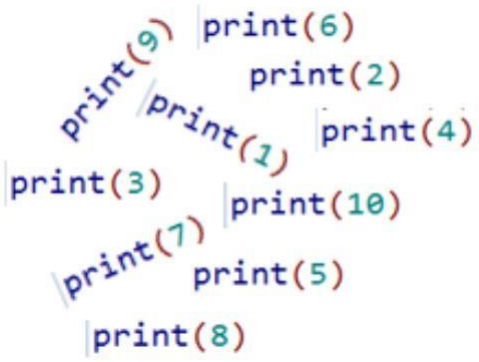
Notice that the loop body (i.e. the conditional code) of the *while* loop might not ever be executed. When the condition is tested and the result is False, the loop body will be skipped and the first statement after the loop body will be executed.

Loop Design Tips

1. Recognise when you need to write a loop. If a block of code might need to be executed more than once you will need a loop - the block of code will be the **loop body**
2. Decide on a **loop guard** - under what condition do you want to execute the loop body? (this might be a counter)
3. Make sure your loop body has a line that will cause the loop guard to eventually become `False` (this might be adding 1 to a counter)

Class Activity – counting from 1 to 10

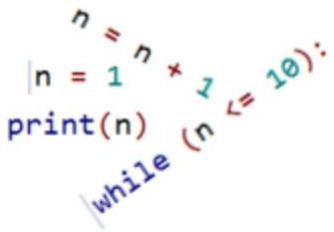
1. Re-arrange the lines of code below to display the numbers 1 to 10 vertically



2. How might a better solution be achieved?

---

3. Now re-arrange the 4 lines below so that the same output is generated



4. Describe in your own words the difference between a *loop guard* and a *loop body*

More on the loop guard

At runtime, the loop guard controls exactly how often the loop body is executed.

Small changes to the loop guard can cause the loop body to be executed either more often or less often than intended by the programmer. Study the examples below carefully.

	Code	Explanation	Output
(i)		The loop variable is `n` The loop guard is `n < 10` The loop body is on lines 3 and 4. The loop body is executed exactly 9 times i.e. while the counter, `n` is *less than 10*. The output is that the numbers 1 to 9 are displayed. On the final iteration the value of `n` is 9. The number 9 is displayed (line 3) and the value of `n` is increased from 9 to 10 (line 4). When the loop guard is tested again it evaluates to `False` (because `n` is no longer less than 10) and so the loop body is not executed.	1 2 3 4 5 6 7 8 9
(ii)	1 n = 1 2 while (n < 11): 3 print(n) 4 n = n + 1	The loop variable is `n` The loop guard is `n < 11` The loop body is on lines 3 and 4. In this example, all the numbers from 1 to 10 are displayed. This is because the loop body is executed as long as the value for `n` is *less than* 11. This only becomes `False` after 10 has been displayed and the value of `n` is changed to 11.	1 2 3 4 5 6 7 8 9 10

The three programs below all display the numbers 1…5 vertically. See if you can figure out how they work.

Figure 6.5: More on ther loop guard

```
n = 1
while (n < 6):
    print(n)
    n = n + 1
```

```
n = 1
while (n <= 5):
    print(n)
    n = n + 1
```

```
n = 0
while (n < 5):
    print(n+1)
    n = n + 1
```

Even though the loop guards are different the three programs all do the same thing

Exercises 1

Key in the programs listed below and see if you can figure out what they do.

Code	Explanation / Output
(i)	
(ii)	
(iii)	
(iv)	
(v)	
(vi)	
(vii)	

```
n = 1
while (n < 12):
    print(n)
    n = n + 1
```

```
n = 1
while (n <= 12):
    print(n)
    n = n + 1
```

```
n = 1
while (n <= 12):
    n = n + 1
    print(n)
```

```
n = 0
while (n < 12):
    print(n)
    n = n + 1
```

```
n = 5
while (n < 15):
    print(n)
    n = n + 1
```

```
n = 1
while (n <= 20):
    print(n)
    n = n + 2
```

```
n = 10
while (n > 0):
    print(n)
    n = n - 1
```

Never ending loops (Infinite Loops)

Loops that execute forever without ever terminating are called *infinite loops*. Although there are some cases where infinite loops are deliberately written into software systems, in general they should be avoided by novice programmers.

Care should always be taken by the programmer to ensure that a loop will – at some stage – come to an end. In other words, it is the programmer's responsibility to write the necessary code that will eventually cause the loop guard to evaluate to `False`. Consider the two small example programs illustrated below:

Figure 6.6: Terminating Loop

```
1 n = 1
2 while (n <= 5):
3     print(n)
4     n = n + 1
```

Program 1: Terminating Loop

The above program displays the numbers 1 through to 5 inclusive. The loop is only executed as long as the variable n is less than or equal to.

23. At the end of each loop iteration the value of n is increased by 1 (line 4). This guarantees that the loop will terminate at some stage.

Figure 6.7: Infinite Loop

```
1 n = 1
2 while (n <= 5):
3     print(n)
```

Program 2: Infinite Loop

The above loop is only executed as long as the variable n is less than or equal to 5. The variable n is initialised to 1 (line 1) but after that, the program never changes the value of n anywhere. Therefore, the loop guard will always evaluate to True and the loop will execute forever i.e. infinite loop. In summary, the program will display a never ending list of 1's.

Warning! Once you start a program that contains an infinite loop it can be difficult to stop. (On Windows systems you may need to use the Task Manager to end the task.)

Loops that do nothing

It can happen that loops sometimes do not get executed even once. Consider the example below.

Figure 6.8: Loops that do nothing

```
1 n = 1
2 while (n > 10):
3     print(n)
4     n = n + 1
```

Here, the loop body (i.e. lines 3 and 4) never get executed. This is because the condition n>10 will never evaluate to True (as n has just been initialised to 1 on the previous line).

In this example the loop can *never* be executed which of course makes no sense.

The example is just fabricated to illustrate the point that *sometimes* loops are not executed because of particular runtime circumstance that causes the loop guard to evaluate to False

the first time it is tested

Counter controlled repetition

Quite often a counter variable is use to control the loop guard. The counter is initialised before the loop starts and the last line of the loop body changes the value of the counter. This is called counter controlled repetition and is useful for situations where the number of iterations is fixed or known in advance of running the program.

Displaying numbers between 1 and x

Earlier we looked at displaying the numbers from 1 to 10. We now consider the problem of how to display all the numbers between 1 and some variable value entered by the user.

To solve this problem we introduce a variable called max. We prompt the user to enter a value and store it in max. The loop is written as before, with one difference. This time instead of ending when the counter reaches 10, the loop continues until the counter reaches max.

This is illustrated in the program below.

```
n = 1
max = input("Enter the upper limit")
max = int(max)
while (n <= max):
print(n)
n = n + 1
```

If the user enters 4 the program will display the following output:

```
1
2
3
4
```

Counting backwards

In this example we examine a program that displays all the whole numbers backwards between two numbers entered by the user - max and min.

```
   max = input("Enter the upper limit")
   max = int(max)
   min = input("Enter the lower limit")
   min = int(min)
5.
   while (min <= max):
   print(max)
   max = max - 1
```

Program Listing

If the user entered 12 for `max` and 8 for `min` the program will display the following output.

```
12
11
10
9
8
```

Sample Output

Look carefully at how – on each iteration of the loop - line 8 decrements (subtracts 1 from) the value stored in the variable max. This continues until eventually the value of max becomes less than the value of min and the loop guard min <= max evaluates to False

Exercises 2

1. Key in the programs listed below and see if you can figure out what they do.

Code	Explanation / Output
(i) `min = input("Enter the lower limit")` `min = int(min)` `while (min <= 100):` `    print(min)` `    min = min + 1`	
(ii) `min = input("Enter the lower limit")` `min = int(min)` `max = input("Enter the upper limit")` `max = int(max)` `while (min <= max):` `    print(min)` `    min = min + 1`	
(iii) `n = 1` `while (n <= 10):` `    print(n+1)` `    n = n + 1`	
(iv) `n = 1` `while (n <= 10):` `    print(n+1)`	
(v) `n = 0` `while (n <= 100):` `    print(n)` `    n = n + 5`	

In the last three example listings above, what is the value of the variable n after the loop has terminated?

Write a program that displays all the whole numbers between two numbers entered by the user max and min.

Write a program that displays all the numbers from 100 down to 1 inclusive.

Write a program that displays every 10^{th} number between zero and 100 inclusive i.e. 0, 10, 20, 30 etc.

Given the following code to add two random numbers between 1 and 10, write a program that adds two random numbers three times. (Hint: You will need to wrap the code provided inside a loop body and execute the loop three times.)

```
import random

n1 = random.randint(1, 10)
n2 = random.randint(1, 10)
print(n1, "+", n2, "=", n1+n2)
```

Using variables inside loops

The loop variable is important because it controls the number of loop iterations.

Thus far, we have not been doing very much with our loop variable. (Most examples so far just display the contents of the variable and either increment or decrement it.) In the following examples we will develop the possible uses of loop variables.

Computing squares

The program below loops over a variable `n`, to compute and display the following table of values.

Figure 6.9: Computing Squares

n	n*10	n*100	n*1000
1	10	100	1000
2	20	200	2000
3	30	300	3000
4	40	400	4000
5	50	500	5000

Before looking at the solution think about it. Try to answer the following questions.

Does the first row (i.e. the row header) need to be printed inside a loop?

How many times does the loop body get executed to produce the output shown? What is the loop variable?

Which values are calculated by the program?

For each of the 5 data rows what calculation is being carried out? How are the columns so neatly aligned?

The solution is illustrated as follows.

```
n=1
print("n\tn*10\tn*100\tn*1000")
while (n<=5):
       print(n, "\t", n*10, "\t", n*100, "\t", n*1000)
n=n+1
```

Notice the use of '\t' in the above program. '\t' is an example of a special character called an **escape sequence**. Python interprets the '\t' escape sequence as a tab character. Therefore, every time a '\t' character appears in a print statement Python prints a tab in its place. It is through the use of the '\t' character that the data displayed in output columns are kept so neatly aligned.

Some other common escape sequence characters are illustrated in the table below:

Table 6.1: Escape Sequence

Escape Sequence	Meaning
\n	Newline
\t	Tab
\’	Single Quote
\”	Double Quote

Times tables
In this example, we will develop a program to display the 7 times tables (up to 12).

Think about it – we need to display all the lines from 7 × 1 = 7 up to 7 × 12 = 84. The desired output is shown on the right.

```
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63
7 x 10 = 70
7 x 11 = 77
7 x 12 = 84
```

In all of these output lines the first number is constant i.e. 7.

The second number is variable i.e. it varies from 1 up to 12. This could be our loop counter - n.

The value on the right hand side of the equals sign is calculated by multiplying 7 by n.

Each output line is generated by the statement below.

```
print("7 x", n, "=", 7*n)
```

Remember, the parts inside the quotations are string literals and as such are not interpreted by Python. The pieces outside the quotations are variables or expressions. These are evaluated by Python and the result of the evaluation is displayed.

The full program to display the 7 times is now given:

Figure 6.10: Program listing: 7 times tables

```
2 n=1
3 while (n<=12):
4     print("7 x",n,"=",7*n)
5     n=n+1
```

Program listing: 7 times tables

In the program listing shown below we improve our ‘7 times program’ by asking the user what times tables they wish to have displayed.

Figure 6.11: 7 times program

```
tables = input("What times tables do you wish to see?")
tables = int(tables)
n=1
while (n<=12):
    print(tables, "x",n,"=",tables*n)
    n=n+1
```

Key it in and try it out for yourself

Exercises 3

1. Write a program that generates the '4 plus' addition table depicted below.

```
4 + 0 = 4
4 + 1 = 5
4 + 2 = 6
4 + 3 = 7
4 + 4 = 8
4 + 5 = 9
4 + 6 = 10
4 + 7 = 11
4 + 8 = 12
4 + 9 = 13
4 + 10 = 14
4 + 11 = 15
```

2. Modify the program just written to generate the addition table for any number entered by an end user

3. Write a program that displays a table of the squares and cubes of the first 10 numbers as illustrated below:

```
n    n squared   n cubed
1    1           1
2    4           8
3    9           27
4    16          64
5    25          125
6    36          216
7    49          343
8    64          512
9    81          729
10   100         1000
```

4. Given the formula below to convert from degrees Celsius to Fahrenheit, write a program to calculate and tabulate Fahrenheit values for all Celsius values between zero and 100 in steps of 10.

The output should look something like this:

```
C    F
0    32.0
10   50.0
20   68.0
30   86.0
40   104.0
50   122.0
60   140.0
70   158.0
80   176.0
90   194.0
100  212.0
```

Accumulating Totals

We now turn our attention to using loops to accumulate or add values

Adding the first 10 numbers
The program below adds all the numbers between 1 and 10 inclusive and then displays the answer. In essence it carries out the following calculation.

1+2+3+4+5+6+7+8+9+10

```
print(1+2+3+4+5+6+7+8+9+10)
```

As a slight improvement we will introduce a variable - total, to store the result and display a more meaningful message.

```
total = 1+2+3+4+5+6+7+8+9+10
print("The sum of the first 10 natural numbers is", total)
```

Although, the programs above work, their use is very limited. A better and more general solution can be written as follows using a simple while loop.

```
24.      # Program to add the first 10 natural numbers
25.      total=0
26.      counter=1
27.      while (counter <= 10):
28.    total=total+counter
29.    counter=counter+1

30.      print("The sum of the first 10 natural numbers is", total)
```

The key to understanding the above code lies in the use of the variable total. Notice that total is initialised to zero before the loop begins execution. We focus in particular on line 5. Notice how the variable total appears on both sides of the assignment operator (=). Line 5 calculates a new value for total by adding its old value to the value of counter. The table below shows how a new value for total is calculated on each iteration of the while loop.

Table 6.2

Iteration (`counter`)	`total` (before assignment in line 5)	`total` (after assignment in line 5)
1	0	1
2	1	3
3	3	6
4	6	10
5	10	15
6	15	21
7	21	28
8	28	36
9	36	45

10	45	55

The technique of using a variable to accumulate a total is quite common (and therefore important) in computer programming. (As an example, think about how a retail point of sale system might calculate the total bill.)

Finally, by the time loop ends (i.e. the value of counter is no longer less than or equal to 10) the value stored in variable total is the sum of the first 10 natural numbers. Line 7 displays the result.

Calculating a class average mark

Consider the following scenario. A class of ten students took a test. Imagine that the percentage marks (integers in the range 0 to 100) for the ten students are available to you. We need to write a program to determine the class average for the test.

The program should prompt the user to enter each of the ten marks. It will need to keep a running total of all the marks. Once the total is known the average can be calculated, simply by dividing the total by 10.

We use pseudo-code as an initial step in the development of our solution.

```
Initialise the total mark to zero
Initialise a counter to 1

Loop 10 times (while the counter is less than or equal to 10)
   Prompt the user to enter a student mark
   Convert the mark from a string to an integer (so that it can be added)
   Add the mark just entered to the total mark
   Add one to the counter

Calculate the average by dividing the total mark by 10
Display the result
```

The Python solution is shown below – except ***the lines are all jumbled up and not indented properly***. Can you re-arrange into a working solution?

Figure 6.12: Code not indented properly

```
average = total/10
counter=counter+1
print("The average mark is", average)
mark = input("Please enter a student mark")
total=0
# Program to compute class average
total = total + mark
counter=1
mark = int(mark)
while (counter <= 10):
```

Describe the role of the variable `total` in the final solution.

Accumulating strings
The code below display the digits 1 through to 9 vertically i.e. each digit is displayed on a separate line.

```
counter = 1
while (counter < 10):
   print(counter)
   counter = counter + 1
```

Let us consider how the digits 1 through to 9 could all be displayed (using a loop) on the same line i.e. horizontally as shown below.

```
123456789
```

The problem with the above solution is that the program moves onto a newline each time the digit (i.e. the value of counter) is displayed. If we want to display all the digits on a single line then we need a solution that calls the print() function just once. The call to print() will have to be after the loop body. The loop itself just builds up the output to display.

We introduce a variable called output. Initially output is set to be an empty string (i.e. "").

On each iteration of the loop the value of counter is concatenated (i.e. added) to output. The table below illustrates how the value of output changes with each loop iteration.

Figure 6.13: Loop number pattern

```
1
12
123
1234
12345
123456
1234567
12345678
123456789
```

By the time the loop is finished the value of output has been accumulated to the desired string . The solution is shown below.

```
31.      output=""
32.      counter = 1
33.      while (counter < 10):
34.    output = output + str(counter)
35.    counter = counter+1
36.      print(output)
```

Line 4 is the key line. This line builds up the output string by adding the current value of counter to the output string accumulated so far. Since counter is a numeric type it must be converted to a string so that it can be added to output. The build in function str() does this by converting counter to a string.

Notice from the listing shown on the left hand side below, that with a very slight change we can generate the output displayed on the right.

Figure 6.14: How to do loop number pattern

```
1 output=""
2 counter = 1
3 while (counter < 10):
4     output=output+str(counter)
5     print(output)
6     counter = counter+1
```

Program listing

```
1
12
123
1234
12345
123456
1234567
12345678
123456789
```

Output

In the above listing the call to `print()` has been moved inside the loop. Therefore, the contents of `output` are displayed on each loop iteration.

Addendum - n+1 problem
An interesting problem arises if we stipulate that each digit in the output should be separated by a comma. In other words the desired output becomes:

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

At an initial glance a solution might seem trivial – just modify line 4 of our original solution to append a comma after each digit as follows:

```
37.      output=""
38.      counter = 1
39.      while (counter < 10):
40.    output = output + str(counter) + ","
41.    counter = counter+1
42.      print(output)
```

However, this would result in a string with an extra, unwanted comma at the end as follows:

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

So, what about changing the code to add a comma in before each digit?

```
43.      output=""
44.      counter = 1
45.      while (counter < 10):
46.    output = output + "," + str(counter)
47.    counter = counter+1
48.      print(output)
```

This just moves the extra, unwanted comma to the start of the string as shown:

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

The problem is that we have 9 digits but only need 8 commas. In other words there is a difference of one between the number of iterations required to display the digits and the commas.

This type of situation occurs quite often in computer science i.e. iterations are required to do one job and + 1 iterations are required to do another.

In the case of this example a solution can be achieved by using the loop to accumulate the output string following the sequence - *digit followed by comma* - up to but not including the digit 9.

```
49.      output=""
50.      counter = 1
51.      while (counter < 9):
52.    output = output + str(counter) + ","
53.    counter = counter+1
54.      output = output + str(counter)
55.      print(output)
```

Once the loop has finished, line 6 completes the construction of the output string by adding the digit 9 - without the trailing comma.

An alternative solution – shown below – works by building up the output string using the sequence *comma followed by digit* – from digit 2 onwards.

```
56.      output="1"
57.      counter = 2
58.      while (counter < 10):
59.    output = output + "," + str(counter)
60.    counter = counter+1
61.      print(output)
```

Figure it out
Try running the following code.

Figure 6.15: Try this code!

```
1 sentence=""
2 counter = 1
3 while (counter <= 5):
4    word = input("Enter a word:")
5    sentence = sentence + word + " "
6    counter = counter+1
7 print(sentence)
```

Can you figure out what it does? The following questions might help you.

What are your inputs?
What is the output?

Exercises 4

1. Write a program that uses a loop to sum (i.e. add) the first 100 natural numbers.

2. Write a program that sums all the numbers from 1 to x where x is some number entered by the end-user. For example, if the end-user entered 12, the program would compute and display the result of 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + 11 + 12.

3. Write a program that sums all the numbers from x to y where both x and y are two numbers entered by the end-user. For example, if the end-user entered 8 and 13 the program would compute and display the result of 8 + 9 + 10 + 11 + 12 + 13.

4. The reciprocal of a number x is denoted by 1/x. For example, the reciprocal of 5 is 1/5. Write a program to sum the reciprocal of the first 10 natural numbers i.e.

$$\frac{1}{1}+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\frac{1}{5}+\frac{1}{6}+\frac{1}{7}+\frac{1}{8}+\frac{1}{9}+\frac{1}{10}$$

5. Develop a 'mini point of sale system' to calculate the total amount due based on the items –quantities and prices – listed in the table below.

Item	Number of Items	Price (€)
1	2	4.99
2	5	0.57
3	1	7.99
4	6	0.99
5	1	2.49

6. The factorial of a non-negative integer n, denoted by *n*!, is the product of all positive integers less than or equal to n. For example, 5! = 5 × 4 × 3 × 2 × 1 = 120. Write a program that prompts a user to enter a number and then compute and display its factorial

7. Write a program that uses a loop to display the output shown below.

```
1 +
2 +
3 +
4 +
5 +
= 15
```

8. Now modify your program so that the output is accumulated as a string and displayed on a single line as follows:

```
1 + 2 + 3 + 4 + 5 +  = 15
```

9. Finally, enhance the program so that the final plus sign is not displayed i.e. the output should look like:

```
1 + 2 + 3 + 4 + 5 = 15
```

Indefinite Loops and Sentinels

Thus far we have been using a counter to control the number of loop repetitions. The main limitation of counter controlled repetition is that the number of iterations is fixed. There are many situations where the programmer does not know the number of iterations required in advance.

Consider for example the problem of adding an unknown number of numbers.

The challenge here is to develop a loop guard that allows the loop to be executed a variable number of times.

```
Initialise the total to zero
Prompt the user to enter the first number

Loop as long as there are more numbers to add
   Add the number to the running total
   Prompt the user to enter another number


Display the total
```

What exactly does 'as long as there are more numbers to add' mean? To answer this question we need to understand *sentinels*.

A **sentinel** is a special type of loop guard used to end a loop when the number of iterations is not known before the program is run. The sentinel value is decided upon by the programmer and used to terminate a loop.

In the case of this example, if we choose zero as the sentinel it will mean that the loop should end when the user enters the number zero. A programmer needs to be careful not to choose a sentinel that is a possible legitimate value (and so should be processed like every other value by the loop). Since the purpose of this program is to add a list of numbers, it seems reasonable to use zero as a sentinel.

The use of the sentinel can be seen in line 5 of the Python implementation shown below.

```
62.      # Program to add any number of numbers
63.      total=0
64.      num = input("Please enter a number (0 to end)")
65.      num = int(num)
66.      while (num != 0):
67.    total=total+num
68.    num = input("Please enter a number (0 to end)")
69.    num = int(num)

70.      print("The sum of the all the numbers entered is", total)
```

You should verify the program works by keying it in and running it. Notice how lines 3 and 7 both prompt the user to enter a number. In both cases the end-user is reminded that the sentinel is zero. Line 6 keeps the running total by adding the number just entered to the total so far.

It is quite common to maintain a counter in a sentinel controlled loop. The main reason for this is that by the time the loop has finished execution the counter can be used to indicate how many iterations of the loop took place.

For example, if we wished to calculate the average of an unknown number of values – each entered by the user – we would need to divide by the counter once all the values had been added. This is illustrated in the program below.

```
71.      # Program to average any number of numbers
72.      total=0
73.      counter = 0
74.      num = input("Please enter a number (0 to end)")
75.      num = int(num)
76.      while (num != 0):
77.    counter = counter+1
78.    total=total+num
79.    num = input("Please enter a number (0 to end)")
80.    num = int(num)

81.      print("The sum of the all the numbers entered is", total)
82.      average = total/counter
83.      print("The average of the all the numbers entered is", average)
```

Notice that the variable `counter` is initialised to zero (line 3) and, how on every iteration of the loop the `counter` is incremented by 1 (line 7). Finally, when the loop is finished, the `counter` is used to compute the `average` (line 12).

Sentinels Summary
A sentinel is a value used to terminate a loop. The sentinel is chosen by the programmer and forms part of the loop guard which controls access to a loop.

There should be no chance that the value chosen as the sentinel could be a legitimate value that should be processed by the loop body. For example, if the loop body is dealing with numbers greater than zero, then the sentinel could be either zero or any negative number.

In general, the psuedo-code for a sentinel loop looks something like this

```
read the first value
while value is not the sentinel:
   process the value
   read the next value
```

Exercises 5

1. Write a program that calculates and displays the product of a variable list of numbers entered by the user. The program should end when the user enters 0 (i.e. the sentinel is 0.)

   If you are unsure just implement the following pseudo-code.

```
Initialise product to one
Read the first number
while number is not equal to zero:
   calculate the product (i.e. product=product*number)
   read the next number
Display the total product of all the numbers entered
```

   Why is the variable `product` initialised to 1 in this example?

2. Write a program that repeatedly reads a positive number and displays its square root. The program should end when the user enters −1 (i.e. the sentinel is −1.)

   If you are unsure you should modify the following mixture of Python and pseudo-code.

```
import math

read the first number
while number is not the sentinel:
   root = math.sqrt(number)
   display the root
   read the next number
```

   Can you think of a better sentinel? (*Hint:* any negative number)

3. Write a program that selects a random number between 1 and 10 and asks the user to repeatedly enter a number until s/he has guessed the random number.

   If you are unsure you should modify the following mixture of Python and pseudo-code.

```
import random

generate a random number between 1 and 10
read the first guess from the user
while the random number is not the same as the guess:
   read the next guess
```

4. Write a program that generates two random numbers (say between 1 and 20) and asks the user to enter their sum (or product). The program should continue until the user enters the correct answer.

5. Enhance the last two solutions so that the user has at most three chances.

Using loops to validate data

One common use for loops and sentinels is to validate date. Consider for example the following pseudo-code used to ensure some value entered by the user is valid.

```
84.    read value from end-user
85.    loop for as long as the value is invalid:
86.   display error message
87.   read value from end-user
88.    process value
```

Here the loop body (i.e. lines 3 and 4) is only executed if the condition in the pseudo-code i.e. 'value is invalid', is true. We need not concern ourselves with the details of how the program actually determines whether the value entered is valid or not. (Suffice to say that an actual implementation would need to contain the code that tests for the validation of value in question. The test would evaluate to true if the value is invalid and false otherwise.)

The important point to recognise is that the loop body may be executed zero or more times depending on whether the value entered by the user is valid or not. If the user enters a valid value the first time, loop body will not be executed. On the other hand, if the user enters an invalid value, the loop body will be executed – and continue to be executed – as long as the value entered by the end-user remains invalid. In this way the program guarantees, that by the time line 5 is reached, the value to be processed will be valid.

Examples – data validation
Study carefully the following two examples which illustrate how loops can be used to validate data.

In the first example, the end user is prompted to enter a response to the question "Do you wish to continue (Y/N)". The only valid (acceptable) responses are either "Y" or "N". The loop ensures that the user is repeatedly prompted as long as their answer is not equal to "Y" and not equal to "N". In this way the programmer can be sure that by the time the loop has exited the user has entered one of "Y" or "N" and can take appropriate action based on the user decision.

```
ans= input("Do you wish to continue (Y/N)")
while (ans != "Y" and ans != "N"):
    ans= input("Do you wish to continue (Y/N)")
```

In the second example, the program prompts the end user to enter a numeric value for a month. Since the only valid values are any number between 1 and 12 inclusive the program keeps looping until a value in this range has been entered.

```
month = input("Enter month number")
month = int(month)
while (month<1 or month>12):
    month = input("Invalid Month. Enter a number between 1 and 12")
    month = int (month)
```

Exercises 6

1. Study the code below carefully and see if you can figure out what it does.

```
print("Read the menu below and make your choice ......")
print("")
print("\t 1. Add")
print("\t 2. Subtract")
print("\t 3. Multiply")
print("\t 4. Divide")
print("\t 0. Exit")
print("")

choice = input("Enter your choice : ")
choice = int(choice)
while ((choice<0) or (choice>4)):
    choice=input("Enter your choice : ")
    choice=int(choice)

if (choice!=0):
    n1 = input("Enter the first number : ")
    n1 = int(n1)

    n2 = input("Enter the second number : ")
    n2 = int(n2)

    if (choice == 1):
        print(n1, "+", n2, "=", n1+n2)
    elif (choice == 2):
        print(n1, "-", n2, "=", n1-n2)
    elif (choice == 3):
        print(n1, "*", n2, "=", n1*n2)
    elif (choice == 4):
        print(n1, "/", n2, "=", n1/n2)
    elif (choice != 0):
        print("Invalid choice.")
```

2. Write a program that repeatedly adds two numbers entered by the user. At the end of each iteration the program should prompt the user with the message “Do you wish to continue (Y/N)”. The program should continue as long as the user answers “Y”.
3. Write a program that asks a user whether s/he wants to add (0), subtract (1), multiply (2) or divide (3). The program should then ask for two numbers to be input and, based on the code entered, perform the appropriate operation. At the end of each calculation the program should prompt the user with the message - *Do you wish to continue (Y/N)*. The program should end when the user enters N.
4. Suggest a possible validation rule for the following data values:
    - Student mark
    - PPSN
    - Vehicle Registration Number

Making decisions inside loops

From time to time it will be necessary for a running program to make a decision inside a loop body. This involves the inclusion of an if statement (or some variation such as if-else or if-elif-else) inside the loop body. The if statement is used to make a decision on each iteration of the loop.

Consider a simplified payroll scenario where tax is calculated at 20% on income less than €40,000 and 50% on income of €40,000 or greater. We wish to write a program that accepts the 5 incomes listed below and calculate the net pay for each income provided. For each income entered, the program should display the gross pay, the tax payable and the net pay.

Table 6.3: Gross income

Gross Income
€50,000
€10,000
€40,000
€25,000
€82,500

Our solution will use a loop to prompt the user for each of the 5 values. The loop will need to contain an if-else statement to decide whether the income is less than the threshold of €40,000 or not. If the income is less than €40,000 the code will calculate the tax to deduct based on the low rate of 20%. Otherwise, the tax will be calculated based on the 50% rate.

The full Python solution is presented as follows:

```
counter=1
while (counter <= 5):
   gross = input("Enter gross income:")
   gross = float(gross)
          if (gross < 40000):
             tax = gross * .2
   else:
      tax = gross * .5

   net = gross - tax
   print("Gross Pay:", gross)
   print("Tax:", tax)
   print("Net Pay:", net)

   counter=counter+1
```

Key in the above code and then run. Use the output to complete the table below:
Table 6.4: Gross income with Tax and Net Pay

Gross Income	Tax	Net Pay
€50,000		
€10,000		
€40,000		
€25,000		
€82,500		

Example – even and odd numbers

As another example illustrate the use of decisions inside loops this let us consider the problem of displaying all the even numbers between 1 and 100.

From the problem statement it should be clear that we are going to require a loop – to iterate over all the numbers between 1 and 100. A number is defined to be even if it is evenly divisible by 2. In other words, if we divide a number by 2 and the remainder is zero we can conclude that the number is even.

We can use this information to code the following Python condition to test for evenness. If the condition evaluates to True the program can conclude that n is even; otherwise n must be odd.

```
(n % 2 == 0)
```

The full solution presented below wraps the condition as part of an if statement inside a while loop.

```
89.     n = 1
90.     while (n <= 100):
91.   if (n % 2 == 0):
92.     print(n)
93.   n = n+1
```

For every value of n between 1 and 100, n is tested for evenness. If the condition evaluates to True, the value in the variable n is displayed. As there is no else statement, there is no processing carried out if n is not even. In other words odd numbers are ignored. Line 5 is executed regardless of whether n was even or not.

The program below displays all odd numbers between 1 and 100.

```
94.     n = 1
95.     while (n <= 100):
96.   if (n % 2 != 0):
97.     print(n)
98.   n = n+1
```

Can you spot the subtle difference between the two programs?

An interesting variation to the above is the following program which counts all the numbers between 1 and 100 that are evenly divisible by 4.

```
99.     count = 0
100.    n = 1
101.    while (n <= 100):
102.  if (n % 4 == 0):
103.    count = count + 1
6.
104.    print("There are", count, "numbers evenly divisible by 4 between 1 and 100")
```

Key it in and try running it!

Calling functions from loops

As a general rule of thumb the code inside a loop body should not be too long (i.e. it should not contain too many lines). In situations where the loop body is becoming too long, the programmer should consider putting some of the code into a function and calling the function from the loop body.

Let us wrap our code – from the previous page - to determine the 'evenness' of a number into a function.

```
105.    def isEven(number):
106.
107.  if (number % 2 == 0):
108.    return True
109.  else:
110.    return False
```

This is an example of a Boolean function because it returns a Boolean value i.e. either True or False. The function uses the remainder operator (%) to test whether the number passes in is even or not. If the value passed in as number turns out to be even the function returns True. Otherwise the function will return False.

To code below demonstrates how the above function could be used to display all the even numbers between 1 and 100.

```
111.    n = 1
112.    while (n <= 100):
113. if isEven(n):
114.    print(n)
115. n = n+1
```

Line 3 is key. Here the call to the function appears as part of a conditional statement. This is fine, since conditions evaluate to True or False and the function isEven(number) is guaranteed to return one of these values.

As before, with a very slight modification we can display all the odd numbers between 1 and 100.

```
116.    n = 1
117.    while (n <= 100):
118.  if !isEven(n):
119.    print(n)
120.  n = n+1
```

When a function call appears in a loop body, the function will be called on each iteration of the loop.

Question. How might the above code be modified to count all the even/odd numbers displayed?

Exercises 7

1. Write a program to iterate over all the numbers between 1 and 100. Every time it comes across a multiple of 5 it prints 'fizz', for every multiple of 10 it prints 'buzz', and for every other number it just prints the number.

2. Write a program that repeatedly prompts a user to enter a number. If the user enters an even number the program should display *Even*; otherwise, the program should display the message *Odd*. The program should end when the user enters zero.

3. Write a program that keeps a count of the number of even numbers entered by a user until zero is entered.

4. A prime number is a number that has exactly two factors, 1 and itself.

   The Boolean function below `isPrime` determines whether the number passed in is prime or not. The function will return `True` if `number` is a prime number; `False` otherwise.

```
1  import math
2
3  # A function to test whether a number is prime or not
4  # Returns True if the number is prime; False otherwise
5  def isPrime(numToCheck):
6
7      # Any number less than 2 is not prime
8      if numToCheck < 2:
9          return False
10
11     # see if num is evenly divisible by any number up to num/2
12     divisor = 2
13     while (divisor < numToCheck/2):
14         if (numToCheck % divisor == 0):
15             return False
16         divisor = divisor+1
17
18     # The number must be prime so return True
19     return True
```

This behaviour is illustrated in the following example calls to the function.

```
isPrime(13) --> True (because 13 is a prime number)
isPrime(10) --> False (because 10 is not a prime number)
```

a) Use the `isPrime` function to display all the prime numbers between 2 and 100 inclusive.
b) Use the `isPrime` function to count and display the number of prime

numbers between 2 and 1000.

c) Use `isPrime` to display the first 50 prime numbers.

5. A factor is a whole number which divides exactly into a whole number, leaving no remainder. For example, 5 is a factor of 20 because 5 divides exactly into 20 (20 ÷ 5 = 4 leaving no remainder. The complete list of factors of 20 is: 1, 2, 4, 5, 10, and 20 (all these divide exactly into 20). Write a program that prompts a user for a number and displays all that numbers factors.
6. Enhance the program from the previous exercise to print out the total number of factors the number entered has.
7. Maths Teacher. Write a program that generates two random numbers (between 1 and 100) and prompts the user to enter the sum of the two numbers. If the user enters the correct value display *Correct*. Otherwise display the message *Incorrect. Try again.* The program should give the user three chances and at the end of each cycle should ask the user – *Do you wish to continue (Y/N)*. The program should end when the user enters N.
8. Perfect Numbers. (Advanced)

These are numbers whose divisors (excluding the number itself) add up to the number. Excluding the number 1, the first perfect number is 6 because 6 = 3 × 2 = 6 × 1 and 6 = 3 + 2 + 1

In fact 496 is the third perfect number, and 8128 is the fourth. Although there are still many unknown results concerning perfect numbers, it has been shown that

(a) all **even** perfect numbers will be of the form

$$2^{n-1}\left(2^n-1\right)$$

when n is a prime number. This number is in fact perfect when 2^n-1 is prime;

(b) all even perfect numbers end in 6 or 8;

(c) the sum of the inverses of all divisors of a perfect number add up to 2

e.g. for 6, $\frac{1}{6}+\frac{1}{3}+\frac{1}{2}+\frac{1}{1}=2$.

Use the above information to write a program that displays a complete version of the table shown to the right.

n	2^n-1	prime	$2^{n-1}\left(2^n-1\right)$	perfect
2	3	√	6	√
3				
5				
7				
11				
13				

Can you spot any relationship between 220 and 284?

For loops

We mentioned at the beginning of this chapter that Python supports two different types of loop constructs – while and for. So far we have concentrated on while loops only. We now turn our attention to for loops.

Although, the semantics of while and for loops is the same, the syntax is substantially different. Let's jump straight into an example.

The program below uses a for loop to display all the numbers from 0 up to, but not including, 10.

```
121.     A program to iterate between 0 and 10 (excluding 10)
for num in range(0,10):
   print ('The number is:', num) # Notice the indentation

print("Good bye!") # This is outside the loop body
```

When the above code is executed, it produces following result:

```
The number is: 0
The number is: 1
The number is: 2
The number is: 3
The number is: 4
The number is: 5
The number is: 6
The number is: 7
The number is: 8
The number is: 9
Good bye!
```

In order to understand the above program it is first necessary to understand that range is a build-in function which returns a list of values. In the above example the call range(0,10) returns a list of all the numbers from 0 up to but not including 10 i.e.

range(0,10) -> 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 range(8,12) -> 8, 9,
10, 11

If only one argument to range is provided, the first argument is taken to be zero. Therefore,

range(5) -> 0, 1, 2, 3, 4

In the case of this example, the for loop works by iterating over each value in the sequence (i.e. 0 through to 9). At the start of each iteration the value of the next item in the sequence is assigned to the loop variable. In this example, the name of the loop variable is num.

Notice how at the start of each iteration the iterating variable is assigned the next value in the sequence. The loop ends when all the values in the sequence have been used.

Once the loop terminates, execution continues at the next line after the loop body.

The for loop - syntax and semantics

The syntax of a **for** loop look is as follows:

```
for <iterating_variable> in <sequence>:
  statements(s)
```

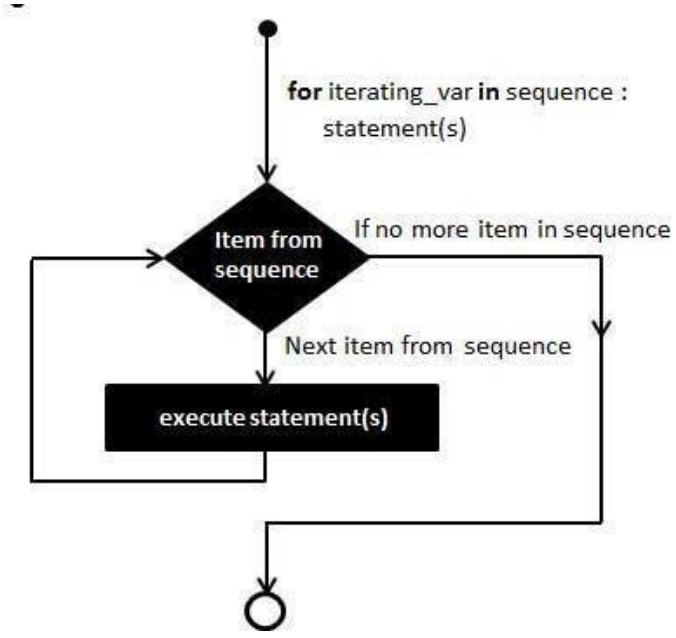
The sequence is typically generated by the range function.

The loop starts by assigning the first item in the *sequence* to the loop variable referred to as *iterating_var*. Next, the statement(s) that make up the loop body are executed. The loop continues the cycle of assigning the next item in the sequence to the iterating variable and then processing it in the loop body until the entire sequence is used up.

Observe the use of colon (:) and also that the statements which make up the loop body are indented.

The syntax of the Python for loop is illustrated in the following flow diagram.

Figure 6.16: For Loop Syntax Flow Diagram



for vs. while loops

Sometimes students rightly ask what's the difference between a for and a while loop. When should I use one construct over the other?

The answer is that there is no clear cut answer! Since both constructs are semantically equivalent the use of one over the other is a matter of programmer taste. However, you may find the following guidelines helpful – particularly the first point.

- As a novice programmer you should not worry too much – use whichever you feel more comfortable with.
- Many programmers prefer to use a for loop when the number of iterations is known in advance i.e. for counter controlled repetition.
- When the number of iterations is not known in advance a while loop is often preferred. This covers situations where you want to keep running a loop until a certain condition is met as well as sentinel controlled repetition. (This includes the common task of reading (and processing) the contents from a file until an end-of-file (EOF) character is read.)
- In general for loops are more suitable for iterating over lists/arrays. We will cover the topic of lists/arrays at a later stage.

For the moment it is important that you can recognise situations where you need to use a loop in your program. The choice of which type of loop to use i.e. for or while, doesn't really matter. Loops that can be coded using a while construct can also be constructed using a for construct and vice versa.

For example, the programs on the left-hand-side and right-hand-side are logically equivalent i.e. they do exactly the same thing.

Table 6.5: For vs While Loop

Example loops using `while`		Example loops using `for`
`count = 0` `while (count < 10):` `  print("Hello World")` `  count = count + 1`		`for counter in range(10):` `  print("Hello World")`
Both programs display Hello World 10 times		
`n = 1` `while (n <= 12):` `  print(n)` `  n = n + 1`		`for n in range(1, 13):` `  print(n)`
Both programs display the numbers 1 to 12 inclusive		
`n = 10` `while (n > 0):` `  print(n)` `  n = n - 1`		`for n in range(10, 0, -1):` `  print(n)`
Both programs display the numbers (backwards) from 10 down to 1 inclusive		

Examples
The purpose of the following examples is to serve to illustrate that anything that can be done with a while loop can also be done using a for loop.

Counting Numbers
Earlier we encountered an example to display all the numbers from 1 up to some number (max) entered by the user. The solution using a while loop is shown below.

```
122.    n = 1
123.    max = input("Enter the upper limit")
124.    max = int(max)
125.    while (n <= max):
126.   print(n)
127.   n = n + 1
```

The same problem can be solved using a `for` loop as follows:

```
128.    max = input("Enter the upper limit")
129.    max = int(max)
130.    for n in range(1, max+1):
131.   print(n)
```

Cubes
The program on the left hand side below generates and displays the first 10 cubes as shown on the right hand side.

Figure 6.17: Creating Cube in loop

```
1 print("n \t n cubed")
2 for n in range(1, 11):
3     print(n, "\t", n**3)
```

Program to display the first 10 cubes

```
n     n cubed
1     1
2     8
3     27
4     64
5     125
6     216
7     343
8     512
9     729
10    1000
```

Program output

Accumulators

Recall the pseudo-code below used as an initial step in the development of our solution to calculate and display the average of 10 student marks.

```
Initialise the total mark to zero
Initialise a counter to 1

Loop 10 times (while the counter is less than or equal to 10)
   Prompt the user to enter a student mark
   Convert the mark from a string to an integer (so that it can be added)
   Add the mark just entered to the total mark
   Add one to the counter

Calculate the average by dividing the total mark by 10
Display the result
```

An implementation using a `for` loop is shown below:

```
total=0
for n in range(1, 11):
    mark=input("Enter student mark: ")
    mark=int(mark)
    total=total+mark

average=total/10
print("The average mark is", average)
```

Contrast this with the while implementation provided earlier.

```
132.     # Program to compute class average
133.     total=0
134.     counter=1
135.     while (counter <= 10):
136.   mark = input("Please enter a student mark")
137.   mark = int(mark)
138.   total = total + mark
139.   counter=counter+1

140.     average = total/10
141.     print("The average mark is", average)
```

As a final point it is worth noting that for loops do not lend themselves very well to situations where there are an unknown number of iterations. (In other words for loops lend themselves to scenarios where the number of iterations is known in advance of runtime.) For this reason, it is rare to use sentinels with for loops.

Exercises 8 (for loops)

1. Study the code snippets below and once you have figured out what they do re-write each snippet using a for loop.

	`` min = input("Enter the lower limit") min = int(min) while (min <= 100): print(min) min = min + 1 ```
	`` min = input("Enter the lower limit") min = int(min) max = input("Enter the upper limit") max = int(max) while (min <= max): print(min) min = min + 1 ```
	`` n = 1 while (n <= 10): print(n+1) n = n + 1 ```
	`` n = 0 while (n <= 100): print(n) n = n + 5 ```
	`` def isEven(number): return (number%2 == 0) for n in range(100): if isEven(n): print(n) ```

2. The built in function pow(x,y) returns x raised to the power of y. Use a for loop to implement your own power(x,y) function.
3. Write a program that uses a for loop and the power function you just implemented in the previous exercise to compute the sum $1^2 + 2^2 + 3^2 + 4^2 + \ldots + ^2$.
4. Look back over the exercises you completed earlier – in particular, look at your solutions to exercises 2, 3, 4 and 7. Now attempt to provide an implementation using `for` loops.

break and continue
These two keywords – explained briefly below – are associated with both for and while loops.

A break statement causes the flow of control to be transferred to the statement immediately following the loop i.e. exits the loop

A continue statement causes the loop to skip the remainder of its body and transfer control back to the loop guard

```
142.     # Break Example
143.     var = 10
144.     while var > 0:
145.   print('Current variable value :', var)
146.   var = var -1
147.   if var == 5:
148.     break
8.
9.  print("Good bye!")
```

This will generate the following output:

```
Current variable value : 10
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Good bye!
```

```
149.     # Continue Example
150.     var = 10
151.     while var > 0:
152.   var = var -1
153.   if var == 5:
154.     continue
155.   print('Current variable value :', var)
156.     print("Good bye!")
```

This will generate the following output:

```
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Current variable value : 4
Current variable value : 3
Current variable value : 2
Current variable value : 1
Current variable value : 0
Good bye!
```

Nested loops
A nested loop is a loop within a loop.

Example – 24 hour clock
To help us gain an understanding of nested loops it might be useful to consider every possible time - to the nearest minute - in a 24 hour clock. There are 24 hours and each hour has 60 minutes.

The first element in our list would be 00:00, the second element would be 00:01, followed by 00:02 and so on until we reach 00:59. The next elements would be 01:00, 01:01, 01:02, 01:03 etc. The final element in the list would be 24:59.

In effect we are moving through the hours in sequence and for each hour we move through the minutes. Conceptually, this is a loop within a loop.

The following code lists all the possible times – minute by minute - of the 24-hour clock as discussed earlier.

```
for hour in range(0, 25):
    for minute in range(0, 61):
        print(hour, ":", minute)
    print("")
```

In this example the outer loop iterates over the variable `hour` and the inner loop iterates over the variable `minute`. The outer loop changes only after the inner loop has made 60 iterations. The inner loop causes the value hour:minutes to be printed vertically. The final line (i.e. `print("")`) is only executed after the inner loop has completed. This results in the values for each hour being separated by a blank line.

Care should be taken when using nested loops as they can impact on program runtime performance. The above simple program makes $24 \times 60 = 1440$ iterations!

Example – horizontal printing
The program on the left hand side uses a nested loop to generate the output displayed on the right.

```
for i in range(5):
    output=""
    for j in range(1, 10):
        jStr=str(j)
        output=output+jStr
    print(output)
```

```
123456789
123456789
123456789
123456789
123456789
```

Note the use of the built in function `str` to convert an integer to a string. In this case the

integer to convert is contained in the variable `j`. The string equivalent of the integer is stored

in the variable `jStr`. The string to display (i.e. `output`) is constructed by concatenating each successive value of `jStr`.

Example – nested loop to print multiplication tables
Earlier on we saw a program to display the 7 times tables and its output.

Figure 6.18: Multiples of 7

```
n=1
while (n<=12):
    print("7 x",n,"=",7*n)
    n=n+1
```

Program listing: 7 times tables

```
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63
7 x 10 = 70
7 x 11 = 77
7 x 12 = 84
```

Output

The program is fine in the sense that it illustrates how to use a variable (i.e. `n`) inside a `while` loop. The same functionality could easily have been achieved using a `for` loop as follows.

Figure 6.19: 7 times tables

```
for n in range(1, 13):
    print("7 x", n, "=", 7*n)
```

7 times tables – alternative implementation

Regardless of which type of loop was used the key point is that the loop is useful because each output line can generated by using the same statement (i.e. 7 multiplied by the iterating variable, `n`).

What if we wanted to print out all the multiplication tables from 1 to 12? In order to solve this problem we need to use nested loops.

Our solution will make use of the fact that we already have the code to generate and display the 7 times tables. We will take our '7 times' code and generalise it to work for every value between 1 and 12. To do this we will introduce a new variable called `tables` and another loop to iterate over `tables`. Two solutions - using `while` and `for` loops – are presented on the next page.

Study both solutions carefully (key them in and run them) before reading the commentary which follows.

Figure 6.20: Key them in and run them

```
2 for tables in range(1, 13):
3     for n in range(1, 13):
4         print(tables, "x", n, "=", tables*n)
5     print("")
```

Solution 1: Print times tables from 1 to 12 inclusive

Figure 6.21: Print tables from 1 to 12

```
2 tables = 1
3 while (tables < 13):
4     n = 1
5     while (n < 13):
6         print(tables, "x", n, "=", tables*n)
7         n=n+1
8     tables=tables+1
9     print("")
```

Solution 2: Print times tables from 1 to 12 inclusive

The first thing to notice is that although the first solution is terser (i.e. shorter) both solutions do exactly the same thing. The main reason that the solution with the for loop is terser is because it does not have to increment the loop variables i.e. lines 7 and 8 in the second solution are not needed in the first.

However, the key point is the use of nested loops in both solutions. The indentation of code is critical to the understanding of the use of both loops. In the case of this example there are two loops – the first loop is referred to as the *outer loop* and the second loop is called the *inner loop*. Each loop is marked by a new level of indentation. In the solutions above we see two levels of indentation – each level corresponds to a particular loop.

Iteration around the outer loop always occurs at a slower pace than iteration around the inner loop. For every step in the outer loop there are 12 iterations of the inner loop. The outer loop is controlled by the variable tables and the inner loop is controlled by the variable n. In effect what is happening is that for every iteration of the variable tables there are 12 iterations of the variable n. Thus, the 1 times tables are displayed (i.e. 1 × 1 = 1, 1 × 2 = 2, …. 1 × 3 = 3), followed by the 2 times tables (i.e. 2 × 1 = 2, 2 × 2 = 4, …. 2 × 3 = 6)
and so on right up to the 12 times tables (i.e. 12 × 1 = 12, 12 × 2 = 24, …. 12 × 3 = 36).

Example – prime numbers

The listing below – used to generate and display all the prime numbers from 2 up to 20 – demonstrates a good example of how while loops can be nested.

```
i = 2
while(i < 20):
   j = 2
   while (j <= (i/j)):
      if not(i%j):
         break
      j = j + 1
   if (j > i/j):
      print(i, " is prime")
   i = i + 1
```

The program generates the following output:

```
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
```

Use the table below to trace through the program variables and deepen you understanding of how the program works.

i	j	i/j	I&j

Final notes on nested loops
In theory loops can be nested to any depth but in practice it is quite uncommon for programmers to need to write loops with more than single level of nesting i.e. a loop within a loop.

Nested loops are most commonly used for iterating over two dimensional (2D) arrays (i.e. the inner loop does the horizontal processing and the outer or slower loop does the vertical

processing). 2D arrays in turn are used in the implementation of many board games. Arrays and lists will be discussed in more detail at a later stage.

Exercises 9 – nested loops

1. Write a program to print out all the addition tables 1 through to 12.

2. Write a program to compute and tabulate the factorial of all numbers from 0 to 10? Your output should look like ….

n	$n!$
0	1
1	1
2	2
3	6
4	24
5	120
6	720
7	5,040
8	40,320
9	362,880
10	3,628,800

3. Write a program that asks for a number (call it) and, based on the number entered display a triangle such as the one depicted as an example with = 5.

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1
```

4. Write a program that uses a loop – and 4 conditionals – to print out the following:

```
12 midnight
1am
2am
...
12 noon
1pm
...
11pm
```

1. Lotto Quick Pick. Write a program to generate and display 6 random numbers between 1 and 45

2. Unlucky 13. Write a program to generate and display 6 random numbers between 1 and If any of the numbers generated are 13 stop the program with a message “Process terminated – unlucky 13!”.

3. Five Quick Picks. Write a program to generate and display 5 rows of 6 numbers between 1 and 45.

4. Write a program that simulates the rolling of a dice (Hint: use a random number between and 6.) The program should continue until a six is ‘rolled’. The program should display the number of rolls it took to reach 6.

5. Extend the above program to roll a dice 1000 times. Display the average score. Display the number of times each number came up (i.e. number of ones, twos, threes, fours, fives and sixes.)

6. Extend the earlier program to count the number of throws it takes to get ‘snake eyes’ Implement with one dice first i.e. a single roll at a time. Then two. (Snake eyes mean two ones in a row with one dice or double ones if using two die.)

7. A Fibonacci sequence is a sequence of numbers where each successive number is the sum of the previous two. Thus, the first 7 numbers in the Fibonacci sequence are 1, 1, 2, 3, 5, 8, 13. Write a program to that computes the nth Fibonacci number. For example, if the user entered 6 the program would output 8 (as 8 is the 6th Fibonacci number)

8. Study the program listing shown on the left hand side below and see if you can figure out how it generates the output displayed on the right hand.

```
1 number = 82137
2 while (number > 0):
3     digit = number % 10
4     print(digit)
5     number = number//10
```

```
7
3
1
2
8
```

Program Listing *Output*

a) What does line 3 do in the above listing?
b) Explain how the loop works? What causes it to terminate?
c) What is the main difference between the code shown below and the listing above?

```
1 def displayDigits(number):
2     while (number > 0):
3         digit = number % 10
4         print(digit)
5         number = number//10
```

d) What output would be generated by each of the following function calls to the above function:
displayDigits(87)
displayDigits(261297)
displayDigits(5)
displayDigits(999)

e) Use the displayDigits() function as a template for a function to count the number of digits in a number. Call your function countDigits(). Test your function countDigits() by making the following calls:
countDigits(87)
countDigits(261297)
countDigits(5)
countDigits(999)

f) Use your countDigits() function as a template to write another function which coutns the number of times a specific digit occurs within a number. For example if the number was '999' and the specific digit was '9' the function would return 3.

9. Study the code on the left hand side below carefully.

```
def mystery(number):
    i=1
    x=number
    while (number > 0):
        y = x % 10**i
        print(y)
        i=i+1
        number = number//10


mystery(81237)
```

number	I	x	y

Now trace the value of each variable used by completing the table provided on the right hand side.

10. Write a program to accept 3 numbers(i.e. DD MM YY) and determine whether the numbers entered constitute a valid date. If the date entered is invalid print an appropriate error message to the user. (Hint: You need to answer the question – what is a valid date? Ignore the possibility of leap years.)

11. The CAO awards points to students based on their achievements in the Leaving Certificate examination. A student's points are calculated according to the tables displayed below, counting their best six subjects only.

Higher Level Grade	Points	Ordinary Level Grade	Points
H1	100		
H2	88		
H3	77		
H4	66		
H5	56	O1	56
H6	46	O2	46
H7	37	O3	37
H8	0	O4	28
		O5	20
		O6	12
		O7	0
		O8	0

Write a program that asks a user to enter 6 results. For each result enter a code ('H' to indicate the result is higher level; 'O' for ordinary level) followed by the actual percentage. The program should then determine the relevant points for the percentage entered and keep a running total of the points to date. Once the last result has been entered the program should display the points total accumulated.

Can you think of any possible ways the program could be made more realistic? What if the user didn't enter the best 6 results? What about data validation? What about additional points for Higher level Mathematics? How many times in one run could a user enter a point's value greater than 100? How might you incorporate an averaging functionality?

Lesson 7

String Processing

INTRODUCTION

A *string literal* is a sequence of characters enclosed in quotation marks (either single or double). Sometimes a string literal is referred to as *an array of characters*.

Recall our very first Python program.

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

In this program the string *Hello World* is a string. It is passed as an argument into the built in Python function called print. The function print causes the string to be displayed the string on the output console.

It is useful to think of strings as a type of data for our programs to process. Python, like most programming languages supports two basic types of data – strings and numbers. Numeric data can be either whole numbers (i.e. integers) or decimal (i.e. float).

Variables can be used to reference string literals (in a similar manner to the way we have already been using variables to store numeric data). For example, the following code snippet would have the same effect as the above:

```
message = "Hello World"
print(message)
```

In the above program the string *Hello World* is assigned to the variable called message. Any subsequent reference to this variable is a reference to the string. Therefore, when message is passed as an argument to the print function, the effect is to display the text *Hello World* as output.

Key Point: Strings can be referenced directly using string literals or indirectly by using variables.

It is important to understand strings for the simple reason that they are one of the most common types of data processed by computer programs and systems. Consider any system that prompts a user to enter non-numeric data such as name or address. The values entered are more than likely stored by the program as strings. Even data such as phone number which you might think is numeric are often stored as strings – this is because a phone number can contain non-numeric characters such as open bracket, '(', close bracket, ')', dashes, '-', or even the plus symbol, '+'.

As is the case with other datatypes such as 'int' and 'float' there are certain operations

associated with strings. The main operations are addition, multiplication, indexing and slicing. These are now explained in turn.

LEARNING OBJECTIVES

1. The use of variables to store strings
2. How to concatenate two strings together
3. Accessing the characters of a string
    - Indexing
    - Slicing
4. String Processing(using loops to traverse a string)
    - Counting letters, vowels
    - Building up an output string

LECTURE DISCUSSION

String addition and multiplication

Strings, just like numbers can be added and multiplied. (Since the operation of string multiplication is not widely used by other programming languages it is considered to be relatively unimportant and therefore will not be given any further consideration here.)

The operation of adding one string to another is commonly referred to as *string concatenation*. We say one string is concatenated to another string to form a new (and longer) string. The plus operator, '+' is used to concatenate two strings.

Both programs below illustrate how the plus operator is used to concatenate strings.

```
print("Cavan "+"Institute")
```

```
str1 = "Cavan "
str2 = "Institute"
print(str1+str2)
```

The program on the left concatenates two string literals while the program on the right hand side concatenates the two string referenced by the variables `str1` and `str2`.

This technique of concatenation may be used in a program to build up (i.e. construct) a meaningful string we want our program to display. For example consider a system that displays a personalised welcome message to a user every time they log on. The initial string may be a string literal such as *Welcome* and the string to concatenate may be the user's name referenced by a variable.

Assuming that the name of the user is stored in a variable called `name`, the following program snippet will display such a welcome message.

```
name="Joe"
...
message = "Welcome"
print(message+" "+name)
```

The second line in the above snippet concatenates a space followed by the contents of the variable `name` to the initial string referenced by `message`. Let's say the string referenced by `name` is *Joe*. The effect would be to display the message *Welcome Joe*.

Notice that the program snippet below does exactly the same thing.

```
name="Joe"
...
message = "Welcome "
print(message+name)
```

The difference between the two programs is very subtle but important to our understanding of strings. In the second snippet the space character which separates the two strings is contained (as the last character) in the string referenced by `message`. Therefore, it is not necessary to concatenate the space character in the argument to `print`.

One of the key skills to the art of computer programming is a keen eye for detail. It is therefore essential that programmers are completely aware and in control of every single character that makes up a string. A characters can be thought of as any single symbol that than be typed in from the keyboard.

The *alphabetic characters* consist of the letters from the alphabet i.e. 'A' to 'Z' and 'a' to 'z'. (This is the case for the Roman alphabet – of course there are other alphabets too which we will not go into here.) It is important to note that computer programs treat lowercase and uppercase characters differently i.e. internally the character 'A' is stored different to the character 'a'. *Numeric characters* consist of the digits from '0' through to '9'. *Whitespace characters* refer to characters that cannot be seen when printed. Examples include space, tab and newline. An *alpha-numeric string* is one that consists of alphabetic or numeric characters only.

The following program demonstrates the addition of two string variables.

Figure 7.1: Adding 2 string variables

```
1
2 #Adding more than 2 strings in a single expression
3 string1 = "Cavan"
4 string2 = "Institute"
5 string3 = string1+" "+string2
6 print(string3)
7
```

The two strings referenced by variables `string1` and `string2` are concatenated together in the same expression (line 3). Notice how the space is also included. The resulting string is assigned to the variable `string3`. When the program is run it displays the string *Cavan Institute*.

It is also perfectly legal (i.e. syntactically correct) to add/concatenate any mixture of string literals and string variables. For example the following code snippets are valid.

Figure 7.2: Adding more than 2 string variables

```
2 #Adding more than 2 strings in a single expression
3 string1 = "Cavan"
4 string2 = "Institute"
5 string3="Welcome to "+string1+" "+string2+"!"
6 print(string3)
7
```

Exercises 1 (Introduction)

1. State whether a string would be appropriate for the following types of data:
    - Country of Origin
    - Address
    - Product Name
    - Product Description
    - PPSN
    - Invoice Number

2. Write a one line program that displays the string *Hello Python users!*

3. Change the program so that the string *Hello Python users!* is assigned to a variable before displayed.

4. Write a program that stores your first name in one variable (e.g. `firstName`) and your surname in another variable (e.g. `lastName`) and finally prints out the result of concatenating the two variables. (For example, if your first name is *Joe* and your surname is *Blogs* the program will output *JoeBlogs*.)

5. Modify the above program so that it concatenates a space at the end of the first name before concatenating the surname. (This time the output will be something like *Joe Blogs*)

6. Write a program that prompts a user to input their first name(e.g. *Joe*) followed by their surname (e.g. *Blogs*) and then print a message along the lines:
   Hello Joe Blogs, Welcome to my crazy world!

7. Write a program that asks a user to enter a noun(e.g. apple) and then a number. If the number is 1 the program should output the message – *The singular apple is apple.* If the number is greater than one the program output message should be – *The plural of apple is apples.*

8. Read the short program bellow (type in if you want) and then answer the questions shown.

```
1. s1 = "Testing, one, two"
2. s2 = s1+" three"
3. s3 = "three"
4.
5. print(s1)
6. print(s2)
7. print("s2")
8. print(s1+s3)
9. print(s1+" "+s3)
10 print(s1-s3)
11 print(s4)
12 print(s1+" "+s3+" "+"four")
13 print(s1+" "+s3+" "+four)
```

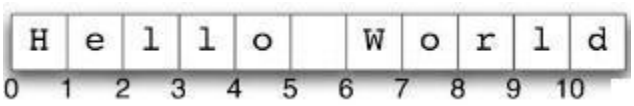
(i) How many variables are there? What are their names?
(ii) What does line 2 do?
(iii) Identify the other lines in the program where string concatenation occurs.
(iv) What is the difference between lines 6 and 7?
(v) What is the difference between lines 8 and 9?
(vi) What output does the program display?
(vii) Identify the syntax errors (there are at least 3!)

String Indexing
Indexing is by far the most important and one of the most common operations carried out on strings. An index is simply a numeric position of a character in a string.

Consider the one line program shown below (the program does not display any output).

```
message = "Hello World"
```

The program simply initialises a variable called `message` with the string *Hello World*. The variable `message` is actually a *reference* to the string i.e. rather than containing the string, it contains the address of the memory location where the string is stored. We shall see later on that this concept of referencing is important but for the moment let us concentrate on the string itself. In particular, we will look at the index or position of each character that makes up the string. This is illustrate in the diagram below where the index of each character is shown below the character.



As can be seen the first character i.e. 'H' has an index of zero. This is important. The first character in a string always occurs at index position zero (this is often called the *zeroth* position), the second character at index position 1, the third character at index position 2 and so on. The string *Hello World* contains 11 characters and each character has a unique index ranging from 0 to 10.

In general, when there are *n* characters in a string the last character always occurs at index position *n-1*. Therefore, if there are 5 characters in a string the last character occurs at index position 4.

Each individual character of a string can be accessed through the process of indexing. In Python (and most programming languages) square brackets are used to denote the index operation. To access a particular character in a string the character's index is enclosed in square brackets. For example, the first element of the above string can be accessed using message[0]. Similarly,

message[0] → H
message[1] → e
message[2] → l
message[3] → l
message[4] → o
message[5] → ' '
message[6] → W
message[7] → o
message[8] → r
message[9] → l
message[10] → d

The following code snippet demonstrates the use of square brackets to access the individual characters in a string.

Figure 7.3: String indexing

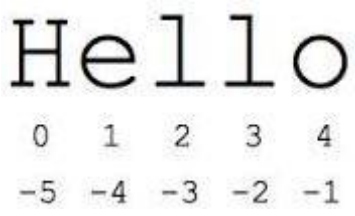
```
1
2 #String indexing with square brackets
3 string = "Cavan"
4
5 c1=string[0]
6 print(c1)
7
8 c2=string[1]
9 print(c2)
10
11 print(string[2])
12 print(string[3])
13 print(string[4])
```

Line 5 assigns the character stored at index position 0 i.e. 'C' to the variable `c1`. Line 9 assigns the character stored at index position 1 i.e. 'a' to the variable `c2`. The program displays the following output:

```
C
a
v
a
n
```

As a final point it is worth noting that unlike most other programming languages, Python permits the use of negative numbers as indices.

The diagram to the right depicts the positive and negative indices of the string *Hello*.



As can be seen the last (rightmost) character of the string can be accessed using index -1. Working from right to left the index of each preceding character is reduced by 1.

The following code snippet demonstrates the use of negative indexing

Figure 7.4: Indexed using negative integers

```
#Strings can be indexed using negative integers
string = "Cavan"

print(string[-1])
print(string[-2])
print(string[-3])
print(string[-4])
print(string[-5])
```

Note the use of negative indexing is not encouraged as it is not supported in most other programming languages. As an exercise try adding the line 'print(string[-6])' to the above. What happens?

Common Index Errors

If you attempt to access a character in a string using an index that is too big (or too small) for that string, Python returns a runtime error telling you that the index is out of range. For example running the following code snippet would result in a runtime error being displayed:

Figure 7.5: Indexing out of range

```
2 #String indexing with an out of range index
3 string = "Cavan"
4
5 print(string[5])
6
```

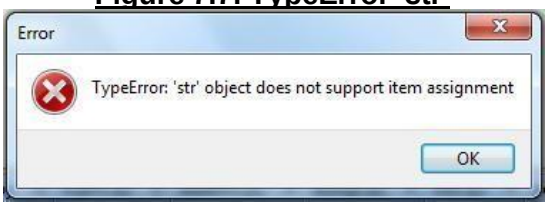
Strings are *immutable* meaning that they cannot be changed. For this reason it is not permitted to use the index operator ([]) on the left hand side of an assignment expression. For example, the string *Cavan* could not be changed to *Navan* by using the following:

Figure 7.6: Immutable strings

```
2 #Strings cannot be changed (i.e. they are immutable)
3 string = "Cavan"
4
5 string[0]="N"
6
7 print(string)
```

The following runtime error is displayed.

Figure 7.7: TypeError 'str'



Exercises 2 (Indexing)

1. The string Python is made up of 6 characters. Now answer the following questions:
   i. Which character occurs at the last position
   ii. Which character occurs at the zeroth position
   iii. What character occurs at index position

2. For each of the strings listed below answer the questions which follow.
   - *No. 11 The Laurels, Dublin 4*
   - *The Godfather, Part 2*
   - *St. Patricks Day, March 17*

   i. How many characters are in each string
   ii. Which word describes the type of string best – alphabetic, numeric or alpha-numeric.
   iii. What is the index position of e very digit(i.e. numeric character)?
   iv. What is the index position of every capital letter?
   v. What is the index position of every space character?
   vi. What is the index position of every vowel?

3. What output does the following program display?

```
1. town = "Cavan"

2. print(town[0])
3. print(town[1])
4. print(town[2])
5. print(town[3])
6. print(town[4])
```

4. What error would be generated if the following line was added to the above program?

```
print(town[5])
```

5. What output would result from each call to the print function in following program snippet?

```
str1 = "Hello"
str2 = "world"
str3 = str1+" "+str2+"!"
pos = 6

print(str1[1])
print(str2[4])
print(str3[0])
print(str3[4])
print(str3[5])
print(str3[pos]) # Note the use of a variable as the index
print(str3[10])

ch1 = str1[0]
ch2 = str2[1]
ch3 = str3[11]
print(ch1+ch2+ch3)
```

Slicing (substrings)

Thus far we have used indexing to return individual characters of a string. (It is interesting to note that these characters are actually strings of length 1.) We can also use indexing to extract a sub-string from a string. In Python this is referred to as *slicing*. Try the following:

Figure 7.8: Accessing substrings

```
2  # Accessing substrings (slicing)
3  string = "A town like Cavan"
4
5  print(string[0:1])
6  print(string[0:6])
7  print(string[2:5])
8  print(string[2:6])
9  print(string[12:])
10 print(string[2:])
11 print(string[:6])
```

Can you figure it out?
The program displays the following output:

```
A
A town
tow
town
Cavan
town like Cavan
A town
```

Slicing is a useful way to extract part of a string (i.e. a substring) from a larger string. The colon is delimited by the start and end position of the substring we are interested in extracting. The resulting substring (or slice) runs from the starting index up to but not including the end. If the start is missing it us assumed to be zero i.e. the first position of the string. If end is missing it is assumed to be the last position of the string. What do you think the statement 'print(string[:])' would output?

String Length

The length of any string is the number of characters in that string. A string length can be obtained by using the `len` built in function. The use of the `len` function is illustrated in the following example.

Figure 7.9: len() function

```
2  # Use of the built in function 'len'
3  str="The quick brown fox jumps over the lazy dog"
4
5  length=len(str)
6  print("There are ", length, "characters in", str)
7
8  print("The first one is", str[0])
9  print("The last one is", str[length-1])
10
```

Since `len` return the actual number of characters in a string we say it is one based. Recall that indices are zero based. Care must therefore be taken when using the result returned by `len` as an index. The following line of coded (which attempts to access the last character of the `str`) causes an 'index out of range error' because the index given by `len(str)` is one greater than the required index. Line 9 of the previous program shows the correct way to access the last character of a string using the `len` function.

Figure 7.10: len(str)

```
2 str="The quick brown fox jumps over the lazy dog"
3 lastChar=str[len(str)]
```

Exercises 3 (String slicing)

1. What output would the following program display?

```
uprCaseLetters = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
lwrCaseLetters = "abcdefghijklmnopqrstuvwxyz"
letters = uprCaseLetters+lwrCaseLetters

print(uprCaseLetters[0:26])
print(uprCaseLetters[0:5])
print(uprCaseLetters[1:5])
print(uprCaseLetters[:5])
print(uprCaseLetters[20:25])
print(uprCaseLetters[20:])
print(letters[0:26])
print(letters[26:52])
```

2. Write a small program to extract the following strings from the string:
   The quick brown fox jumps over the lazy dog
   a) "quick"
   b) "fox"
   c) "The"
   d) "The quick brown fox"
   e) "jumps over the lazy dog"

3. Write a program that asks a user to input their first name (e.g. *Joe*) followed by their surname (e.g. *Blogs*) and then outputs the initial of the forename followed by the surname (e.g. *JBlogs*).

4. Write a program that asks a user to input their first name (e.g. *Javier*) followed by their surname (e.g. *Mascherano*) and then outputs the initial of the forename followed by the first seven characters of the surname (e.g. *JMascher*). Test your program using *Joe Blogs* as the name.

5. Put a loop in the program from the previous exercise t okeep going until the user enters "quit" as the first name (i.e. use quit as the sentinel to end the loop).

6. Write a program that accepts a date in the format DDMMYY (e.g. 010114) and outputs three lines e.g. *Day is 01, Month is 01, Year is 2014.*

7. Modify the last program to remove any leading zeros from the day and month (if they occur). For an input of **010114** the output would be *Day is 1, Month is 1, Year is 2014*. For an input of **120114** the output would be *Day is 12, Month is 1, Year is 2014.*

String Comparisons
As was the case with numbers, the relational operators >, >=, <, <=, == and != can be used with strings. Recall that the relational operations return either True or False. The following table illustrates the results of some string comparisons.

Table 7.1: Relational operations comparison

Test (comparison)	Result	Comment
"apple" < "banana"	`True`	The letter 'a' comes before 'b'. The string "apple" is therefore less than the string "banana" and the test returns `True`.
"apple" > "banana"	`False`	The string "apple" is not greater than the string "banana"
"apple" > "aardvark"	`True`	Since the first letter in both strings is the same (i.e. 'a') the test continues from the second letter
"Apple" == "apple"	`False`	The two strings are *not* the same. IMPORTANT: Python (like all programming languages) is case sensitive.
"Apple" != "apple"	`True`	Since the strings are *not* the same the test for inequality returns `True`
"Apple" < "apple"	`True`	All upper case letters are less than their lower case equivalents.
"Zebra" < "giraffe"	`True`	All upper case letters are less than lower case letters.

You should check each of the above out by passing the test string as the argument into the print function. For example, use *print("apple" < "banana")* to verify the first test.

The comparison operators in and not in check whether a value occurs (does not occur) in a string (or indeed any other ordered list). Like the relational operators, in and not in return either True or False depending on whether the item being checked for is a member of the list on the right hand side. The use of the in operator is illustrated in the table below.

Table 7.2: Test Comparison (in and not in)

Test (comparison)	Result	Comment
"apple"in "banana"	`False`	The string "apple" is not contained in the string "banana"
"ana"in "banana"	`True`	The string "ana" is a member of the string "banana"
"o"in "aeiou"	`True`	The string "o" is a vowel
"e"not in "aeiou"	`False`	The string "e" is a vowel

Recall the use of the `in` operator as part of the for loop.

Test your understanding - String Comparison
For each of the following tests state whether the result would be True or False

a) "apple" < "orange"
b) "apple" > "orange"
c) " aardvark" > "Aardvark"
d) "Apple" in "apple"
e) "Zebra" > "elephant"

String Processing

We are now ready to write programs that process strings in a more meaningful manner. Computers spend a significant portion of time processing strings. Strings are typically processed by writing programs to traverse or cross the string and examine it one character at a time.

The applications of string processing are wide and varied. One simple application of string processing might be to analyse large volumes of text. For example, a user might be interested in knowing the number of sentences or words a piece of text contains. Perhaps a user might be interested in searching for and/or counting the number of occurrences of a certain character (e.g. a vowel) or sequence of characters (e.g. a word) in a piece of text. Another application of string processing is in the generation of ciphers that could be used to encrypt or decrypt data.

The technique for string processing relies on the use of loops. The following program demonstrates the use of a `for` loop to count the number of characters in a string.

Figure 7.11: Traversing a string

```
2 # Traversing a string - counting the number of characters
3 str="The quick brown fox jumps over the lazy dog"
4 count=0
5 for character in str:
6     count=count+1
7
8 print(count)
```

Lines 3 and 4 of the above program initialise the variables `str` and `count`. The purpose of the program is to count the number of characters in the variable referenced by `str`. The variable `count` is just used to store the number of characters processed by the loop on line 5. Line 5 is the key line in this program. This line iterates through every character in the string, `str`. Line 6 – the only line in the loop body - increments the value of `count` on each iteration. By the time the loop has finished execution, the value of `count` will be the number of characters in the string (i.e. the length of the string). Line 8 simply displays the value of `count` before the program terminates.

In the interest of good design practice it is worth pointing out how the essence of the above code could be captured using a function – we will call it length. It is important to study the listing below and understand how it works and why it is more useful than the above code.

```
  Function to return the length of a string, s
def length(s):
    count = 0
      for character in s: count
                    = count+1

    return count

string = "The quick brown fox jumps over the lazy dog"
l = length(string)
print(l)
```

Can you figure out what the following program does?

Figure 7.12: Traversing a string (for loop)

```
2 # Traversing a string - using a for loop
3 str="The quick brown fox jumps over the lazy dog"
4
5 for character in str:
6     print(character)
7
```

The program is very similar to the one shown on the previous page. This time instead of counting the characters in the string the program just displays each individual character on the screen. The following code snippet has the same effect – this time using a for loop.

Figure 7.13: Index and Length

```
2 # Traversing a string
3 str="The quick brown fox jumps over the lazy dog"
4
5 length=len(str)
6 index=0
7
8 while index < length:
9     print(str[index])
10    index=index+1
```

Note the use of the variables `index` and `length` in the above program – the former is used as the index into the current character in string, `str` and the latter is used to store the length of the string. Both variables are used to form the loop guard shown on line 8. The while loop keeps going as long as the value of `index` is less than the value of `length`. Line 9 causes the character at the current index position in `str` to be displayed. Line 10 increments the `index` on each iteration of the loop.

Counting Letters in a string
The following code counts the number of times the letter ‘a’ occurs in the string ‘banana’:

Figure 7.14: Modify with traversing

```
2 # Traversing a string - counting the number of characters
3 fruit = "banana"
4 count = 0
5 for char in fruit:
6     if char == "a":
7         count += 1 # this is the same as count=count+1
8 print(count)
9
```

Challenge. Modify the above code so that it uses a while loop instead of the for loop.

Counting Vowels in a string
The following code prints out the number of vowels in a string entered the user:

```
1. string = input("Please enter a string")
2. vowels = "aeiou"
3. count=0
4. for ch in string:
5. if ch in vowels:
6. count = count +1

7. print(count)
```

Lines 4 and 5 are key. Line 4 sets up the loop to iterate through the string which was entered by the user when line 1 was executed. The variable `ch` is used to store a reference to each character in string as they are being iterated through. Line 5 asks the question is the current character a vowel? If the answer is true line 6 is executed and the variable `count` is incremented. In any case, execution continues at line 4 until all the characters in the string have been processed. When the loop ends the value in `count` is displayed and the program terminates.

The same functionality can be achieved using a while loop as follows:

```
string = input("Please enter a string")
vowels = "aeiou"
count=0
index=0
length=len(string)

while index < length:
    ch = string[index]
    if ch in vowels:
        count = count +1
    index = index+1
print(count)
```

You should study the above code carefully paying particular attention to the use of variable `index`.

Challenge. Modify the above code so that it is encapsulated in a function. The name of the function should be `countVowels`. The function should accept one parameter – the string – and return the number of vowels in the string. A sample call to the function would be:

```
print(countVowels("Hi there. Nice day"))
```

This will result in the number 6 displayed.

Building up an output string
It is quite common to need to build up a string (sometimes called an output string) before finally displaying it. Let us say we wanted to display a sentence on a single line of output. The sentence is to be constructed from a number of words entered separately by the user (until the user enters the word *stop*). The code below represents an initial attempt.

```
1. word = input("Please enter a word (stop to end)")
2. while (word != "stop"):
3. print(word)
4. word = input("Please enter a word (stop to end)")
```

The problem with the above program is that each word is displayed on a new line straight after it is entered by the user. Our requirement is that all the words should be displayed on a single line of output. A better solution is presented below:

```
1. sentence=""
2. word = input("Please enter a word (stop to end)")
3. while (word != "stop"):
4. sentence = sentence + word + " "
5. word = input("Please enter a word (stop to end)")

6. print(sentence)
```

Notice the use of the variable `sentence`. The above program works by building up the string referenced by this variable – from line 1 where `sentence` is initialised to an empty string right through to line 7 which displays the final sentence. The loop works in the same way as it did for the previous program i.e. it keeps executing until the word entered by the user is *stop*. Line 4 is the key line. This line works by concatenating the word just entered to the sentence accumulated so far. The resulting reference is assigned to the variable sentence. Notice also how the space to separate the individual words is tagged on to the end each time.

The table below illustrates how the sentence is built up for a sample run of the program in which the user enters the following four words individually – The, quick, brown, fox.

Table 7.3: The quick brown fox…

word	sentence	Comment
	`""`	Initially blank (empty string)
`"The"`	`"The"`	User enters the word `"The"`
`"quick"`	`"The quick"`	User enters the word `"quick"` which is concatenated to the sentence
`"brown"`	`"The quick brown"`	The word `"brown"` is concatenated to the sentence
`"fox"`	`"The quick brown fox"`	The word `"fox"` is concatenated
`"stop"`	`"The quick brown fox"`	User enters `"stop"`

We now turn our attention to the problem of extracting all the vowels from a string and displaying the resulting string on a single line. For example, if the initial string was *`The quick brown fox jumps over the lazy dog`* the required output string would be `Th qck brwn fx jmps vr th lzy dg`.

The following initial attempt displays all the characters of `str` excluding vowels:

```
1. string = "The quick brown fox jumps over the lazy dog"
2. vowels = "aeiou"
3. for character in string:
4. if character not in vowels:
5. print(character)
```

An alternative implementation using a `while` loop is shown below.

```
1. string = "The quick brown fox jumps over the lazy dog"
2. vowels = "aeiou"
3. index = 0
4. length = len(string)
5. while index < length:
6. character = string[index]
7. if character not in vowels:
8. print(character)
9. index = index + 1
```

Both implementations have the same problem i.e. all the non-vowel characters are printed as they are found. This means that they are separated by a newline. In order to have the resulting string printed on a single line it is necessary to construct a new string by add the non-vowel characters to it as they are found. This is done as follows:

```
1. string = "The quick brown fox jumps over the lazy dog"
2. newStr = ""
3. vowels = "aeiou"
4. for character in string:
5. if character not in vowels:
6. newStr = newStr + character
7.
8. print(newStr)
```

Notice how in the above example, the variable newStr is built up and added to every time a non-vowel character is encountered.

Challenge. Modify the above code so that it uses a while loop instead of the for loop.

Reversing the characters in a string
Consider the problem of reversing the characters in a string. For example the string “Python” reversed is “nohtyP”. Can we design an algorithm to solve this problem? (Recall that an algorithm is a sequence of steps which if followed will always lead to the desired result.) We begin our design by analysing, very carefully the process for reversing the string. The string to be reversed must first be initialised.

In essence, the reversed string is produced by first taking the last character of the input string, and then the second last and so on until we reach the first character. In other words we traverse the input string, starting from the last character and ending at the first. As we encounter each character along our traversal we add (i.e. concatenate) it to some new output string. By the end of the traversal this output string will have become the desired reversed string. We can now develop our solution using the following pseudo-code:

Initialise the input string to be reversed
Initialise the reversed string to blank
Set index to be last character of the input string
While the index is not at the start of the input string
    Add the character at the index position of the input string to the output string
    Move the index back by 1
Print the reversed string

We are now ready to implement our solution. Try running the following code to verify that it works.

Figure 7.15: String revearsal

```
1 # A program to reverse the characters in a string
2 inputString="The quick brown fox jumps over the lazy dog"
3 reversedString=""
4 index=len(inputString)
5 while index>0:
6     reversedString=reversedString+inputString[index-1]
7     index=index-1
8
9 print(reversedString)
```

Although the above short program does what we initially set out to do i.e. it reverses a string, its use is *very* limited because it only works once. Furthermore, the only string it ever reverses is "The quick brown fox jumps over the lazy dog". It would be much better to have a piece of code that could reverse any string and which could be used as often as we wanted without having to duplicate the above lines. This can be accomplished by *encapsulating* the functionality of the above in a single function (we will call `reverseString`). The string to be reversed can be passed in as an argument to the function and the reversed string can be returned from the function. The implementation is shown on the next page.

Figure 7.16: Function to reverse

```
1 # A function to reverse the characters in a string
2 def reverseString(inputString):
3     reversedString=""
4     index=len(inputString)
5     while index>0:
6         reversedString=reversedString+inputString[index-1]
7         index=index-1
8
9     return(reversedString)
10
11 oustputString=reverseString("The quick brown fox jumps over the lazy dog")
12 print(oustputString)
```

It should be evident from the above that anytime we want to reverse a string we just need to make a call to our function (similar to line 11) using the string to be reversed as an argument.

Challenge. Modify the above code so that it uses a `for` loop instead of the `while` loop.

Exercises 4 (String Processing)

If possible you should first develop a solution and then, attempt to generalise it by encapsulating each solution in a function.

1. Write a program to count and display the number of vowels in a string.
2. Modify the above program to count and display the number of non-vowel characters also.
3. Write a program to count the number of words in a string. (*Hint:* Use the space character)
4. Write a program to count the number of uppercase letters in a string
5. Extend the program so that the uppercase letters are displayed on a single line of output.
6. Describe in as much detail as you can the differences between the following two snippets of code.

```
# Traversing a string - counting the number of characters
fruit = "banana"
count = 0
for char in fruit:
    if char == "a":
        count=count+1
print(count)
```

```
# Encapuslation using functions

# Returns the number of times 'letter' occurs in ''string'
def countLetters(string, letter):

    count = 0
    for char in string:
        if char == letter:
            count=count+1

    return count

# START EXECUTION FROM HERE
print(countLetters("banana", "a"))

```

7. What would be the effect of adding the following lines of code to the second snippet?

```
print(countLetters("banana", "n"))
print(countLetters("banana", "o"))
print(countLetters("fruit", "p"))
fruit="pineapple"
print(countLetters(fruit, "p"))
letter="p"
print(countLetters("fruit", letter))
print(countLetters(fruit, letter))
```

Generalist the function `countLetters` so that the second argument is also a string.

Chapter Exercises

1. (Write A Program) WAP to count the number of letters in a string entered by the user
2. WAP to print out the first (or last) letter of each word in a string
3. WAP to count the number of occurrences of a particular letter in a string
4. WAP to count the number of words in a string entered by the user
5. WAP to calculate the average word length of the words contained in a string entered by the user
6. WAP to reverse the words of a sentence (e.g. for an input of “get on your horse” the output would be “horse your on get”). SEE CASE STUDY
7. WAP that reverses the letters of each words and the words of a sentence (e.g. for an input of “get on your horse” the output would be “esroh ruoy no teg”)
8. As an extension to the above exercises write a program that restores the encoded string to its original state.
9. An acronym is a word formed by taking the first letters of the words in a phrase and making a word from them. For example, TLA is an acronym for .three letter acronym. WAP that allows the user to type in a phrase and then outputs the acronym for that phrase. Note: the acronym should be all uppercase, even if the words in the phrase are not capitalised.
10. WAP that encodes English language sentences into Pigs Latin. Many variations of Pigs Latin exist – attempt to implement the following two:
    a. Insert ‘eg’ at the end of every vowel in every word of the sentence so that an input of ‘*She sat under the table*’ would become ‘*Sheeg saegt uegndeegr theeg taegble*’
    b. Move the first letter of every word in the input sentence to the end of that word and the add on ‘ay’. In this way an input of ‘*He switched on the computer*’ would become ‘*ehay witchedsay noay hetay omputercay*’
11. WAP to remove all leading spaces from a string. (Extend your program to remove trailing spaces, all spaces, all punctuation marks from a string)
12. A *palindrome* is a word or sentence that reads the same forward as it does backward. WAP to test if a string is a palindrome or not. (Easy – if you think about it!)
13. A *pangram* is a phrase that contains all of the letters of the alphabet. WAP to test if a string is a pangram or not. (Difficult)
14. Write a program to compare two strings (Difficult)

Working with Dates

15. Write a program to accept a date in the format DDMMYY and output the month in words (if it is a valid value). For example an input of 010113 would result in “January” being output.
16. Write a program to accept a date in the format DDMMYY and determine whether it is valid or not. If the date entered is invalid print an appropriate error message to the user.
17. Write a program to accept a date in the format DDMMYY and output the date in long format (if it is valid). For example an input of 010113 would result in “1st January 2013” being output.

Ciphers
18. Pg 119 Zelle

Chapter 8

Lists (Arrays)

INTRODUCTION

By now you should realise that a string is made up of a collection of single characters. Since each of these characters is actually another string (albeit a single character string) we could define a string recursively as a collection of single character strings. For this reason we say that strings are an example of a **compound datatype**. This can be contrasted with other datatypes such as `int` and `float` which are examples of **simple datatypes**.

Lists are another example of a compound datatype. Unlike strings, which can only be composed of other strings, lists can contain values of any datatype. (A simple analogy is a shopping list - this is made up of a collection of numbers and strings e.g. 8 cartons of milk, 3 slice pan etc.)

A list is therefore an ordered collection of values whose datatypes can be different. Lists are known as **arrays** in many other programming languages. (There are some technical differences between lists and arrays, but these differences are so subtle that from a beginner's viewpoint it is fair to consider lists and arrays as the same thing.)

LEARNING OBJECTIVES

1. What is an array?
2. Array operations – indexing and slicing
3. Aliasing and Cloning an array

LECTURE DISCUSSION

Lists can be initialised by enclosing comma delimited values with square brackets.

```
daysOfWeek = ['Sun', 'Mon', 'Tue', 'Wed', 'Thur', 'Fri', 'Sat']
suits = ['diamonds', 'hearts']
cards = ['Jack', 'Queen', 'King']
shoppingList = [6, 'Milk', 3, 'Bread']
dice = [1, 2, 3, 4, 5, 6]
```

The following string operations (which you learned earlier) also apply to lists.

Table 8.1: Concatenation

List Operation	Concatenation

```
suits = ['diamonds', 'hearts']
cards = ['Jack', 'Queen', 'King']
print(cards+suits)
```

The two lists are joined together using the '+' operator. The above code yields the following output:

['Jack', 'Queen', 'King', 'diamonds', 'hearts']

Table 8.2: Indexing

List Operation	Indexing

```
shoppingList = [6, 'Milk', 3, 'Bread']
dice = [1, 2, 3, 4, 5, 6]

print(shoppingList[1])
print(shoppingList[3])

for i in range(0, 6):
    print(dice[i])
```

*Individual elements of a list can be accessed by **indexing**.* The first element of every list occurs at index position zero, the second element at index position one and so on. The index is specified inside square brackets positioned immediately after the name of the list variable (or actual list.) The above code displays the following output.

```
Milk
Bread
1
2
3
4
5
6
```

```
dayNo = input("What day (1-7)? ")
dayNo = int(dayNo)
daysOfWeek = ['Sun', 'Mon', 'Tue', 'Wed', 'Thur', 'Fri', 'Sat']
if (dayNo >= 1) and (dayNo <= 7):
    print("The day is", daysOfWeek[dayNo - 1])
```

The above code prints "The day is" followed by the abbreviated day. This is the day held in the list at the position entered by the user. Remember that lists (like strings and arrays) use a zero-based index and so we subtract one from the index.

```
shoppingList = [6, 'Milk', 3, 'Bread']
print(shoppingList)
shoppingList[0]=2
shoppingList[3]='Brown Bread'
print(shoppingList)
```

While lists and strings have lots in common one important distinction is that **lists are mutable** (i.e. their contents can be changed) while strings are not. This means that elements can be inserted into lists, removed from a list, or simply changed.)

The above code would result in the following:

[6, 'Milk', 3, 'Bread']
[2, 'Milk', 3, 'Brown Bread']

Table 8.3: Slicing

List Operation	Slicing

We can extract sub-lists from lists using the exact same slicing technique that we used to extract substrings from strings. This is demonstrated by the following code.

```
# Extract to demonstrate list slicing
fruits = ['apple', 'pear', 'orange', 'banana', 'kiwi']

print(fruits[1:])
print(fruits[1:3])
print(fruits[:3])
```

The above code causes the following output to be displayed:

['pear', 'orange', 'banana', 'kiwi']
['pear', 'orange']
['apple', 'pear', 'orange']

It is worth pointing out that the sliced list can be assigned to a variable thereby creating another list. For example the following code results in the new list being stored in the variable nonIrishFruits

```
# Extract to demonstrate list creation using slicing
fruits = ['apple', 'pear', 'orange', 'banana', 'kiwi']

nonIrishFruits=fruits[2:5]
```

The contents of this new list are: ['orange', 'banana', 'kiwi']

In reality, a programmer will probably want to do some more sophisticated processing than just printing the list value on each iteration – the previous snippet just demonstrates how a list can be traversed. An alternative implementation using the `while` loop is shown next.

```
# Extract to demonstrate list traversal - using a while loop
daysOfWeek = ['Sun', 'Mon', 'Tue', 'Wed', 'Thur', 'Fri', 'Sat']
noOfDays=len(daysOfWeek)
i=0
while (i < noOfDays):
    print(daysOfWeek[i])
    i=i+1
```

It is important to understand the use of the variable `i` as the index to the list, in the above examples.

Finally, the following implementation of a list traversal algorithm is worth mentioning. Once again this is just an alternative to the preceding two examples.

```
# Simple list traversal (no indexing)
daysOfWeek = ['Sun', 'Mon', 'Tue', 'Wed', 'Thur', 'Fri', 'Sat']
for listItem in daysOfWeek:
    print(listItem)
```

It should be noted that the above implementation does not make use of either indexing or the `len` function and could therefore be considered a simpler more elegant solution to the problem of list traversal.

List methods

A number of list methods (i.e. types of functions only associated with lists) deserve some special mention.

Table 8.4: Method call table

Method call	Description
`lst.append(item)`	Appends the `item` to the list named `lst`
`lst.count(item)`	Returns count of how many times `item` occurs in `lst`
`lst.extend(anotherList)`	Appends the list contained in `anotherList` to `lst`
`lst.index(item)`	Returns the lowest index in list that `item` appears
`lst.insert(index, item)`	Inserts `item` into `lst` at offset `index`
`lst.pop()`	Removes and returns last item from `lst`
`lst.remove(item)`	Removes `item` from the `list`
`lst.reverse()`	Reverses the order of all the items of `lst`
`lst.sort()`	Sorts objects of `lst`

The following block of code demonstrates the use of these methods. The last line (line 25) results in the list ['apple', 'apple', 'banana', 'kiwi', 'orange', 'pear', 'peas'] being displayed (i.e. at the end of the program this is the contents of the list `fruits`.)

You should read through the code carefully and try to understand how the final output is arrived at.

Figure 8.1 List methods

```
1  # A short program to demonstrate the use of list methods
2  fruits=['pear', 'apple', 'orange', 'banana', 'kiwi']
3  fruit='apple'
4  vegs=['peas', 'carrots']
5
6  # Append fruit to the fruits list
7  fruits.append(fruit)
8  # Returns the number of apples in the fruits list
9  fruits.count(fruit)
10 # Appends the list vegs to the list fruits
11 fruits.extend(vegs)
12 # Returns the lowest index in fruits list that fruit appears
13 fruits.index(fruit)
14 # Inserts apple into list at offset 2
15 fruits.insert(2, fruit)
16 # Removes and returns last item from the fruits list
17 fruits.pop()
18 # Removes the first apple from the list fruits
19 fruits.remove(fruit)
20 # Reverses all the items in the fruits list
21 fruits.reverse()
22 # Sorts all the items in the fruits list (alphabetically)
23 fruits.sort()
24
25 print(fruits)
```

Quick Exercise – test your understanding of list methods
Write down the value of countries at the end of each line (lines 6-12) of code below:

Figure 8.2: List Methods

```
2 countries=['Ireland', 'England', 'Scotland', 'Wales']
3 country='France'
4 poles=['North Pole', 'South Pole']
5
6 countries.insert(1, country)
7 countries.append(country)
8 countries.extend(poles)
9 countries.pop()
10 countries.remove(country)
11 countries.sort()
12 countries.reverse()
13
14 print(countries)
```

List examples and exercise (these examples demonstrate the use of lists in game development.)

Example 1
The following program simulates the selection of a random card from a deck of cards. (i.e. pick a card, any card.)

Figure 8.3: Pick a card!

```
1 # Program to pick a card from a pack
2 import random
3
4 suits=['Hearts','Diamonds','Spades','Clubs']
5 faces=['A','1','2','3','4','5','6','7','8','9','J','Q','K',]
6
7 # pick a random number between 0 and 3 (inclusive)
8 r1=random.randint(0,3)
9 suit=suits[r1]
10
11 # pick a random number between 0 and 12 (inclusive)
12 r2=random.randint(0,12)
13 face=faces[r2]
14
15 card=face+' of '+suit
16 print(card)
```

As an exercise extend the above program to deal five unique cards. Start off by encapsulating the above code in a function called `dealCard`. In phase 2 you will need to create a loop inside which you call `dealCard` 5 times. (*Hint:* you will need to build up a new list – call it `hand` – and each time a random card is generated append it to hand only if it is not already in the hand.)

Example 2
The following program using a list to keep track of the frequency of outcomes from a roll of a 6-sided dice. (Each position in the list is used to represent the number of times that number came up.)

Figure 8.4: 6-sided dice

```
# Program to keep a count of the number of times each face of a dice is rolled
import random

count=[0, 0, 0, 0, 0, 0,]

# pick a random number between 1 and 6 (inclusive)
roll=random.randint(1,6)
count[roll-1]=count[roll-1]+1

print(count)
```

Exercises

Extend the above code to run 1000 times (*Hint*: `for i in range (1000):`)

Modify your program to simulate the rolling of two dice – how many rolls before the sum of the two dice make seven?

Try keeping track of the frequencies of the sum of both die.

List Aliasing and List Cloning

An **alias** for a list is simply another name for the same list whereas a **clone** is an exact copy of a list. In the listing below the variable `count2` is an alias for `count` 1 and `count` 3 is a clone. (Note how the cloned list is created using slicing.). Any changes made to an object are also made to its alias (and vice versa).

The operator `is` (and `is not`) can be used to determine whether two objects are aliases of one another or not (i.e. whether they are really the same object or not.) The `is` operator returns `True` if the objects being compared are aliases of one another.

Try running the following code which demonstrates the difference between aliasing and cloning.

Figure 8.5: *List Aliasing*

```
# Program to demonstrate list aliasing and list cloning
count1=[0, 0, 0, 0, 0, 0,]
count2=count1
count3=count1[:]

print(count1==count2)
print(count1 is count2)
print(count1 == count3)
print(count1 is count3)
```

In the above code any changes that are made to count1 are also made to count2. However, since count3 starts out as a clone of count1, changes to count1 have no effect on count3.

Passing lists as parameters

When lists which have been passed into a function are changed inside that function the change(s) remain in effect outside the function (i.e. after control has been returned from the function). *This is a very powerful technique employed by programs to avoid having to use dreaded global variables.*

Consider the function `rollDice` below which encapsulates the code in example 2 on the previous page.

Figure 8.6: Lists as parameters

```
1  # Program to demonstrate parameter passing with lists
2  import random
3
4  count=[0, 0, 0, 0, 0, 0,]
5
6  # pick a random number between 1 and 6 (inclusive)
7  def rollDice(freqList):
8      roll=random.randint(1,6)
9      freqList[roll-1]=freqList[roll-1]+1
10
11 for i in range(5000):
12     rollDice(count)
13
14 print(count)
```

Line 12 of the above code passes in the list referenced by `count` as an argument into the function `rollDice` where it is received as the parameter named `freqList`. It is important to recognise that `freqList` is an alias for `count` (as opposed to a clone). Any changes made to the contents of `freqList` inside the function will therefore be reflected in the list `count` when the function returns.

The above process of parameter passing is known as passing by reference and is an important technique used by programs to share information between functions.

AND FINALLY SOME TIPS TO REMEMBER

- ✓ MAKE SURE YOU UNDERSTAND A PROBLEM BEFORE YOU TRY TO SOLVE IT

- ✓ DESIGN YOUR SOLUTION USING PEN AND PAPER BEFORE WRITING YOUR PROGRAM.

- ✓ MAGINE WHAT THE RUNNING PROGRAM WOULD LOOK OUT – screens, inputs, outputs – AND WORK TOWARDS THAT.

- ✓ ONCE YOU HAVE YOUR PROGRAM WRITTEN NEVER ASSUME IT WORKS CORRECTLY

- ✓ ALWAYS ASSUME YOUR PROGRAM DOES NOT WORK CORRECTLY!

- ✓ TEST FOR NORMAL CONDITONS FIRST. ENSURE YOUR TEST CASES CAUSE EVERY LINE OF CODE TO BE EXECUTED AT LEAST ONCE and CHECK BOUNDARY CONDITIONS

- ✓

Also,

✓

- Develop a style of programming and stick to it consistently e.g. be consistent with the number of spaces you use for indentation
- Keep `import` statements at the top of your file
- Functions must be defined before they are called so keep all function definitions towards the top of your program (after the imports)
- Use two/three blank lines to separate functions
- Use meaningful names for variables and functions.
- Variable names cannot begin with a number (or contain spaces). Same for function names.
- Use lowercase letters for naming variables and functions (e.g. address). If a variable or function name is two words use ‘camel case’
- Avoid the use of global variables if at all possible – pass information using parameters and as return values.
- Get into the habit of using lots of comments to explain your code
- Practice, practice, practice – you won’t get it right first time so keep going back over the exercises

Good luck and enjoy the challenge of programming!