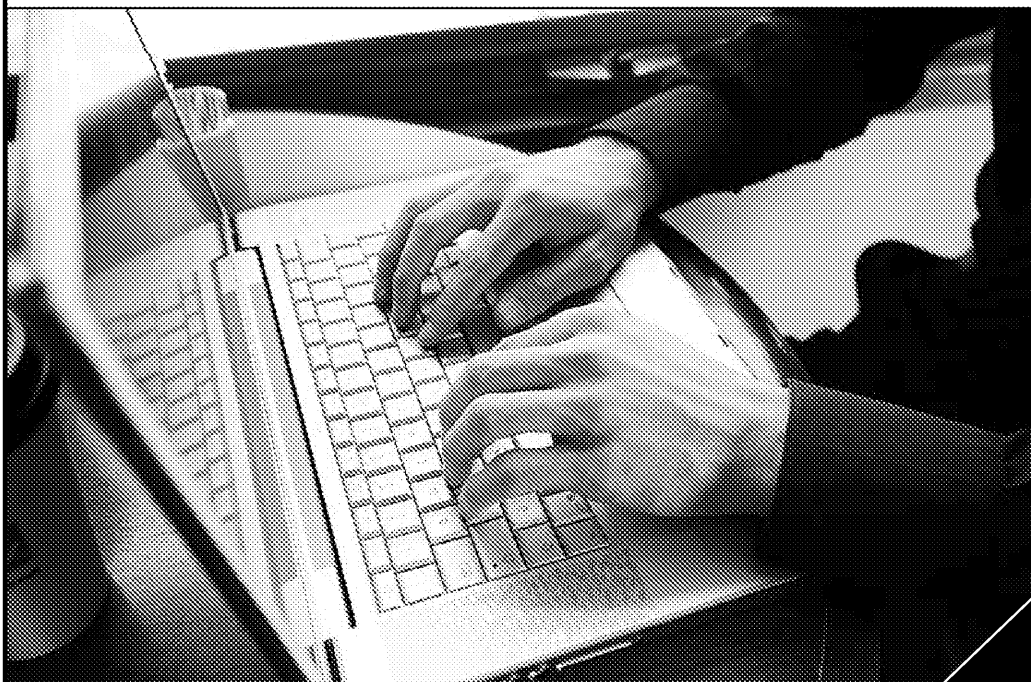


Python for Key Stage 4

P DAWKINS



POD 5083



computing@zigzageducation.co.uk
zigzageducation.co.uk

*Photocopiable/digital resources
may only be copied by the
purchasing institution on a single
site and for their own use*

Become a published author...

Register@

PublishMeNow.co.uk

Contents

INSPECTION COPY

COPYRIGHT
PROTECTED



Thank You for Choosing ZigZag Education	
Teacher Feedback Opportunity	
Terms and Conditions of Use	
Teacher's Introduction	
<i>Key to Conventions Used in this Resource.....</i>	
<i>Specification Mapping.....</i>	
Introduction: Computer Science and Python	
0.1 <i>Definitions</i>	
<i>Introduction Practice Exercise</i>	
Chapter 1: Numbers and Basic Operations.....	
1.1 <i>Let's Write Our First Program.....</i>	
1.2 <i>Order of Operation.....</i>	
<i>Practice Exercise 1</i>	
Chapter 2: Values, Variables and Expression	
2.1 <i>Definitions.....</i>	
2.2 <i>Multiple Assignments</i>	
<i>Practice Exercise 2</i>	
Chapter 3: Data Types	
3.0 <i>Data Types in Python.....</i>	
3.1 <i>Integers.....</i>	
3.2 <i>Floating-Point Number.....</i>	
3.3 <i>Strings.....</i>	
3.4 <i>Other String Operations</i>	
3.5 <i>Upper-Case Character.....</i>	
3.6 <i>Testing for Membership Using the In Operator</i>	
3.7 <i>Converting Data Type.....</i>	
3.8 <i>Taking Input from the User.....</i>	
<i>Practice Exercise 3</i>	
Chapter 4: Functions.....	
4.1 <i>Definition.....</i>	
4.2 <i>Parameters and Arguments.....</i>	
4.3 <i>Saving Your Program Code.....</i>	
4.4 <i>Program Flow.....</i>	
<i>Practice Exercise 4</i>	
Chapter 5: Control Structures	
5.1 <i>Selection or Conditional Statements.....</i>	
5.2 <i>Loops</i>	
<i>Practice Exercise 5</i>	
Chapter 6: List.....	
6.1 <i>Definition.....</i>	
6.2 <i>List Operations.....</i>	
6.4 <i>Editing a List.....</i>	
6.5 <i>Lists of Lists.....</i>	
6.6 <i>List Methods.....</i>	
6.7 <i>List Methods.....</i>	
<i>Practice Exercise 6</i>	

Chapter 7: Working with Files.....	
7.1 Writing to a File.....	
7.2 Printing a File to the Screen.....	
7.3 Appending a File	
7.4 File Path	
7.5 Working with Databases.....	
7.6 SQL.....	
7.6 The WHERE Clause.....	
7.7 SQL and Python.....	
Practice Exercise 7	
Chapter 8: Creating Your Own Type – Classes.....	
8.1 Object-Oriented Programming (OOP).....	
8.2 Why Do We Use Classes?.....	
8.3 Data Types in Python.....	
8.4 Defining a Class	
Practice Exercise 8	
Chapter 9: Dealing with Errors.....	
9.1 Exceptions and Errors	
9.2 Syntax Error.....	
9.3 Runtime Errors	
9.4 Semantic or Logical Error.....	
9.5 Dealing with Errors.....	
Practice Exercise 9	
Project Task	
Answers	
Practice Exercises Solutions.....	
Project Task Solution	
Glossary	
References.....	

**COPYRIGHT
PROTECTED**



Teacher's Introduction

The resource is ideal for anyone wanting to learn how to program in Python, in Computer Science at KS4. It may also be useful for introducing KS5 students to Python, although this was not the intention of the resource.

Python for KS4 will introduce students to Python using a practical approach. This is followed by specific syntax, followed by examples. The interpreter is used through hands-on experience. There are extensive **practice exercises** at the end of each chapter, test what has been covered in each chapter. The practice exercises will have a range of difficult questions, with the majority of the questions designed to test student understanding. A **task** is a scenario-based exercise, similar to a controlled assessment, and is designed to test experience in developing a larger solution.



Answers and worked solutions are provided in printed format (at the end of the book) and as data files on the accompanying CD-ROM.

Optional: each chapter is summarised in videos on my YouTube channel; you can watch the videos for free. Please note that the purpose of these videos is to supplement the resource, not as a substitute for this resource. <http://www.youtube.com/user/syringe876>

How to use this resource

To be an effective computer scientist and programmer requires good theoretical knowledge and practical skills. The aim of this guide is to offer support to achieve both of these.

It is recommended to use a "flipped classroom" method with this guide, although a traditional method. Here, teachers can use technology (the resource and videos in this case) to introduce students to the key concepts before their lesson which will give the teacher time to work with students in the classroom and to solve more difficult problems.

- 1) The teacher should give students access to chapter notes, examples and exercises.
 - 2) The students should read, practise and watch the companion videos before the lesson.
 - 3) The teacher should guide the students through solving the practice exercises.
- Ideally, these should be done during the lesson. Hint: adapt the solution to the problem.

The project task was deliberately created to use most of the concepts covered in the guide, focusing on the use of data structure (class and list) for a working solution. Teachers can use the project task as an outcome, focusing on how students use functions, variables, classes and other concepts. Other project tasks and assessments have a similar structure.

Teachers should allow students to refer to and use this guide in preparation for other project work, as there is a natural build-up to this.

About the author

Patrick Dawkins is a qualified teacher with more than 10 years teaching experience. He read for a BSc in Computer Science and Mathematics. He obtained the qualified teacher status (QTS) from Brunel University, England. He lectured in Computer Science and Mathematics at a teacher training institution for two years. He has served as an examiner for Mathematics and Computer Science. He now teaches GCSE and GCE ICT, GCSE Computer Science and GCE Mathematics. While not teaching or writing, Patrick enjoys reading, exercising and watching Criminal Minds.

Free

Register your email to receive updates* made to resources your school subscribes to.
Go to [Zig Zag Education](#)

* resulting from minor updates to resources
based on feedback from teachers and peer reviews

INSPECTION COPY

**COPYRIGHT
PROTECTED**



Key to Conventions Used in this Resource

- 1) Program code that can be interpreted is displayed using the Courier New font.

Example:

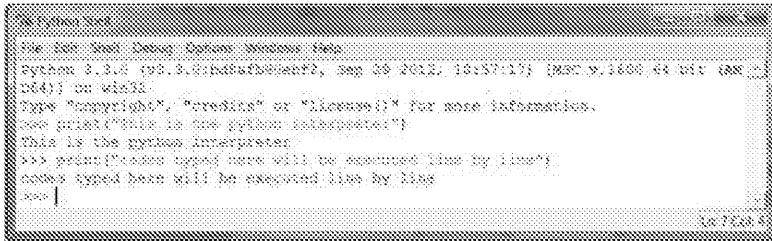
```
print("This code can be interpreted")
```

- 2) Place holders (normally used for variables) are denoted by the use of angle brackets.

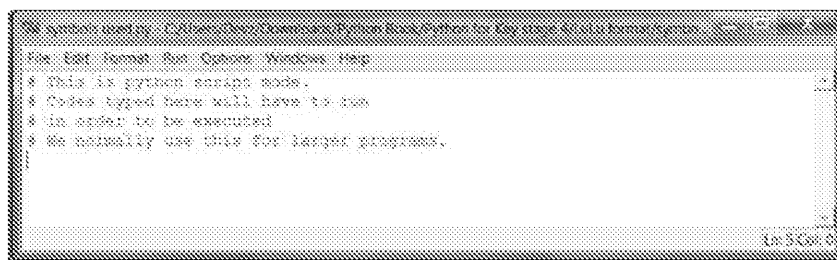
Example:

```
len(<string_name>)
```

- 3) This is the Python interpreter. Codes typed here will be interpreted.



- 4) This is the script mode. Codes written in this mode will have to run in order to be executed. We normally use this for larger programs. Notice that there are no chevrons.



Version of Python

The version of Python that is used throughout this resource is version 3.3.0.

**COPYRIGHT
PROTECTED**



Specification Mapping

AQA GCSE Computer Science (4512)

Subject Content	Learning Outcomes
3.1.1 Constants, Variables and Data Types	Be able to describe the difference between constants and variables Understand when to use constants and variables in problem-solving Understand the different data types available to them Be able to explain the purpose of data types within code Understand and be able to program with one- or two-dimensional arrays
3.1.2 Structures	Be able to explain what a data structure is Be able to create their own data types Understand and be able to explain why data structures can make code simpler
3.1.3 Program Flow Control	Understand the need for structure when designing a coded solution Understand and be able to describe the basic building blocks of programming (sequencing, selection and iteration)
3.1.4 Procedures and Functions	Understand what procedures and functions are in programming Know when the use of a procedure or function would make sense in the coded solution Know how to write and use their own simple procedures and functions Know about, and be able to describe, common built-in functions in their chosen language Use common built-in functions in their chosen language when solving problems Understand what a parameter is when working with procedures and functions Know how to use parameters when creating efficient solutions Understand the concept of parameters and return values when using procedures and functions
3.1.5 Scope of Variables, Constants, Functions and Procedures	Know what is meant by the scope of a variable, constant, function or procedure
3.1.6 Error Handling	Be able to discuss and identify the different types of error that can occur (i.e. syntax, runtime and logical) Understand that some errors can be detected and corrected during development Understand that some errors will occur during the execution of a program Know how to detect errors at execution time and how to handle them to prevent the program from crashing, where desirable
3.1.7 Handling External Data	Know how to use an external file to read and write data in a way that suits the programming language used and the problem being solved Know how to read and write from an external database in a way that suits the programming language used and the problem being solved
3.1.15 Database Concepts	Be able to explain the terms record, field, table, query
3.1.15a Query Methods (SQL)	Be able to create simple SQL statements to extract, add and edit data in a database Have experience in using these SQL statements from within their chosen programming language

INSPECTION COPY

**COPYRIGHT
PROTECTED**



Unit A451

Subject Content	Learning Outcomes
2.1.7 Programming: <i>Programming Languages</i>	c) Explain the difference between high-level code and machine code
	d) Explain the need for translators to convert high-level code to machine code
2.1.7 Programming: <i>Control Flow in Imperative Languages</i>	g) Understand and use sequence in an algorithm
	h) Understand and use selection in an algorithm
	i) Understand and use iteration in an algorithm
2.1.7 Programming: <i>Handling Data in Algorithms</i>	j) Define the term variable and constant as used in an imperative programming language
	k) Use variables and constants
	l) Describe the data types integer, real, Boolean, character and string
	m) Select and justify appropriate data types for a given problem
	n) Perform common operations on numeric and Boolean data
2.1.7 Programming: <i>Testing</i>	o) Use one-dimensional arrays
	p) Describe syntax errors which may occur while developing a program
	q) Understand and identify syntax and logic error

Unit A452

Subject Content	Learning Outcomes
2.3.1 Programming Techniques	a) Identify and use variables, operators, inputs, outputs and assignment
	b) Understand and use the three basic programming constructs to control the flow of a program: sequence; conditionals; iteration
	c) Understand and use suitable loops including count- and sentinel-controlled loops
	d) Use different types of data including Boolean, string, integer and real appropriately in solutions to problems
	e) Understand and use basic string manipulation
	f) Understand and use basic file handling operations: open, read, write and close
	g) Define and use arrays, as appropriate, when solving problems

INSPECTION COPY

COPYRIGHT
PROTECTED

Introduction: Computer Science a

What we will learn:

- ✓ What is computer science?
- ✓ Algorithms
- ✓ High-level and low-level languages
- ✓ Compilers and interpreters

What you need to know:

- ✓ How to use a computer
- ✓ How to use, download and install a computer program



Computer science

Programming

Algorithm

Python

Com

0.1 Definitions

Computer science is the study of the design of algorithms, their properties and their mechanical realisation.

An **algorithm** is a collection of unambiguous and executable operations to solve a problem in a minimum amount of time.

Computer scientists are therefore concerned with designing computer programs. Although computers are fairly recent in the form we know them (the first computers were built many years ago), computer science can trace its roots back far before that. Both the art and science of building some mechanical device to execute them date back some 1,500 years. One of the most important skills that a computer scientist must possess is problem solving. The ability to find multiple solutions is prevalent throughout this text. With this in mind, it is one of the most important skills for a computer scientist to be able to assess and compare solutions and choose the best one. In addition to thinking about a solution, this text is concerned with expressing a solution in Python code.

Computer programming is an interesting way of creating instructions for a computer to execute which produces results for the users.

These instructions can be written using programming languages such as C, Java, and Python. This book will focus on programming in Python.

What is Python?

Python is an interpreted, general purpose programming language which can be used for a wide range of programming called **programming paradigms**. Languages that support multiple programming paradigms are called hybrid languages.

Python is an interpreted language. Python can operate in two modes, namely **script mode**. We typically use the interactive mode to test and debug small programs and the script mode for running larger projects.

INSPECTION COPY

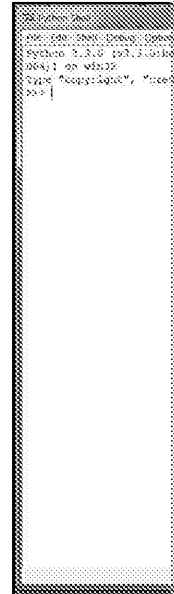
**COPYRIGHT
PROTECTED**



Downloading and Installing Python

Python is an open-source programming language and is therefore free to use. You can download Python from <http://www.python.org/download/>. In this resource we will be using version 3.3.0.

Once you have installed the correct version of Python, you can start the IDLE Python GUI. *Figure 1* shows what the interpreter should look like.



Low-Level vs High-level Language

Python is a **high-level** language. High-level languages are written using codes that are similar to human read language; for example, we use statements such as `print()`, `input()`, to print and take input from the user respectively. One could perhaps guess what these statements do without even seeing them in action.

There are also low-level languages, these are also known as machine language. Computers understand low-level language (machine language) and therefore all high-level languages are translated to machine language before they can be executed. *Table 1* below compares these two types of language.

Low-Level Language	High-level Language
No conversion needed since the computer already understands machine language; as a result, code will execute faster.	Codes have to be converted to machine language and therefore take a long time for conversion.
Much more difficult to read and understand by other programmers as all instructions have to be written in machine code.	Easier to understand and align with everyday language.
Very specific to the computer that the program is written for.	Programs are portable and can be run on any CPU.
Only support primitive data types understood by the computer.	Support a wider range of data types and functionality where the programmer can choose the data type.

Table 1

There are two methods of execution available to the programmer, namely compilation and interpretation.

**COPYRIGHT
PROTECTED**

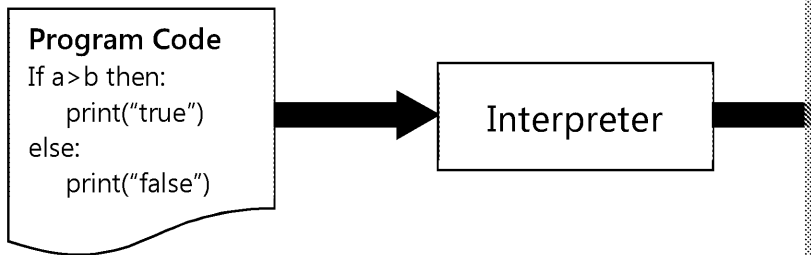


Compiler vs Interpreter

A programming language is interpreted if the source code goes directly to the computer. However, with a compiled language, such as C, an intermediary file is created which is then executed.

The instructions that are written in Python, or any other programming language, are called source code. Compilers then create object code or byte codes of the entire program for the computer. On the other hand, interpreters take each line, translate and execute it before turning its attention to the other line.

Interpreter



Compiler

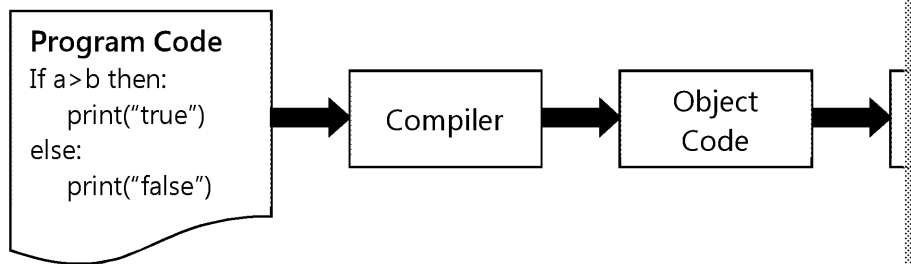


Figure 2

There are clear trade-offs with using compilers and interpreters. Compiled programs execute faster, there are two reasons for this; firstly, the compiler does all the translation. Since all the translation is done during compilation, there is no need to do it again at execution time; therefore the compiled code is executed faster. With the interpreters, each line of code is translated and then executed; the interpreter interleaves, translates and then executes the code. Since the entire program is examined during compilation, the compiler has the ability to optimise and find opportunities for optimisation, whether by using less processing power or a more efficient method for carrying out a computation or by using fewer resources. Interpreters execute one line at a time and therefore use more resources and process time.

Interpreted programs are easier to debug. Quite often one writes a program and it doesn't work as expected but which produces a different output when executed. With a compiler, it's often impossible to stop a program in the middle of execution to examine the state of the program. With an interpreter it is often easy to do this. After all, an interpreter translates and executes one line at a time; therefore you can stop execution after each line. Fortunately, compilers have a facility called **steppers**. Steppers are used to stop programs in the middle of execution (look for errors in) them; these are becoming increasingly available with both compilers and interpreters.

**COPYRIGHT
PROTECTED**



Introduction Practice Exercise

1. Download and install Python.
2. What is computer science?
.....
.....
3. What is the difference between a compiler and an interpreter?
.....
.....
.....
.....
4. What are the advantages and disadvantages of using a compiler?
.....
.....
.....
.....
5. What are the advantages and disadvantages of using an interpreter?
.....
.....
.....
.....
6. The central notion of computer science is that of an algorithm. Explain what an algorithm is.
.....
.....
.....
7. What are the two modes that Python can operate in?
 1.
 2.

INSPECTION COPY

**COPYRIGHT
PROTECTED**



8. Python is a high-level language. How does high-level language differ from low-level language?

.....

.....

9. What do programmers use debuggers for?

.....

.....

.....

10. What is the difference between programming and debugging?

.....

.....

.....

.....

**COPYRIGHT
PROTECTED**



Chapter 1: Numbers and Basic Operations

What we will learn:

- ✓ How to do mathematical calculations
 - Addition
 - Subtraction
- Multiplication and division
- Floor quotient
- ✓ Order of operations

What you need to know:

- ✓ Simple mathematical operations
- ✓ Order of operations



Quotient

Integers

Calculations

Built-in-function

1.1 Let's Write Our First Program

To find the value of any calculation in Python we use the `print()` function. It prints the value of the mathematical statements and return the value of the calculations requested. It is a built-in function that prints to the standard output device, which is the monitor. We will discuss these functions in more detail in a later chapter. The `print()` function can also write to a file.

Syntax:

```
print(<value>)
```

Where `<value>` can be a literal or a variable. In this case, it will print the value to the standard output location.

Examples:

Printing a number

```
>>> print(7)
7
>>>
```

Addition

```
>>> print(2+4)
6
>>>
```

Subtraction

```
>>> print(10-3)
7
>>>
```

INSPECTION COPY

COPYRIGHT
PROTECTED



Multiplication

```
>>> print(4*3)
12
>>>
```

Division

```
>>> print(12/3)
4.0
>>>
```

Exponent/Power

```
>>> print(5**2)
25
>>>
```

Floored Quotient

```
>>> print(7//2)
3
>>> print(1//2)
0
>>>
```

In mathematics, if we divide 7 by 2, we get 3.5; if we want to get the floored (down to the nearest whole number) in Python, we can use the floored quotient. It will evaluate to 0 since in mathematics 1 divided by 2 is 0.5.

1.2 Order of Operation

Python evaluates the correct order of operations; therefore, it will give the correct calculation $3*4+2$.

```
>>> print(3*4+2)
14
>>>
```

Hint: $3*4+2$ and $3*(4+2)$
Python knows and understands the correct order of operations and will print the correct result.

is quite different from:

```
>>> print(3*(4+2))
18
>>>
```

The code below prints the number of seconds in a week:

```
>>> print(7*24*60*60)
604800
>>>
```

**COPYRIGHT
PROTECTED**



Practice Exercise 1

1. Write a program in Python to print out the number of seconds in a 3
2. Write a program in Python to print out the number of seconds in a y
3. Use Python as a calculator. The operations are just the same as wha
mathematics.
4. A high-speed train can travel at an average speed of 150 mph. How lo
travelling at this speed to travel from London to Glasgow which is 41
Give your answer in minutes.
.....
5. Using the help facility on Python. Type `help()` to start the online facilit
view the keywords that are available in Python. Get help on the “if”
6. Use the interpreter to execute the following:
 - a. `49 / 7`
 - b. `8**2`
 - c. `20%3`
 - d. `17 // 3`
 - e. `7**3`
7. Use Python to evaluate the following:
 - a. If you are going on holiday to France how many Euros would yo
convert £500 at an exchange rate of £1 = €1.20.
.....
 - b. On return from your holiday, you now have €320. How many C
at an exchange rate of £1 = €1.32. Use Python to calculate this.
.....
8. The volume of a sphere is given by $V = \frac{4}{3}\pi r^3$. Use Python to find the
with a radius of 10 cm.
.....
9. Insert brackets in the expression `36/9-2` to get:
 - a. 2
 - b. 5.12

INSPECTION COPY

**COPYRIGHT
PROTECTED**



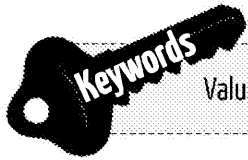
Chapter 2: Values, Variables and

What we will learn:

- ✓ Definitions of values, variables and expressions
- ✓ Assigning vales to variables
- ✓ Evaluate expressions
- ✓ Store values in a statement

What you need to know:

- ✓ Chapter 1: Evaluate calculations in Python



Values

Expressions

Statements

Variables

2.1 Definitions

Values are any literal that can be stored in a memory location. Values can be numbers, characters, or special symbols. We often carry out operations on values.

Examples of values are:

```
6
42
True
False
25
```

A **variable** is a named memory location. We assign values to variables. In Python, a variable name must start with a letter of the alphabet; it can also contain numbers and underscores. Variable names cannot contain spaces. We cannot use keywords for variable names, these are reserved in Python.

Words such as: *4value*, *my age*, *&sales* and *while* are all invalid in Python. This is because they meet the criteria for reserved keywords. *Figure 3* shows a list of reserved keywords used in Python.

```
>>> help('keywords')

Here is a list of the Python keywords. Enter any keyword to get more help:

False      def         if
None       del         import
True       elif        in
and        else        is
as         except      lambda
assert     finally     nonlocal
break      for         not
class      from        or
continue   global      pass

>>>
```

Figure 3

INSPECTION COPY

**COPYRIGHT
PROTECTED**



We should also ensure that meaningful names are used so that one can read the code. Examples of some meaningful names are:

```
age
height
name
length_of_room
item23
position23
```

These are all valid variable names; programmers normally use the underscore for long words as in `length_of_room` above. Another technique used is to join words together, begin each new word with a capital letter, for example `lengthOfRoom` is also a valid variable name.

An **expression** is a combination of operator and operations that evaluate to a value. Examples of expressions are:

```
6
6+5
7*24
(3*23) + 20
a>b
```

An **assignment statement** is used to store the value of an expression in a variable. It consists of an expression to a variable. For example:

```
>>> age = 23
>>> temperature = 20
>>> number_of_students = 28
>>>
>>> minutes_in_a_year = 365*24*60
>>> marks = (3*15) + 12
>>>
```

Hint: The first line is read as 'age equals 23'. After making these assignments, we can use these variables in our program. Example: `print(age * 2)`, `print(minutes_in_a_year)`.

```
>>> print(age, temperature, number_of_students)
23 20 28
>>> print(minutes_in_a_year * 60)
31536000
>>> print(marks * 2)
114
>>>
```

In the example below, we use a variable to store values of the length, width and area of a rectangle.

```
>>> length = 12
>>> width = 4
>>> area_of_rectangle = length * width
>>> print(area_of_rectangle)
48
>>>
```

**COPYRIGHT
PROTECTED**



In this example, we use variables, expressions and statements to find out the number of days all years have 365 days).

```
>>> age = 23
>>> number_of_days_alive = age * 365
>>> print(number_of_days_alive)
8395
>>>
```

2.2 Multiple Assignments

Python allows us to do multiple assignments. Each statement on the right-hand side is stored in the corresponding variable on the left-hand side.

```
>>> age, height = 23, 6
>>> print(age, height)
23 6
>>>
```

The values on the left are actually stored in a tuple (we will discuss this more in Chapter 4) then assigned to the variables on the right. This provides a neat way to assign multiple values in Python. For example:

```
>>> age, height = 23, 6
>>> print(age, height)
23 6
>>> age, height = height, age
>>> print(age, height)
6 23
>>>
```

Notice that after the first assignment, age has the value 23 and height has the value 6. After the second assignment, the values are swapped:

```
age, height = height, age
```

The values are swapped and now age has the value of 6 and height has the value of 23. This is a significant feature and in most other programming languages you will have to use the values of two variables.

**COPYRIGHT
PROTECTED**



Practice Exercise 2

1. Write a program that assigns the variables length and width as 18 and 7 respectively. Use the variables to calculate the perimeter and area of the rectangle. ☐
2. Write a Python program that defines a variable called `days_in_school` and assign 192 to the variable. The program should then print out the total number of days in school from Year 7 to Year 11, assuming that each day you spend 6 hours in school.

Hint:
Area

3. What value will be printed on the screen?

```
marks = 25
marks = marks + 10
print(marks)
```

4. Given the code below, what value will be printed to the screen?

```
time_spent = 34 # in minutes
# after one minutes
time_spent = time_spent + 1
print(time_spent)
```

Hint: We use `#` to include comments. Comments are ignored by the interpreter and are meaningless for the interpreter. They are only information to us humans for maintaining the code.

5. Which of the values below would be printed on the screen from the following code?
- a. 5040
 - b. 210
 - c. 720
 - d. Error

```
hours_in_a_week = hours_in_a_day * 7
hours_in_a_month = hours_in_a_week * 30 # assuming 30 days in
a month
print(hours_in_a_month)
```

6. What is the value of score after running the following code?

```
score = 24
number_of_pieces = 2
new_score = score * 2
```

7. a. True or false? An expression can be assigned to a variable.

- b. What is the value of y after running the code?

```
x, y = 23, 45
y, x = x, y
```

INSPECTION COPY

COPYRIGHT
PROTECTED



8. Which of the following are not valid Python variable names?

name	country1	item34	4gs
&item	_age	bestHeight	tHiS3

9. How do we use the `help()` function in Python?

.....

.....

10. Which of the following statements will assign the value 365 to the variable `days_in_a_year`?

- a. `365 = days_in_a_year`
- b. `there are 365 days in a year`
- c. `days in a year = 365`
- d. `Days_in_a_year = 365`

**COPYRIGHT
PROTECTED**



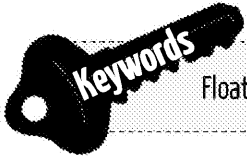
Chapter 3: Data Types

What we will learn:

- ✓ Data type definition
- ✓ Use the type function to find out the data type of an object
- ✓ String operations

What you need to know:

- ✓ Mathematical calculation
- ✓ Assigning values to variables and expressions
- ✓ Evaluate value of statements



Float Integer String List Boolean

3.0 Data Types in Python

Data types tell the interpreter the range of values that can be stored and the operations (addition, subtraction, comparison, concatenation, etc.) that are possible for a given variable. Python has several common data types used in Python. They are also used in most other programming languages.

Data type	Values they store	Python Data Type
Integers	Positive and negative whole numbers	int
Floating-point number	Stores any number that can be represented on the number line Decimal or fractional numbers	Float
List	Holds an array of data	[]
Character	Single character on the keyboard	Char
Boolean	True/false values	
String	Sequence of character	Str

Table 2

3.1 Integers

Integer is one of the three numeric types in Python (int, float and complex). It is a built-in type, which means there is no need to explicitly declare this, but Python knows it by itself. We can find out the type of any object in Python by using the `type()` function. The `type()` function will return the data type of any given object. We will discuss `class` and `object` in detail in the next chapter.

```
>>> value = 35
>>> type(value)
<class 'int'>
>>>
```

We could declare values to be integers explicitly if we want to, for example:

```
>>> x = int(34)
>>> y = int(24.9)
>>> print(x, y, x-y)
34 24 10
>>>
```

INSPECTION COPY

COPYRIGHT
PROTECTED



We declare the variable value to be of type integer; notice that 24.9 will be hence 24 will be stored in the variable y. In Python, integers have unlimited the main operations that are defined for integers in Python.

Operation	Example	Exp
Addition	$x + y$	Sum of x and y
Subtraction	$x - y$	Subtract y from x
Multiplication	$x * y$	Multiply x by y
Division	x / y	Divide x by y
Integer division	$x // y$	Divide x by y and return
Modulo	$x \% y$	Remainder of x / y
Negate	$-x$	Negative value of x
Absolute value	$\text{abs}(x)$	Absolute value or magn
Cast	$\text{int}(x)$	x converted to an integ
Power	$\text{pow}(x, y)$	x raised to the power o
Power	$x ** y$	x raised to the power o

Table 3

3.2 Floating-Point Number

Floating-point numbers are similar to real numbers as encountered in mathematics. The difference is that a floating-point number can have a decimal point. In Python, this is based on the data type called `double` in programming language C. The Python interpreter implements data type based on the underlying machine, and therefore the machine dependent. We use the keyword `float` for floating-point number in Python.

```
>>> value = 56.34
>>> value
56.34
>>> type(value)
<class 'float'>
>>>
```

Similar to integers, there is no need to explicitly declare a variable to be of a certain type. The operations defined in Table 3 above.

3.3 Strings

A string is a collection of characters. Strings are essential in all programming operations defined with them. In Python, we use single or double quotation to define a string.

Which quote to use?

Strings begin and end with single or double quotes. You must end with the same type of quote that you started with. This makes it possible to include quotations within a text. For example:

```
>>> message = "Hello Ben I'm happy to learn about strings today"
>>> print(message)
Hello Ben I'm happy to learn about string today
>>>
```

**COPYRIGHT
PROTECTED**



Python also allows users to use single quotation marks in a text to allow the example:

```
>>> message = 'Martin Luther famous speech "I have a dream"'
>>> print(message)
Martin Luther famous speech "I have a dream" is very inspiring
>>>
```

Hint: Begin and end quotation marks.

3.3.1 Assigning Strings to a Variable

The following code assigns the string September to the variable month:

```
>>> month = "September"
>>> month
'September'
>>>
```

It is possible to use square brackets and index number to access individual characters in a string.

```
>>> month = "September"
>>> print(month)
September
>>> letter = month[1]
>>> print(letter)
e
>>>
```

Above, we assign *letter* to *month[1]*, and the letter *e* is printed. This may be the second character in the word 'September'; it turns out that the first position in the string is the letter 'S', therefore the word September is stored in this manner in Python.

Position →	0	1	2	3	4	5	6	7	8
Content	S	e	p	t	e	m	b	e	r

A loop can be used to go through the string month, we will discuss the "for" loop when we discuss control structures.

**COPYRIGHT
PROTECTED**



`month = "September"` will make this assignment:

```
>>> month
'September'
>>> i=0
>>> for letter in month:
    print(month[i])
    i=i+1

s
e
p
t
e
m
b
e
r
>>>
```

The emphasis here is on the values of *i*; notice that first time *i* is assigned to 0 and therefore the first print statement will execute `month[0]`.

Hint: Whenever you start from the end of a string, the character being -1 will print *b*.

What will `month[-5]` print?

Hint: Both `month[10]` and `month[-10]` will result in an `IndexError: string index out of range`.

```
>>> month = "September"
>>> month[10]
Traceback (most recent call last):
  File "<pyshell#69>", line 1, in <module>
    month[10]
IndexError: string index out of range
>>> month[-10]
Traceback (most recent call last):
  File "<pyshell#70>", line 1, in <module>
    month[-10]
IndexError: string index out of range
>>>
```

The index error is a common programming error and should be avoided.

**COPYRIGHT
PROTECTED**



3.4 Other String Operations

3.4.1 Subsequence

In Python we can select a subsequence of a string by using a colon in square brackets and the start and end position that is required. Python will print the first position and stop at the end position. If we leave the start position out, then Python will start from the beginning of the string; in a similar manner if we leave the end position out then Python will print the rest of the characters in the string.

Syntax: `<string>[<start position>:<end position>]`

Example

```
>>> month = "September"
>>> month[3:6]
'tem'
>>> month[:6]
'Septem'
>>> month[6:]
'ber'
>>>
```

3.4.2 Finding Strings within Strings

The `find()` method returns an integer that represents the starting position of a sub-string within a string. If the sub-string is not contained in the string then -1 is returned.

Syntax: `<string>.find(<substring>)`

Example:

```
>>> hoyle='It is the true nature of mankind to learn from every action'
>>> hoyle.find('nature of mankind')
15
>>> hoyle.find('for every actions')
-1
>>>
```

3.4.3 Finding the Length of a String

The `len()` method returns the number of characters in a string; this includes spaces and punctuation. `len()` is an operator that can be used on any collection of objects, as discussed in Chapter 6.

Syntax: `len(<string>)`

Example:

```
>>> month='September'
>>> len(month)
9
>>>
```

**COPYRIGHT
PROTECTED**



3.5 Upper-Case Character

The *upper()* method returns a copy of the string with all letters in upper case.

Syntax: `<string>.upper()`

Example:

```
>>> month='September'
>>> month.upper()
'SEPTEMBER'
>>>
```

3.5.1 Lower-Case Character

The *lower()* method returns a copy of the string with all letters in lower case.

Syntax: `<string>.lower()`

Example:

```
>>> month='Septemebr'
>>> month.lower()
'septemebr'
>>>
```

3.5.2 Concatenate String (+)

Concatenation refers to joining two or more strings together. We use the '+' operator in Python. Concatenation cannot be used for string and numeric data types together.

Syntax: `<string> + <string>`

Example:

```
>>> greeting = 'hello '
>>> name = 'Benjamin'
>>> greeting + name
'hello Benjamin'
>>>
```

```
>>> 'hello' + 15
```

```
Traceback (most recent call last):
  File "<pysHELL#14>", line 1, in <module>
    'hello' + 15
TypeError: cannot concatenate 'str' and 'int' objects
>>>
```

The error above is a *type error*; this happens whenever you try to apply an operation that is not defined over the data type, in this case whenever you try to concatenate string and integers.

**COPYRIGHT
PROTECTED**



Multiplication is defined for strings and integers.

```
>>> 'hello ' *3
'hello hello hello '
>>>
```

3.6 Testing for Membership Using the In Operator

The `in` operator is used to test for membership within a given sequence. This can be used with a string but can be used with any collection of objects. The `in` operator will return `True` if the object is found, and `False` otherwise.

The following code tests to see if `p` appears in the word "September".

```
>>> month = "September"
>>> 'p' in month
True
>>> 'v' in month
False
>>>
```

From the code snippet above, seeing as `p` appears in `September`, `True` is returned when we test if `v` appears in the word.

3.7 Converting Data Type

`int()`

To convert from one data type to another, the built functions are used. To convert a string to an integer, we use the `int()` function.

Syntax: `int(<value>)`

Where `<value>` is the value to be converted to an integer; Python will convert it if possible, otherwise it will give an error message.

Example:

```
>>> value= '45'
>>> type(value)
<class 'str'>
>>> type( int(value))
<class 'int'>
>>>
```

Note that we assign a value to the string `'45'`, this is confirmed by using `type(value)`. On the fourth line we check the type of `int(value)`. This statement converts the string to an integer and hence `int` will print as the type. It is not possible to convert a string to an integer if it is not a number, as we can see from the example overleaf.

**COPYRIGHT
PROTECTED**



```
>>> type( int('Hello'))
Traceback (most recent call last):
  File "<pyshell#134>", line 1, in <module>
    type( int('Hello'))
ValueError: invalid literal for int() with base
>>>
```

We can convert floating-point numbers to integers.

```
>>> value = 45.9483738
>>> type(value)
<class 'float'>
>>> type( int(value))
<class 'int'>
>>> int(value)
45
>>>
```

Note that `int()` will convert floating-point numbers, but in the conversion number is not rounded up or down, the whole number is simply taken and ignored.

float()

`float()` will convert a given value to a floating-point number.

Syntax: `float(<value>)`

Where `<value>` is the value to be converted to a floating-point number, Python can, otherwise it will give an error message.

Example:

```
>>> float(32)
32.0
>>>
```

str()

`str()` will convert a given value to a string.

Syntax: `str(<value>)` where `<value>` is the value to be converted to

Example:

```
>>> str(45)
'45'
>>>
```

Note that `'45'` is now a string.

**COPYRIGHT
PROTECTED**



3.8 Taking Input from the User

input()

We use the `input()` function to prompt the user for data. We can then assign a default expected value is a string, and therefore the converters discussed in the previous section have to be used to change the data to the required type, if this is possible.

Syntax: `input("<prompt>")`

Where `<prompt>` is the message to be printed on the screen, Python will take the user's input (keyboard).

Example:

```
>>> name = input("What is your name? ")
What is your name? Benjamin
>>> print("Hello " + name)
Hello Benjamin
>>>
```

In the following example, the program asks the user to enter two numbers on the screen. Note that this will require type conversion.

```
>>> number1 = int( input("Please enter the first number: ") )
Please enter the first number: 45
>>> number2 = int( input("Please enter the second number: ") )
Please enter the second number: 12
>>> print("The sum is ", number1+number2)
The sum is  57
>>>
```

**COPYRIGHT
PROTECTED**



Practice Exercise 3

1. Write a Python program that stores your favourite quotation in a variable.
2. Define a variable called "name" that assigns your name to it.
3. Write a Python program that asks the user for three numbers and prints their sum.
4. Write a Python program that asks the user for their name and age and prints a message on the screen with a warm message.
5. What is the data type of x in the code snippet below?

```
>>> x = 34.67
>>>
```

.....

6. What letter will be printed on the screen after running the following code?

```
>>> title = "Python for key stage 4"
>>> letter = title[3]
>>> print(letter)
```

.....

7. What letter will be printed on the screen after running this code?

```
>>> title = "Python for key stage 4"
>>> letter = title[-5]
>>> print(letter)
```

.....

8. Use the slice operation / subsequence to print "key" from the variable below.
a. Title = "Python for key stage 4"
9. Use the variable below and the concatenation operation to print the message.
b. Name = "Benjamin"

Greetings to print: "Hello Benjamin, Python is fun"

10. What will the following code produce?
title= "Python for key stage 4"
for letter in title:
print(letter)

.....

INSPECTION COPY

**COPYRIGHT
PROTECTED**



Chapter 4: Functions

What we will learn:

- ✓ Definition of a function
- ✓ Syntax of functions
- ✓ Return values from a function

What you need to know:

- ✓ Variables
- ✓ Expressions
- ✓ Statements
- ✓ Values



Function

Return

4.1 Definition

Functions are blocks of codes that perform specific tasks. It allows us to reuse the same code for performing computations more than once. Functions take an input and can return an output. Functions can take multiple inputs or no input, have multiple outputs or no outputs at all. Python has many built-in functions for us, such as `print()`, `type()`, `abs()`, these functions are pre-defined. We can declare and define our own functions. Functions that are defined by a user are called user-defined functions.

4.1.1 Why Do We Use Functions?

- Using functions makes our program code more readable. When we define a function, we are naming certain sets of instruction that are performing a specific task and making the program code easier to read.
- Functions can reduce our program code significantly by putting repetitive code in a function.
- Change or adjustments can be made in one place.
- Functions facilitate modular design. We can break down a large problem into smaller chunks, work on these chunks in functions then put the entire working piece together.
- Functions can be placed into modules so that they can be reused.

4.1.2 Function Syntax

Syntax:

```
def <name>(<ParameterList>):  
    <block>  
    return <expression>, <expression>
```

Where `<name>` follows the same conventions as a variable.

`<ParameterList>` represents the inputs to the function separated by commas. `<expression>` is used to specify the output. By default, all functions return a value. If one is not specified, the keyword `None` is automatically returned.

```
return <expression>, <expression>, ...
```

Hint: Functions must start with a capital letter and cannot contain spaces.

INSPECTION COPY

COPYRIGHT
PROTECTED



Example:

```
>>> def sum(a, b):
        return a+b

>>> sum(12, 10)
22
>>>
```

Whenever we express what the function should do we call this **defining a function**. In the example is defining the function `sum`. The `a` and `b` are referred to as **parameters** within the definition of the function; the range of the area that the variable can take is called the **scope of the variable**. Note that `a` and `b` are place holders and are only valid within the function definition.

Whenever we use a function, this is referred to as **calling a function**. In the example `sum(12, 10)` is a call to the function with arguments 12 and 10.

4.1.3 Returning Values

We can define functions that just do something without explicitly returning a value. The following function will take a name and say hello twice.

```
>>> def hello_twice(name):
        print("Hello "+name)
        print("Hello "+name)

>>> print(hello_twice("Ahmed"))
Hello Ahmed
Hello Ahmed
None
>>>
```

Notice that after calling the function, `None` is printed as the returned value. This is because the function did not use the `return` keyword in the function definition. `None` is returned by default.

4.2 Parameters and Arguments

Some books will use parameters and arguments interchangeably; however, it is important to note that parameters are the place holders that are used in defining the function, while arguments are the actual values that are supplied during the function call. Parameters are defined in the function definition; in fact the parameters are data types (defines acceptable range of values). An argument is an instance of the parameter and is used in the function call. Multiple arguments can be passed, and are normally executed at runtime.

Python also uses a technique called operator overloading, where operators are used to perform different operations based on the operands that surround it. Since `+` is defined for numbers, it is also defined for strings (namely concatenation), the following will result in `sum` concatenating the strings that it receives.

Hint: In `sum(a,b)`, `a` and `b` are arguments. In `sum(4,5)`, 4 and 5 are arguments.

**COPYRIGHT
PROTECTED**




```
>>> sum('Hello', ' Bobby')
'Hello Bobby'
```

This is a very powerful concept in object-oriented programming and Python language, utilises these features.

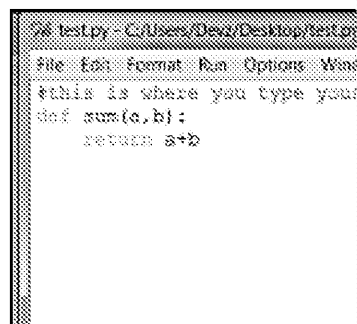
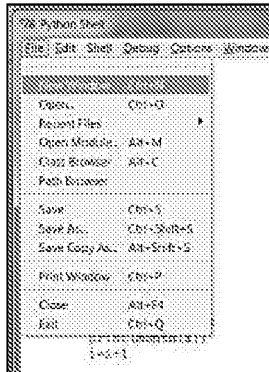
4.3 Saving Your Program Code

Python can operate in two modes, namely the interactive mode and the script mode. You are in script mode if you have the three chevrons (`>>>`) and the blinking cursor at the end of the statement. Python interprets and produces a result. As you have been in interactive mode. You can create a Python file, type all your program code, and then interpret this file. This allows us to make changes and rerun the script. The following will demonstrate how to create a script file.

From the Python interpreter, you can *select file* → *new window* to create a new file. You can type your program code into the new window.

Save this file with a suitable file name and `.py` as the file extension; this will create a Python file. Once you finish typing your program code, you can execute the file by selecting *run* → *run module*. Your output will be displayed in the Python interpreter.

Example: Creating a new script file.



After typing your script you can run it and make changes.

Hint: If we need to add other functions to the sum function. Python is very flexible. For example, we can add a function with strings. For example, we can add a function with two strings will return a string defined for strings.

**COPYRIGHT
PROTECTED**



4.4 Program Flow

A program is executed from top to bottom and line by line unless the flow structure (we will examine control structures in Chapter 6). It turns out that program flow. When a function is called, the execution jumps to the definition, replaces the argument(s) with the parameter(s) (recall that the parameters and arguments are the actual values that are sent to the program). The body of execution continues after the line that calls the function.

This process seems pretty straightforward, but when you imagine that a function can call other functions, keeping track of where you are can be tricky. Further, in Python you can send a function as an argument to another function. Python has a very efficient method of handling this situation. Here are two simple examples to demonstrate this point, but you can imagine that these could be more complex.

Imagine that we have a function called “larger” that returns the larger of two numbers.

```
>>> x = 12
>>> y = 7
>>> z = 79
>>> print(larger(larger(x,y),z))
79
>>>
```

This will print the largest of the three numbers. In the example above, you can see that `larger(x,y)` was an argument to the outer `larger` function. The outer `larger` function returns the value 12, as 12 is larger than 7. Then we evaluate `larger(12,79)` and finally return 79 to the print function.

**COPYRIGHT
PROTECTED**



Practice Exercise 4

1. Define a function called "double", that takes a number and returns
.....
2. Define a function called "perimeter_of_rectangle" that takes two numbers to represent the length and width of a rectangle and returns the perimeter (Remember that the perimeter is twice the length added to twice the width)
.....
3. Define a function called "area_of_circle" that takes one number to represent the radius and returns the area of the circle. (Remember that the area is $\frac{22}{7} * r^2$.)
.....
4. Define a function called "hello_three_times" that takes an argument "name" and prints "hello name" three times on the screen.
.....
5. Write a program that requests the user's name. The program should call a function called "head_name()", that prints the first character from the given name.
.....
6. Write a program that asks the user for their name. The program should call a function called "tail_name()", that returns the tail of the person's name. The tail of a name is the person's name without the first character.
HINT len(<string>) will return the length of a string.
.....
7. Give three advantages of using functions in a program.
 1.
 2.
 3.
8. What is the difference between parameters and arguments?
.....
.....
.....
.....
.....

INSPECTION COPY

**COPYRIGHT
PROTECTED**



9. Define a function called “`power_value()`”, that takes two integers as arguments and returns the first raised to the power of the second.

.....

10. If we had a function called “`absolute_value()`”, that takes a number as argument and returns the positive value of the given number. What value will be returned by the following statements?

```
print(absolute_value(-5))  
print(absolute_value(0))  
print(-absolute_value(-5))
```

.....

**COPYRIGHT
PROTECTED**



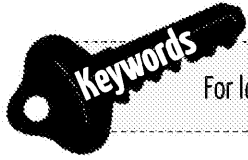
Chapter 5: Control Structures

What we will learn:

- ✓ Selection structures
- ✓ Loops

What you need to know:

- ✓ Data types
- ✓ Functions



For loop

While loop

If selection

If else structures

Control structures are used to affect the natural program flow. In this section, we will explore the control structures that are available in Python.

5.1 Selection or Conditional Statements

The first type of control structure that we will evaluate is statements. Selection statements allow us to execute a particular block of code based on the result from some test/condition. The two selection statements that we will be exploring are IF and IF ELSE statements.

5.1.1 IF Statement

When using the IF statement, we test to see if one condition is true or false. If the condition is true, the block of code following the IF statement will only execute if the condition is true. Below is the syntax for the IF statement.

Syntax:

```
if <condition>:  
    <block>
```

Hint: The IF statement is used when you want to execute a block of code only if a condition is true.

Example:

```
>>> if 5>2:  
    print("5 is larger")
```

```
5 is larger  
>>>
```

Hint: The IF statement is used when you want to execute a block of code only if a condition is true.

INSPECTION COPY

COPYRIGHT
PROTECTED



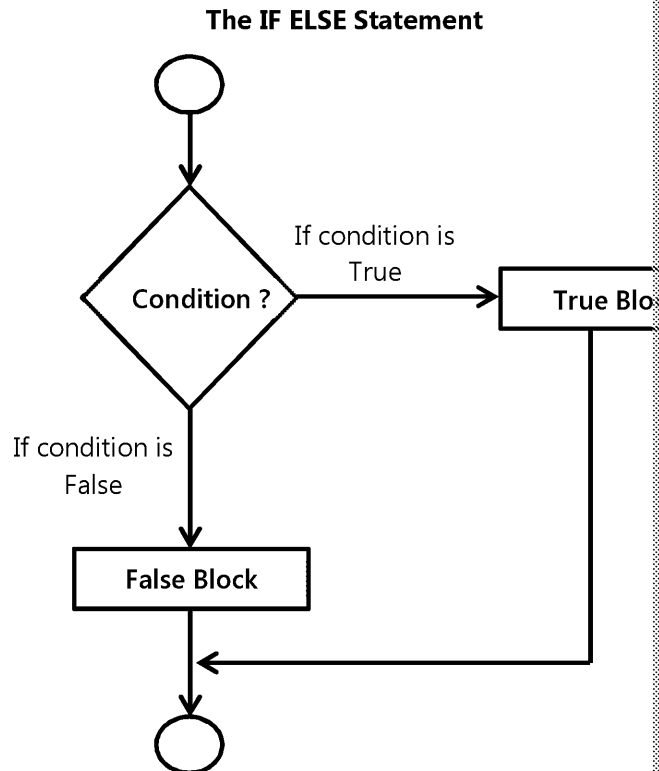
5.1.2 IF ELSE Statement

The IF ELSE statement is an extension of the IF statement; it will execute a block of code if the condition is true and an alternative block if the condition is false.

Syntax:

```
if <condition>:
    <TrueBlock>
else:
    <FalseBlock>
```

<TrueBlock> is executed if the condition is true and <FalseBlock> is executed if the condition is false.



Example:

```
>>> month = 'September'
>>> if len(month)>10:
    print(month + ' has more than ten characters')
else:
    print(month + ' has less than ten characters')

September has less than ten characters
>>>
```

Note: /n and then false, has

**COPYRIGHT
PROTECTED**



5.2 Loops

Loops allow us to execute a block of code a number of times. Each time the code is executed it is called **iteration**. Python offers two looping statements, namely the `for` loop and the `while` loop.

5.2.1 For Loop

The `for` loop iterates over the items in a sequence, which can be a string or a list (see Chapter 6) each time assigning the values in this sequence to a given variable. The condition is that we are at the end of the list.

Syntax:

```
for <variable> in <sequence>:  
    <block>
```

<variable> is a variable that is assigned to each value in the sequence.

<sequence> is a string or a list.

<block> is executed each time through the loop.

Example:

```
>>> month = 'September'  
>>> for letter in month:  
    print(letter)  
  
s  
e  
p  
t  
e  
m  
b  
e  
r  
>>>
```

Example:

If we want to print the month of September, we would assign a variable to hold the values in the list.

The first time it iterates through the loop `s` is assigned to the variable `letter` and the `print` function will print `s`.

The second time through the loop `e` is assigned to the variable `letter` and the `print` function will print `e`.

The third time through the loop `p` is assigned to the variable `letter` and the `print` function will print `p`.

The fourth time through the loop `t` is assigned to the variable `letter` and the `print` function will print `t`.

This continues to the end of the loop. This is determined by the length of the sequence.

Notice that we go through the body of the `for` loop (`print letter`) nine times, once for each letter in the assigned string.

**COPYRIGHT
PROTECTED**



5.2.2 The Range Function

The built-in range function is very useful with the for loop and is used to iterate through numbers.

Syntax:

```
range(<start>, <stop>, <step>)
```

<start> is optional and if it is not present will default to start at zero (0).

<stop> is not optional and is never included in the generated sequence.

<step> can be used to change the default increment from one (1) to another.

Example:

```
>>> print(range(10))
range(0, 10)
>>>
>>> #Now we use range() in a for loop
>>> for i in range(10):
        print(i)

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

Notice that `print(range(10))` will generate a virtual tuple from 0–10, and the for loop iterates through this list 10 times.

**COPYRIGHT
PROTECTED**



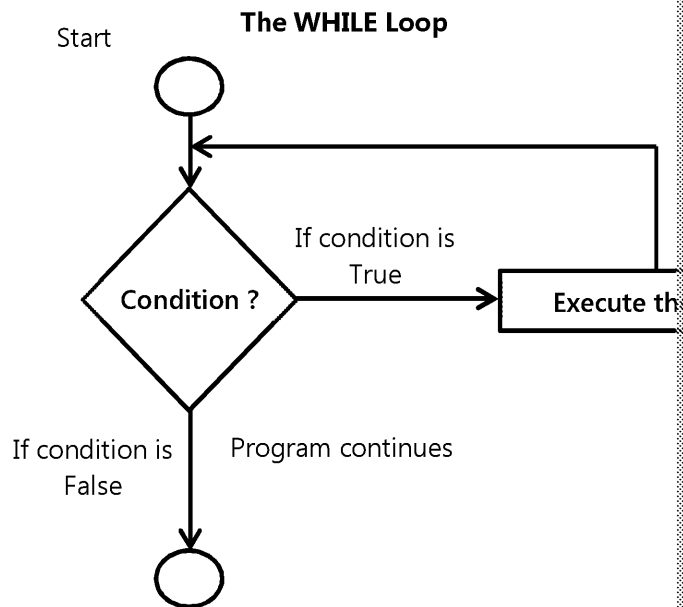
5.3.1 While Loop

While loops are used in situations in which the number of iterations is unknown as long as a given condition is true.

Syntax:

```
while <condition>:
    <block>
```

While loops evaluate whether an expression or condition is true or false. The condition becomes false.



Example:

```
>>> number = 1
>>> while number < 11:
    print(number * number) #Square the number
    number = number + 1
```

1
4
9
16
25
36
49
64
81
100
>>>

Important: There must be a way to break out of the loop that will allow the program to continue. Otherwise, we could be in a situation where the program runs forever.

Here we use the while loop to print the square numbers from 1 to 100. Note that `number = number + 1` will ensure that the condition will become false eventually.

The `#square the number` is a comment. Comments are ignored by the interpreter. We give more explanation to someone reading our program code.

**COPYRIGHT
PROTECTED**



5.3.2 Break

A break allow us to exit the while loop even when the condition is true. A break is used with caution and only if there is no alternative.

Typical Use:

```
while <condition>:
    <code>
    If <BreakTest>:
        Break
    <code>
<code after while loop>
```

Here, during the while loop if the break test is met then execution will jump to the code after the while loop will be executed.

Example:

```
z=[2,5,7,1,90]

def find_seven(p):
    i=0
    while i!=len(p):
        if p[i]==7:
            print("Breaking out of Loop")
            print("Found at position ", i)
            break
        i=i+1
    print("After the while")

find_seven(z)
```

When we run the code above calling `find_seven(z)`, the following output is produced:

```
>>> ===== RESTART =====
>>>
Breaking out of Loop
Found at position 2
After the while
>>>
```

Notice that seven is at position two in the list and once we find seven we exit the loop. Although the while condition (`i!=len(p)`) was still true as `i` is two and `len(p)` is five.

**COPYRIGHT
PROTECTED**



Practice Exercise 5

1. Write a Python program that asks the user to enter two numbers; it should use an IF statement to print the larger of the two.
2. Write a Python program that asks the user to enter a score; it should print a message, based on the score of the user:
 - a. 50 and above → "Pass"
 - b. 0-49 → "Fail"
3. Develop the program below by using a nested IF statement (IF within IF) to add additional information:

50 and above	→ "Pass"
80+	→ "Well Done"

0 - 49	→ "Fail"
<20	→ "You need to try harder."

4. Write a Python program that asks the user for a score and prints a grade based on the following:

80-100	A
60-79	B
40-59	C
30-39	D
< 30	U
5. Suppose we calculate the vowel value of a word based on the following:

a	5 points
e	4 points
i	3 points
o	2 points
u	1 points

Write a Python program that asks the user for a word, then calculate the vowel value of the word entered.

6. Define a function called `larger()` that takes two integers and returns the larger of the two.
.....
7. Define a function called `long_name()` that takes a name input and returns a boolean value (true or false) based on the number of characters in the name. The name is long if it contains more than 14 characters.
.....

INSPECTION COPY

**COPYRIGHT
PROTECTED**



8. Define a function called "*largest()*" that takes three numbers as arguments and returns the largest of the three. HINT: You can write the function from scratch or use the function that was written before.
.....
9. Write a function called "*print_upto()*" that takes a number as argument and prints all whole numbers from 1, up to and including, the given number.
10. Write a function called "*print_even_upto()*" that takes a number as argument and prints the **even** numbers from 1, up to and including, the given number.
11. Write a function called "*magic_number()*" that has a variable as argument. The user should be prompted to guess the magic number. The program should give feedback on the guess.
 - i. If the guess is greater than 7, "Too high"
 - ii. If the guess is less than 7, "Too low"
 - iii. If the guess is correct, "Well Done"
12. Modify the program above to give the user only 5 guesses. If there are more than 5 guesses, then the user should get the following message: "You have gotten your limit."
13. The factorial of a number is the product of all the integers below it. For example, the factorial of 4 is $4*3*2*1 = 24$.
 - a. Write a function called "***factorial()***" that returns the factorial of the given number.
 - b. ***factorial(4)*** → 24

COPYRIGHT
PROTECTED



Chapter 6: List

What we will learn:

- ✓ List definition
- ✓ Syntax for creating lists
- ✓ Selecting elements of a list
- ✓ Selecting subsequence of a list

What you need to know:

- ✓ Data types
- ✓ Assignments



List Sub-list Elements Index Position

6.1 Definition

The list is the most versatile data structure Python uses. It is a mutable collection of items, which can be of the same or different data types. When a data collection is mutable, it means that the items in the collection can be changed. The items are written between square brackets, separated by commas; they can be assigned to a variable.

Syntax:

```
<variable> = [<list_item>, <list_item>, ...]
```

Where `<list_item>` can be empty, or any data type.

Example:

```
>>> days = ["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"]
>>> days
['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday']
>>>
```

6.1.1 Selecting Individual Elements

As a list is a compound data type, we can select individual elements by indexing.

```
>>> days[0]
'Sunday'
>>>
```

Using Negative Index

A negative index will return the value at its position from the end of the list.

```
>>> days[-2]
'Friday'
>>>
```

INSPECTION COPY

COPYRIGHT
PROTECTED



6.2 List Operations

Concatenation

We can concatenate a list by using the concatenation operation (+).

```
>>> x = [1, 2, 3]
>>> y = [6, 7, 8]
>>> z = x+y    #Concatenation Operation
>>> print(z)
[1, 2, 3, 6, 7, 8]
>>>
```

Notice that z now stores the items of x and y.

Multiplication

The multiplication operation has a similar effect on lists as on strings.

```
>>> x = [1, 2, 3]
>>> print(x*3)
[1, 2, 3, 1, 2, 3, 1, 2, 3]
>>>
>>> print(x*4)
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
>>>
```

Notice that multiplying by three produces the list three times.

6.3.1 Selecting a Subsequence in the List – Slicing

We can select a subsequence in a list by using the slice operation. The slice specifies the starting and end points to return a section of a list.

```
>>> days
['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday',
>>> days[2:5]
['Tuesday', 'Wednesday', 'Thursday']
>>> days[:4]
['Sunday', 'Monday', 'Tuesday', 'Wednesday']
>>> days[5:]
['Friday', 'Saturday']
>>>
```

Sub-selection is also called slicing the list. It is important to note that sub-selection creates a copy of the list and does not mutate the original list; from the example above, printing the value of days will give the entire list.

```
>>> days
['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday']
>>>
```

Important: The first number in the slice operation is the starting point and the last number is the end point. If the end point is blank then Python assumes that the start position is the beginning of the list and the end position is the end of the list.

**COPYRIGHT
PROTECTED**



6.4 Editing a List

Because lists are mutable, unlike strings, it is possible to change the values

```
>>> a=[1,3,5]
>>> a
[1, 3, 5]
>>> a[1] = 25 # Change the value at position 1
>>> a #This is the updated a
[1, 25, 5]
>>>
```

Note that we can assign a new edit list by referring to its position and assign

6.5 Lists of Lists

Python allows us to have lists of lists:

```
>>> olympic_host = [{"London",2012}, {"Beijing",2008}, {"Athens",2004}]
>>> olympic_host
[('London', 2012), ('Beijing', 2008), ('Athens', 2004)]
>>> olympic_host[1] #print the first element in the list
('Beijing', 2008)
>>> olympic_host[1][0] #element in pos 0 in the list
'Beijing'
>>>
```

We can make a few observations from the script above.

- It is possible to have different data types in the same list; above we represent a year.
- We can have lists of lists *olympic_host* is a list containing three elements, each element is itself a list containing the Olympic host city and year.
- We can access the individual element in each list by sub-indexing the list, as above *olympic_host[1][0]*.

6.6 List Methods

There are various methods defined on lists, below we will examine some of them.

append()

Append will add an element to the end of the list. Append mutates (changes) the list, the result of invoking append on a list is that the calling list will have the element added to the end.

Syntax:

```
<list>.append(<element>)
```

<list> is the list to be appended.

<element> is the element to be appended to the list.

**COPYRIGHT
PROTECTED**



Example:

```
>>> x=[3,6,2,15,20]
>>> print(x)
[3, 6, 2, 15, 20]
>>> x.append(45)
>>> print(x)
[3, 6, 2, 15, 20, 45]
>>>
```

Notice that after a call to append with argument 45, the list x now has 45 as its last element.

insert()

Insert is used to insert an element at a given index. Insert mutates the existing list and returns the list with the element inserted.

Syntax:

```
<list>.insert(<position>, <element>)
```

<list> is the list that the element should be inserted into.

<position> is the position in the list that the element will be inserted. The first position is position zero (0).

<element> is the element to be inserted into the list.

Example:

```
>>> print(x)
[3, 6, 2, 15, 20, 45]
>>> x.insert(3,35)
>>> print(x)
[3, 6, 2, 35, 15, 20, 45]
>>>
```

Notice that after the insert function is invoked on the list x, 35 is now inserted at the third position.

sort()

The sort function will sort the list.

Syntax:

```
<list>.sort()
```

<list> is the list to be sorted.

Example:

```
>>> y=[35, 30, 23, 38, 10, 15, 2]
>>> print(y)
[35, 30, 23, 38, 10, 15, 2]
>>> y.sort()
>>> print(y)
[2, 10, 15, 23, 30, 35, 38]
>>>
```

**COPYRIGHT
PROTECTED**



count()

Count will return the number of occurrences of an element in a list.

Syntax:

```
<list>.count(<element>)
```

<list> is the list that the count is invoked on.

<element> is the element that you search for the occurrence of.

Example:

```
>>> y
[2, 10, 15, 23, 30, 35, 38, 2]
>>> y
[2, 10, 15, 23, 30, 35, 38, 2]
>>> y.count(2)
2
>>> y.count(15)
1
>>> y.count(45)
0
```

Notice that there are two occurrences of 2 and only one of occurrence of 15 and 45 in the list and hence `y.count(45)` will return 0.

extend()

Extend mutates a list and has the effect of adding a second list to the first. The first list will still exist in its original state.

Syntax:

```
<list1>.extend(<list2>)
```

<list1> is the list to be mutated.

<list2> is the list that will be added to list 1.

Example:

```
>>> x=[1,2,3,4]
>>> y=[5,6,7]
>>> x
[1, 2, 3, 4]
>>> y
[5, 6, 7]
>>> x.extend(y)
>>> x
[1, 2, 3, 4, 5, 6, 7]
>>> y
[5, 6, 7]
```

After `x.extend(y)` is called, x now contains the elements of both x and y and y still exists.

**COPYRIGHT
PROTECTED**



pop()

The pop method will remove and return an element from the list. By default it will remove and return the last element in a list. You can optionally send the index of the element to be removed from the list and `pop()` will return that element.

Syntax:

```
<list>.pop()
```

OR

```
<list>.pop([<index>])
```

`<list>` is the list with the elements.

`<index>` is the optional position of the element to be removed from the list.

Example:

```
>>> x
[8, 5, 12, 9, 20]
>>> x.pop() #Now we invoke pop() on list x
20
>>> x # print the list x
[8, 5, 12, 9]
>>>
```

In the second example we will remove element one. Recall that element one is the second element in the list.

```
>>> x
[8, 5, 12, 9]
>>> x.pop(1) # remove element 1
5
>>> x # print the list
[8, 12, 9]
>>>
```

remove()

Remove will delete the first occurrence of the given value in a list. If the element is not found, then a value error is returned.

Syntax:

```
<list>.remove(<element>)
```

`<list>` is the list with the elements.

`<element>` is the element to be removed from the list.

Example:

```
>>> x=[8,5,12,9]
>>> x
[8, 5, 12, 9]
>>> x.remove(5)
>>> x
[8, 12, 9]
>>>
```

After `x.remove(5)`, 5 is removed from the list.

**COPYRIGHT
PROTECTED**



reverse()

`reverse()` will reverse the elements in a list; that is the first element becomes last and so on, until the last element becomes first.

Syntax:

```
<list>.reverse()
```

`<list>` is the list to be reversed.

Example:

```
>>> x
[8, 5, 12, 9]
>>> x.reverse() # reverse the elements in place
>>> x
[9, 12, 5, 8]
>>>
```

index()

`index()` will return the position of the first occurrence of the given value. If the value does not exist in the list.

Syntax:

```
<list>.index(<element>)
```

`<list>` is the list to be searched.

`<element>` is the value whose position we are interested in.

Example:

```
>>> x
[8, 5, 12, 9]
>>> x.index(12)
2
>>>
```

6.7 List Methods

len()

In 3.2.3 we discussed the `len()` operator and used it to find the length of a list. It can be used on any collection of objects. It is worth pointing out that we can also find out the number of elements in a list.

Example:

```
>>> x
['Lee', 'Adam', 'Peter', 'Clive']
>>> len(x)
4
>>>
```

**COPYRIGHT
PROTECTED**



In Operator

The `in` operator discussed in 3.3 will test if an object is a member of a collection.

Example:

```
>>> x
['Lee', 'Adam', 'Peter', 'Clive']
>>> 'Lee' in x
True
>>>
```

We will discuss two additional operators that can be used with lists, namely

min()

The `min()` operator will return the smallest value in the list.

Syntax:

```
min(<list>)
```

`<list>` is the list that we want to find the smallest element of.

Example:

```
>>> x
['Lee', 'Adam', 'Peter', 'Clive']
>>> min(x) # return the smallest value of x
'Adam'
>>> y
[5, 6, 7]
>>> min(y) #return the smallest value of y
5
>>>
```

Notice that `min()` will compare both strings and numeric data type.

max()

The `max()` operator will return the largest value in the list.

Syntax:

```
max(<list>)
```

`<list>` is the list that we want to find the largest element of.

Example:

```
>>> x
['Lee', 'Adam', 'Peter', 'Clive']
>>> max(x) #return the largest value
'Peter'
>>> y
[5, 6, 7]
>>> max(y) #return the largest value
7
>>>
```

**COPYRIGHT
PROTECTED**



Practice Exercise 6

1. Given the list `bank_holidays_in_month = [1, 0, 1, 1, 0, 2]`, where each element represent the number of bank holiday in January there is 1, in February 0, March there is 1, etc. Write a function `"bank_holiday()"`, that takes a number to represent the month of bank holidays in that month. Example: `bank_holiday(5) →`
2. Without using the `sort()` function that is defined on a list, write a function `"my_sort()"`, that takes an unsorted list and returns the sorted list.
3. Define a function called `"add_hello()"`, that takes any list and adds "hello" to each element of the list.
4. Define a function called `"discount_ten()"`, that takes a list of numbers and returns a list with each element having a 10 per cent discount.
5. Define a function called `"remove_five()"`, that takes a list as input and removes all occurrences of 5 in the list. Write the function in such a way that you return a new list where there are no 5s in the list.
6. A word is a palindrome if it reads the same in both directions. For example "level", "redder", "madam" are all palindromes. Write a function called `"is_palindrome()"`, that takes a word and returns "True" if it is a palindrome and "False" if otherwise.
7. Define a function called `"unique_elements()"` that takes a list as input and returns a list containing unique elements. The program should take repeated elements and return only one value of the element. For example, in the list `[1,2,2,3,3,4]`, it should return `[1,2,3,4]`.
8. Define a function called `"backwards()"`, that takes a list and returns a new list with elements in reverse order.
9. Define a function called `"sum_list()"`, that takes a list as input and returns the sum of the elements in the list.
10. Define a function called `"mean_list()"`, that takes a list as input and returns the mean of the elements in the list. The mean is the sum of all the elements divided by the number of elements. Use the function above defined in Question 9.
11. Define a function called `"list_of_deviation()"`, that takes a list as input and returns a list that represents how much each element deviates from the mean of the list.

INSPECTION COPY

**COPYRIGHT
PROTECTED**



12. Define a function called "`standard_deviation()`" that returns a list. Feel free to use the functions defined in Questions 10 and 11 and the following steps to find the standard deviation:
- Find the mean
 - List of deviation
 - Squares of the deviation
 - Sum of the squares of deviation
 - Divided by one less than the number of items
 - Square root of this number → use `sqrt()`, which is defined in

Example of how to calculate standard deviation:

Given [1, 4, 5, 7, 9, 20]

Mean $(1 + 4 + 5 + 7 + 9 + 20) / 6 = 8$

List of deviations [-7, -4, -3, 0, 1, 12]

Squares of the deviation [49, 16, 9, 0, 1, 144]

Sum of the squares of the deviation = 214

Divided by one less than the number of items in the list $214 / 5 = 42.8$

Now the square root of 42.8 = 6.54

Standard deviation is about 6.54

**COPYRIGHT
PROTECTED**



Chapter 7: Working with Files

What we will learn:

- ✓ How to open files in Python
- ✓ Syntax for manipulating files
- ✓ Creating a database
- ✓ Making queries

What you need to know:

- ✓ Data types
- ✓ Lists



File

Absolute path

Relative path

Database

Query

Files are used to store data on a secondary storage device. All the data that disappears when we stop running the program. In order to have store it on a secondary storage medium. This data can be used after the run whenever we wish to do so.

In Python we use the `open()` function, which is a built-in function, to open

Syntax:

```
<variable_name> = open(<filename>, <mode>)
```

`<variable_name>` is a file object and will be used to refer to the

`<filename>` is the name of the file and extension. If the file is stored the running program then the name of the file is enough, otherwise which can be relative or absolute, we will discuss these later.

`<mode>` refers to the possible operations on the opened file. The summarised in the table below.

Mode	Meaning
'r'	Open for reading (default)
'w'	Open for writing. If the file exists, Python will delete all the does not exist, Python will create a new file
'x'	Create a new file and open it for writing
'a'	Open for writing, appending to the end of the file if it exist
'b'	Binary mode
't'	Text mode (default)
'+'	Open a disk file for updating (reading and writing)
'U'	Universal newline mode (for backward compatibility)

Table 4

Example:

```
#Open the file in write mode
text_file = open("olympic_games.txt", "w")
```

This will create a text file called `olympic_games.txt` in the current work exists it will truncate (delete all the contents) the file and open it ready to be the file handler stream created in the program, that we will use to refer to

INSPECTION COPY

COPYRIGHT
PROTECTED



7.1 Writing to a File

We use the `write()` function to write contents to a file.

```
print("Writing to a text file with the write() method")

#Open the file in write mode
text_file = open("olympic_games.txt", "w")

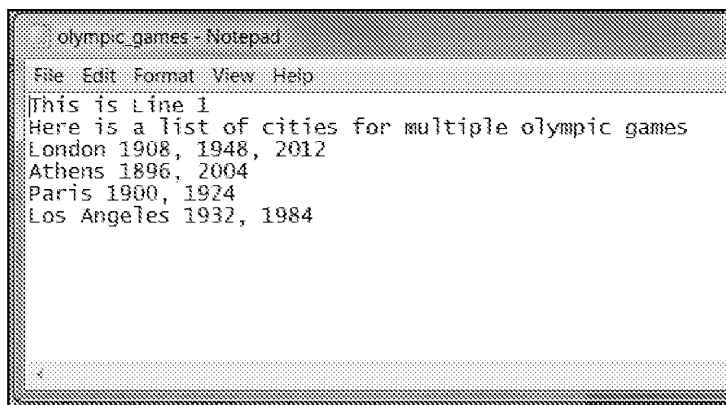
text_file.write("This is Line 1\n")
text_file.write("Here is a list of cities for multiple olympic games\n")
text_file.write("London 1908, 1948, 2012\n")
text_file.write("Athens 1896, 2004\n")
text_file.write("Paris 1900, 1924\n")
text_file.write("Los Angeles 1932, 1984\n")

text_file.close()
```

The code snippet above will create a new file, or open an existing file, called `olympic_games.txt`, write some information to the file then close the file. A few things are worth noting:

- `"w"` mode will either create a new file or open an existing one and overwrite its contents
- `write()` invoked on the file handler will write the file
- `\n` is an escape sequence used to create new lines in text files
- `close()` invoked on the file handler will close the current opened file. It is good practice to close files when you are no longer using them

The following text file can be viewed in the current working directory.



Notice that the name of the file is `olympic_games`, which is the name supplied in the code snippet above. The `\n` inserts a new line in the document.

**COPYRIGHT
PROTECTED**



7.2 Printing a File to the Screen

We use the `read()` function to read data from a file that is open in read mode and print data in the file handler to the screen.

```
#now open the file and print to the screen
text_file = open("olympic_games.txt", "r")
print(text_file.read())

text_file.close()
```

The snippet above will first close the open stream, then reopen the stream and then use the `read()` function to read the stream and print the contents to the screen. Finally, we close the stream.

The entire script and output is repeated below.

The Script

```
# Creating a file and writing to it using the write() method
print("Writing to a text file with the write() method")

#Open the file in write mode
text_file = open("olympic_games.txt", "w")

text_file.write("This is Line 1\n")
text_file.write("Here is a list of cities for multiple olympic games\n")
text_file.write("London 1908, 1948, 2012 \n")
text_file.write("Athens 1896, 2004 \n")
text_file.write("Paris 1900, 1924 \n")
text_file.write("Los Angeles 1932, 1984 \n")

text_file.close()

#now open the file and print to the screen
text_file = open("olympic_games.txt", "r")
print(text_file.read())

text_file.close()
```

The Output

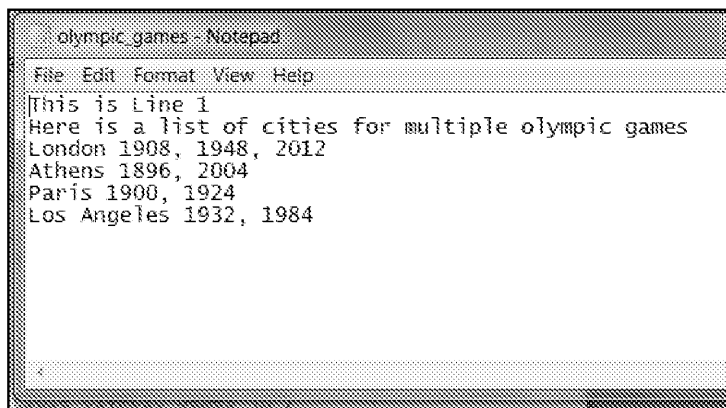
```
>>> ===== RESTART =====
>>>
Writing to a text file with the write() method
This is Line 1
Here is a list of cities for multiple olympic games
London 1908, 1948, 2012
Athens 1896, 2004
Paris 1900, 1924
Los Angeles 1932, 1984
```

**COPYRIGHT
PROTECTED**



The Text File

This text file will be created and saved in the current working directory.



Note that text files take text (string) as input and, therefore, in order to write to a file they will first have to be converted to string. The `str()` function becomes

7.3 Appending a File

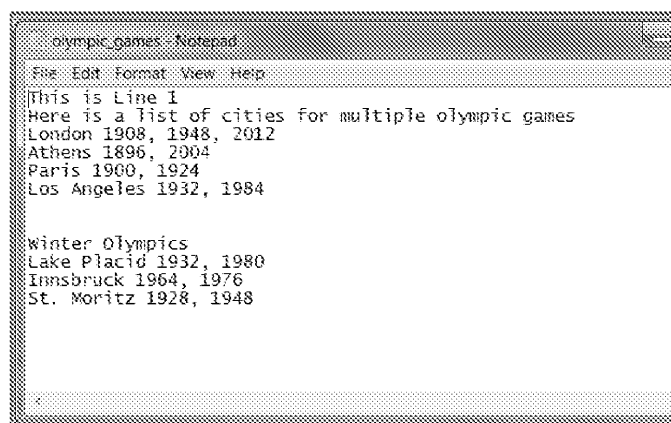
To add data to a file without deleting what is already there, we open the file

```
#First open the file in append mode
text_file = open("olympic_games.txt", "a")

# Now we add the winter games host to the file
text_file.write("\n\nWinter Olympics\n")
text_file.write("Lake Placid 1932, 1980\n")
text_file.write("Innsbruck 1964, 1976\n")
text_file.write("St. Moritz 1928, 1948\n")

#Now close the file
text_file.close()
```

The code above will open the file in append mode and add the winter Olympics more than once. Again, we close the file after using it. We can test the contents to the screen. We should see both sets of information in the file. On our computer we should see the updated text file as seen below.



INSPECTION COPY

**COPYRIGHT
PROTECTED**



7.4 File Path

A **directory structure** (folder structure) is the way in which an operating system organizes files for its user. The Windows operating system uses drive names (typically identified by a letter) to identify a directory or folder. The directory separator is the "\" (backslash), while the Linux operating system uses the "/" (forward slash) as its directory separator. A file name is therefore identified by its address. There are two ways of stating a file name using the address: relative path and absolute path.

In the previous section, we used a relative file path to open files. Relative file paths are relative to the current directory; therefore by running the code, Python will create or open files in the current working directory.

```
>>> new_file = open("sample.txt", "w")
```

The **current working directory** is the folder that the running program is operating in. We can find the current working directory by using the function `getcwd()`. The function `getcwd()` is part of the `os` module and therefore we will have to import this module before we can use the function. Let's print the path of the current working directory:

```
>>> import os
>>> print(os.getcwd())
C:\Users\Devz\PythonExercise
>>>
```

To create the file at some other location, we could use the absolute file path address, for example:

```
>>> new_file = open("C:\\Users\\Devz\\PythonExercise\\sample.txt", "w")
>>>
```

This will open or create a file called `sample.txt` in the specified location `C:\Users\Devz\PythonExercise\sample.txt`. It is worth mentioning that in the command above we use double backslash to escape the backslash. The reason for this is that "\" denotes an escape sequence in Python, therefore we need to use double backslash. Alternatively, we could proceed the file path with "r". File paths starting with "r" are called **absolute file paths**; the difference is that the address is given from the root of the drive.

```
>>> new_file = open(r"C:\Users\Devz\PythonExercise\sample.txt", "w")
>>>
```

**COPYRIGHT
PROTECTED**



7.5 Working with Databases

A database is a collection of tables that store data. The software that operates a database management system (DBMS). Most databases will contain more than one table; each table is identified by a unique name. Almost all organisations use a database at some point. It is therefore important for us to write programs that can create a database and update the database. The most common operation carried out on a database is a Query in its strictest sense is asking the database a question and the database returns the data items that meet the criteria. Before we discuss queries, it is important to understand some database terminologies. For the illustration in the table below, we will use a database of students similar to what is used in schools.

Name	Meaning	
Table	A collection of data relevant to one entity	A student
Records	Collection of all the data on a single entity	001, John
Field	Single column of data in a table	John Benjamin Alex Ahmed Crystal Simon
Field Name	The name given to the field	First Name

Table 5

Here is an example of a table within a database with the sections labelled.

Name of Table: Students

Field				
student_id	student_firstname	student_surname	student_DOB	For
001	John	Doe	12/03/1992	7S
002	Benjamin	Harsh	04/05/1991	7D
003	Alex	Cummings	11/02/1992	7S
004	Ahmed	Ahmed	13/06/1991	7C
005	Crystal	Jones	05/08/1992	7S
006	Simon	Daily	03/01/1991	7S
007	James	Williams	04/02/1992	7D
008	Charlene	Parchment	17/08/1991	7F

Each field in the data has a data type. The data types are similar to the ones used in a universal language used to create and manipulate databases is called structured query language (SQL pronounced sequel).

Some DBMS, such as Microsoft Access, will generate the SQL codes for the databases in Python, it is worth understanding some SQL commands, we will

**COPYRIGHT
PROTECTED**



7.6 SQL

We will discuss the following statements available in SQL:

- CREATE TABLE – *this will create a table in a database*
- SELECT – *display data that is in a table in a database*
- UPDATE – *change data in a table in a database*
- DELETE – *remove data from a table in a database*
- INSERT INTO – *insert new data into a table in a database*

7.6.1 The CREATE TABLE Statement

We first use the CREATE TABLE statement to create the table that we will be using.

Syntax:

```
CREATE TABLE <table_name>
(
  <field_name> <data_type>,
  <field_name> <data_type>,
  <field_name> <data_type>
)
```

<table_name> is the name of the table to be created in the database.

<field_name> is the name of the column to be created in the table.

<data_type> is the data type of the column.

Example:

```
CREATE TABLE Students
(
  student_id text,
  student_firstname text,
  student_surname text,
  student_DOB date,
  Form text
)
```

This will create the following table called “Students” in the database

student_id	student_firstname	student_surname	student_DOB	Form

For the rest of the SQL section, we will use the following table as our running example.

student_id	student_firstname	student_surname	student_DOB	Form
001	John	Doe	12/03/1992	7S
002	Benjamin	Harsh	04/05/1991	7D
003	Alex	Cummings	11/02/1992	7S
004	Ahmed	Ahmed	13/06/1991	7C
005	Crystal	Jones	05/08/1992	7S
006	Simon	Daily	03/01/1991	7S
007	James	Williams	04/02/1992	7D
008	Charlene	Parchment	17/08/1991	7F

**COPYRIGHT
PROTECTED**



7.6.2 The SELECT Statement

The SELECT statement is used to display data in a table. We can view all the data, or we can view data based on a specific criterion, in which case only data that meets the criteria will be displayed.

Syntax:

```
SELECT <field_name> FROM <table_name>
```

<field_name> is the name of the column to be selected.

<table_name> is the name of the table in the database.

Example:

```
SELECT student_firstname FROM Students
```

This will produce the following output:

student_firstname
John
Benjamin
Alex
Ahmed
Crystal
Simon
James
Charlene

```
SELECT student_firstname, student_surname FROM Students
```

This will produce the following output:

student_firstname	student_surname
John	Doe
Benjamin	Harsh
Alex	Cummings
Ahmed	Ahmed
Crystal	Jones
Simon	Dally
James	Williams
Charlene	Parchment

The wildcard (*) is a quick way of selecting all columns in the table and, therefore, the entire table.

```
SELECT * FROM Students
```

This will produce the entire table

student_id	student_firstname	student_surname	student_DOB	Form
001	John	Doe	12/03/1992	7S
002	Benjamin	Harsh	04/05/1991	7D
003	Alex	Cummings	11/02/1992	7S
004	Ahmed	Ahmed	13/06/1991	7C
005	Crystal	Jones	05/08/1992	7S
006	Simon	Dally	03/01/1991	7S
007	James	Williams	04/02/1992	7D
008	Charlene	Parchment	17/08/1991	7F

**COPYRIGHT
PROTECTED**



7.6 The WHERE Clause

The "Where" clause is used in conjunction with the select statement; this is the selection.

Syntax:

```
SELECT <field_name> FROM <table_name>
WHERE <field_name> operator <value>
```

<field_name> is the name of the column to be selected.

<table_name> is the name of the table in the database.

operator is any operator from the table below.

<value> is any legal value that can be stored in the field name selected.

Operator	Meaning
=	Equal to
<> or !=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
BETWEEN	Value between a given range including the bounds
LIKE	Matches a given pattern
IN	To find more than one value in a column

Table 6

Example:

```
SELECT * FROM Students
WHERE Form='7S'
```

This will produce

student_id	student_firstname	student_surname	student_DOB	Form
001	John	Doe	12/03/1992	7S
003	Alex	Cummings	11/02/1992	7S
005	Crystal	Jones	05/08/1992	7S
006	Simon	Daily	03/01/1991	7S

Notice that '7S' requires single quotation marks as the data type is text; the same applies to numeric data if the data type is numeric.

**COPYRIGHT
PROTECTED**



7.6.4 The UPDATE Statement

This statement is used to change records by updating them in a table in a database.

Syntax:

```
UPDATE <table_name>
SET <column1> = value1, <column2>=value2, ...
WHERE <field_name> = value
```

<table_name> is the name of the table in the database.

<column1> is the first column to be updated.

<column2> is the second column to be updated.

<field_name> specifies the record to be updated.

Note that the WHERE clause specifies the particular record to be updated. All records will be updated.

Example:

```
UPDATE Students
SET Form = '7S'
WHERE student_firstname='Charlene'
```

Before the update:

student_id	student_firstname	student_surname	student_DOB	Form
001	John	Doe	12/03/1992	7S
002	Benjamin	Harsh	04/05/1991	7D
003	Alex	Cummings	11/02/1992	7S
004	Ahmed	Ahmed	13/06/1991	7C
005	Crystal	Jones	05/08/1992	7S
006	Simon	Dally	03/01/1991	7S
007	James	Williams	04/02/1992	7D
008	Charlene	Parchment	17/08/1991	7F

After the update:

student_id	student_firstname	student_surname	student_DOB	Form
001	John	Doe	12/03/1992	7S
002	Benjamin	Harsh	04/05/1991	7D
003	Alex	Cummings	11/02/1992	7S
004	Ahmed	Ahmed	13/06/1991	7C
005	Crystal	Jones	05/08/1992	7S
006	Simon	Dally	03/01/1991	7S
007	James	Williams	04/02/1992	7D
008	Charlene	Parchment	17/08/1991	7S

Notice that the Charlene is now in form 7S.

**COPYRIGHT
PROTECTED**



7.6.5 The DELETE Statement

This statement is used to remove records from a database.

Syntax:

```
DELETE FROM <table_name>
WHERE <field_name> = value
```

<table_name> is the name of the table in the database.

<field_name> = value specifies the record to be deleted.

Example:

```
DELETE FROM Students
WHERE student_firstname='Charlene'
```

Before the delete statement:

student_id	student_firstname	student_surname	student_DOB	Form
001	John	Doe	12/03/1992	7S
002	Benjamin	Harsh	04/05/1991	7D
003	Alex	Cummings	11/02/1992	7S
004	Ahmed	Ahmed	13/06/1991	7C
005	Crystal	Jones	05/08/1992	7S
006	Simon	Dally	03/01/1991	7S
007	James	Williams	04/02/1992	7D
008	Charlene	Parchment	17/08/1991	7F

After the delete statement:

student_id	student_firstname	student_surname	student_DOB	Form
001	John	Doe	12/03/1992	7S
002	Benjamin	Harsh	04/05/1991	7D
003	Alex	Cummings	11/02/1992	7S
004	Ahmed	Ahmed	13/06/1991	7C
005	Crystal	Jones	05/08/1992	7S
006	Simon	Dally	03/01/1991	7S
007	James	Williams	04/02/1992	7D

**COPYRIGHT
PROTECTED**



7.6.6 The INSERT INTO Statement

The insert into statement is used to insert a new record into a database.

Syntax:

```
INSERT INTO <table_name> (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...)
```

<table_name> is the name of the table in the database.

column is the column into which the value should be inserted.

value is the value to be inserted into the matching column.

Note that the column is optional, but SQL will insert the values into consecutive columns if any are omitted. Specifying the column is a good way to insert data into non-consecutive columns.

Example:

```
INSERT INTO Students
VALUES ('009', 'Paul', 'Piggot', '15/07/1992', '7C')
```

Before the insert into statement:

student_id	student_firstname	student_surname	student_DOB	Form
001	John	Doe	12/03/1992	7S
002	Benjamin	Harsh	04/05/1991	7D
003	Alex	Cummings	11/02/1992	7S
004	Ahmed	Ahmed	13/06/1991	7C
005	Crystal	Jones	05/08/1992	7S
006	Simon	Daily	03/01/1991	7S
007	James	Williams	04/02/1992	7D
008	Charlene	Parchment	17/08/1991	7F

After the insert into statement:

student_id	student_firstname	student_surname	student_DOB	Form
001	John	Doe	12/03/1992	7S
002	Benjamin	Harsh	04/05/1991	7D
003	Alex	Cummings	11/02/1992	7S
004	Ahmed	Ahmed	13/06/1991	7C
005	Crystal	Jones	05/08/1992	7S
006	Simon	Daily	03/01/1991	7S
007	James	Williams	04/02/1992	7D
008	Charlene	Parchment	17/08/1991	7S
009	Paul	Piggot	15/07/1992	7C

We could have used the field name to specify the columns where we want to insert. In this case we want to insert it into all columns and therefore it is not necessary.

**COPYRIGHT
PROTECTED**



7.7 SQL and Python

Python provides the `SQLite3` library to manipulate lightweight disk databases. To use a database, we first have to import the `SQLite3` library. We use the key demonstrated below.

```
#we first import the sqlite3 library

import sqlite3
```

Connect

Now that the `SQLite3` library is imported, we can connect to and use a database. To use a database we need to have a handler in our program that interacts with a disk or in the RAM. We use the `connect()` function, as defined in the `SQLite3` library, to open a connection to the `SQLite` database file. The following code snippet is called "School".

```
#we first import the sqlite3 library

import sqlite3

#Create a handler called new_db that connects to database
new_db = sqlite3.connect('C:\\Users\\Devz\\PythonEx\\School.db')
```

The argument to the function is the file path for the database; notice that in our file path as single backslash denotes an escape sequence.

Create Cursor Object

In Python, the cursor object allows us to work with and manipulate databases. To use a cursor, we call the cursor method on the connection object. In our example, the code will create a new cursor called `c`.

```
#we first import the sqlite3 library

import sqlite3

#Create a handler called new_db that connects to database
new_db = sqlite3.connect('C:\\Users\\Devz\\PythonEx\\School.db')

#Create a new cursor object to manipulate the database
c=new_db.cursor()
```

Now that we have a connection to our database and a cursor object, we can create our table and insert data into our tables. We will create a table called `properties` as before.

**COPYRIGHT
PROTECTED**



Create Table

We first create a table.

```
#Now create a table called Students
c.execute('''CREATE TABLE Students
(student_id text,
student_firstname text,
student_surname text,
student_DOB date,
Form text)
''')
```

Notice that we are using SQL commands to create the table.

Use the INSERT INTO statement to populate the table.

Next, we use the INSERT INTO statement to insert values into our table.

```
#insert data into our table
c.execute('''INSERT INTO Students
VALUES ('001', 'John', 'Doe', '12/03/1990', 'Form 1')''')
```

Notice that we are using the version where we do not specify the field name information into all columns.

Commit and Close

We use the `commit()` statement to save/commit the final transaction and

```
#Save changes using the commit() function
new_db.commit()
```

```
#Close the connection to the database
new_db.close()
```

**COPYRIGHT
PROTECTED**



Putting all the codes together:

```
#we first import the sqlite3 library

import sqlite3

#Create a handler called new_db that connects to db
new_db = sqlite3.connect('C:\\Users\\Devz\\PythonExercise\\school.db')

#Create a new cursor object to manipulate the database
c=new_db.cursor()

#Now create a table called Students
c.execute('''CREATE TABLE Students
(student_id text,
student_firstname text,
student_surname text,
student_DOB date,
Form text)
''')

#insert data into our table
c.execute('''INSERT INTO Students
VALUES ('001', 'John', 'Doe', '12/03/1992', '7')''')

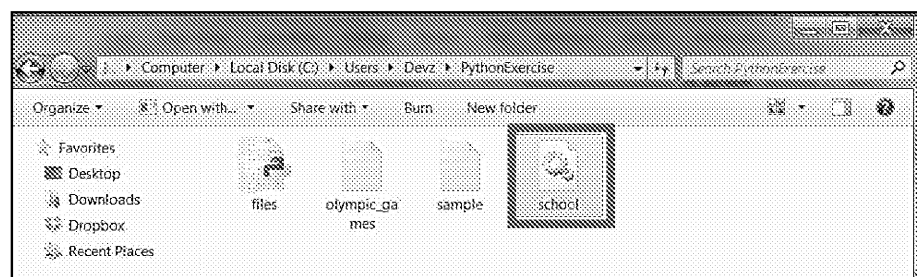
#Save changes using the commit() function
new_db.commit()

#Close the connection to the database
new_db.close()
```

After executing this code, the database called school.db will be created in the folder (C:\\Users\\Devz\\PythonExercise\\school.db), with a table called "Students" and the following data:

student_id	student_firstname	student_surname	student_DOB	Form
001	John	Doe	12/03/1992	7

The physical files are shown below. Note that this file is not readable by notepad, and therefore if we open this file we will only see symbols.



We can, however, use Python to read the data from this file and print this data. The following code snippet reconnects to the database, reads the file and prints the data.

**COPYRIGHT
PROTECTED**



```
#Re-connect to the database
new_db=sqlite3.connect('C:\\Users\\Devz\\PythonExe

#Create a cursor object
c=new_db.cursor()

#Use the SELECT statement with wildcard to select
c.execute("SELECT * FROM Students")

#use the fetchone function to fetch one row of data
row=c.fetchone()

#print the row to the screen
print(row)

#Finally close the connection
new_db.close()
```

The result of executing this code is:

```
>>> ===== RESTART =====
>>>
('001', 'John', 'Doe', '12/03/1992', '7S')
>>>
```

It is always good practice to close the database connection once you have finished. If you do not, other processes will not be able to access the opened database.

Here are some other functions defined on the object cursor and a description of what they do:

executemany(...)

Repeatedly execute a SQL statement

executescript(...)

Executes multiple SQL statements at once

fetchall(...)

Fetches all rows from the result set

fetchmany(...)

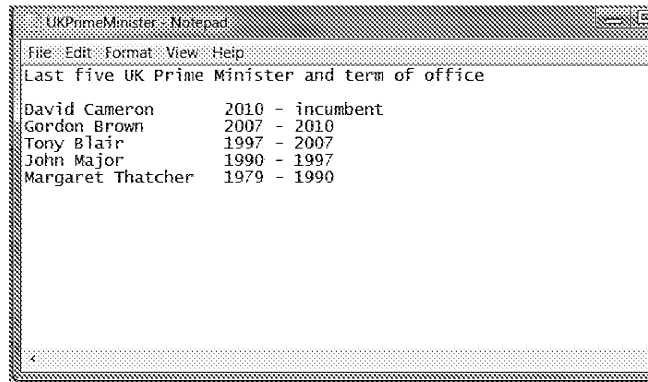
Fetches several rows from the result set

**COPYRIGHT
PROTECTED**



Practice Exercise 7

1. a. Create a text file in the current working directory that creates a file called "UKPrimeMinister.txt" which shows a list of the last five (5) Prime Ministers of the United Kingdom.



- b. Now write a Python program that will read the file called "UKPrimeMinister.txt" that was created above, and print the content to the screen. Then run the program.

```
>>> ===== RESTART =====
>>>
Writing text file
Last five UK Prime Minister and term of office

David Cameron      2010 - incumbent
Gordon Brown       2007 - 2010
Tony Blair         1997 - 2007
John Major         1990 - 1997
Margaret Thatcher  1979 - 1990

>>>
```

2. The code below should create a text file called "my_quote.txt". The function `update_file()` is defined to take a file name and a string containing a quote. The function should open the file called "my_quote.txt" and update it with a new quote. The program should prompt the user to enter a quote three times. Then print the content of the file. At the moment there are two things wrong with the code.
 - a. Only the last quote is saved in the file
 - b. The user is being prompted four times instead of three

Examine the code and correct it to get the desired outcome.

Desired Outcome:

```
>>> ===== RESTART =====
>>>
Enter your favourite quote: Education is the most powerful weapon which you can use to
Enter your favourite quote: Education is not preparation for life it is life itself,
Enter your favourite quote: The only thing that interferes with my learning is my education
This is an update
Education is the most powerful weapon which you can use to chang the world, Nelson Ma

This is an update
Education is not preparation for life it is life itself, John Dewey

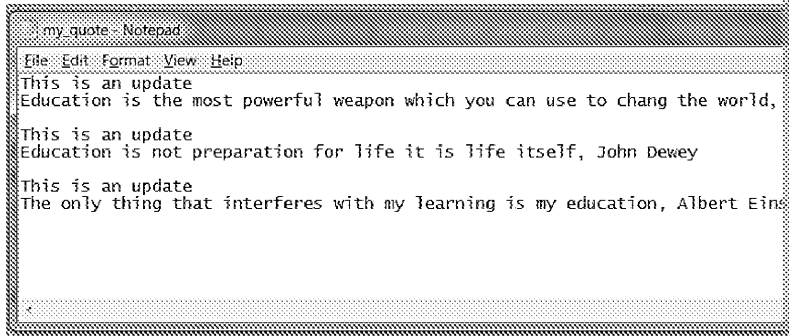
This is an update
The only thing that interferes with my learning is my education, Albert Einstein

>>>
```

INSPECTION COPY

**COPYRIGHT
PROTECTED**



Text File:**Code to be Corrected:**

```
# update three quotes to a file

file_name = "my_quote.txt"
#create a file called my_quote.txt
new_file = open(file_name, 'w')
new_file.close()

def update_file(file_name, quote):
    #First open the file
    new_file = open(file_name, 'w')
    new_file.write("This is an update\n")
    new_file.write(quote)
    new_file.write("\n\n")

    #now close the file
    new_file.close()

for index in range(1,3):
    quote = input("Enter your favourite quote: ")
    update_file(file_name, quote)

# Now print the contents to the screen
new_file = open(file_name, 'r')
print(new_file.read())

# And finally close the file
new_file.close()
```

3. Complete the following code to create a database called "Library.db" "Books". The code should then populate the table with the following

book_isbn	book_title	book_type	book_author
978-0-340-88851-3	A2 Pure Mathematics	Non fictional	Cather
978-1-118-10227-5	Android 4 Application Development	Non fictional	Reto M
0-596-00699-3	Programming C#	Non fictional	Jesse L

It should then print the snippet overleaf back to the screen.

**COPYRIGHT
PROTECTED**



Incomplete Code

```

#we first import the sqlite3 library
import sqlite3

#insert the correct path here
new_db = sqlite3.connect('C:')

#Create a new cursor object to manipulate the database
c=new_db.cursor()

#Now create a table called Students
c.execute('''CREATE TABLE Books
(book_isbn text,
book_title text,
book_type text,
book_author text,
publisher text)
''')
#insert Statements here to add data to the table

new_db.commit()

#Close the connection to the database
new_db.close()

#Re-connect to the database - insert the correct path
new_db=sqlite3.connect('C:')

#Create a cursor object
c=new_db.cursor()

#Use the SELECT statement to select all the data

#use the fetchone function to fetch one row of data
book_library=c.fetchall()

#print the row to the screen
for book in book_library:
    print(book)

#Finally close the connection
new_db.close()

```

4. Explain the effect of running the following code on a database.

a. *SELECT Product.Name, Product.Quantity, Product*

.....

b. *FROM Product*

.....

c. *WHERE Product.Quantity > 20*

.....

**COPYRIGHT
PROTECTED**



Chapter 8: Creating Your Own Type

What we will learn:

- ✓ Object-oriented programming
- ✓ What is a class
- ✓ How to create a class
- ✓ Assigning values to a class

What you need to know:

- ✓ Data types
- ✓ Methods



Object-oriented

Object

Attributes

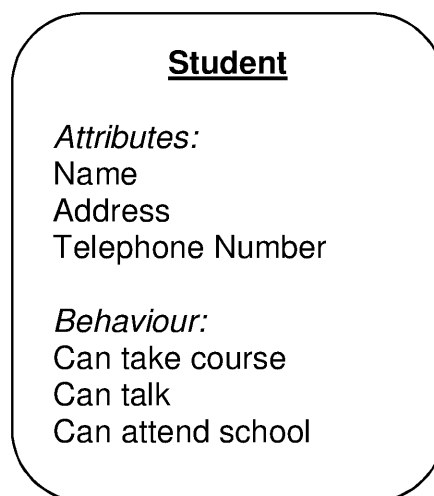
Class

Abstract

8.1 Object-Oriented Programming (OOP)

We had a thorough discussion about data types in Chapter 3. It turns out that real life will require other data types that are not supplied by Python. Luckily, Python lets you create your own data type. New data types created are often referred to as classes and are called object-oriented programming. In the Introduction, when we introduced Python, we said Python was a hybrid language and supported a multiple programming paradigm (in addition to about computation); it turns out that the ability to create classes and manipulate them is to be an object-oriented programming language. As an example, let us consider a student management system used at a school or university. It would be a great idea if we could represent a student as an object. We were to understand that we are talking about a person who has a name, address, and is registered for certain courses, and can do things such as walk, run, take an exam, etc. We find out that these entities (students in this case) are referred to as objects.

An **object** is a thing or an event in our application. Objects can have data information called **attributes**. Objects also have behaviours which are called **methods**. Earlier we had classified a student as an object. The diagram below shows the student object and its attributes and behaviours:



INSPECTION COPY

COPYRIGHT
PROTECTED



8.2 Why Do We Use Classes?

- 1) *To bring the solution space closer to the problem space.* It is more natural to refer to the name of a student than just referring to the string name. Computer programmers solve real-world problems. Where the problem occurs is referred to as the problem space. The solution space in our example above is a school and, in particular, in a course room. When we are using the computer to solve this problem, then the solution space is the computer. Classes will give the programmer the ability to relate real-life entities (students) to the computer, bringing the solution space closer to the problem space.
- 2) *To develop so-called abstract data types (ADT).* Abstract data types are used to define numerous operations on. For example, if we develop a new ADT for a car, we can define interesting methods, such as `draw_self()`, or `colour_self()` which will draw the car and colour itself.
- 3) *For reuse of code.* If we have a definition of a good ADT, then we can reuse it. We can use the methods that are already defined. Let us say we developed the car ADT mentioned above, and now we want to develop an extra-curricular club for a school. If we have information then there is no need to redefine the student ADT that was developed before. Furthermore, other programmers can use ADT that was developed by others.
- 4) *To enforce data encapsulation.* Data encapsulation is information hiding. In classes, a programmer can use a new ADT without knowing the underlying details. This allows programmers to develop codes that other programmers can use without knowing that are defined over these ADTs, without worrying about the intricacies of the implementation. This is accomplished. This also gives the unique advantage, where if a new way of doing something comes about then the writer of the ADT can make this change and the users of the ADT will not struggle. We have seen this modular design in engineering and in the real world. A driver of a car is not necessarily concerned about how the pedal uses a master cylinder which in turn applies the disk pads to the wheel to stop the car. The driver is concerned with the fact that when he pushes on the brake pedal, this will stop the car. The designer of the car can choose whether to use drum brakes or disc brakes or another braking system that is not discovered as yet; this will not affect the driver. In a similar manner, if a more efficient data structure is discovered, the designer of the ADT can implement this and all programmers benefit without being affected.

8.3 Data Types in Python

We have seen objects before. All data types in Python are objects. The implementation of a class is called a class.

```
>>> name = 'Patrick'
>>> print(type(name))
<class 'str'>
>>>
>>> number = 45
>>> print(type(number))
<class 'int'>
>>>
>>> z = [1,2,3]
>>> print(type(z))
<class 'list'>
>>>
```

**COPYRIGHT
PROTECTED**



From the code snippet on the previous page, we assign "Patrick" to name, then we see get <class 'str'>. This means that "name" is an instance of a str which is an instance of the integer class and "z" is an instance of the list class.

We have encountered string (a collection of characters) before. We also encountered methods defined on string. Upper() and lower() are two such methods. It turns out that many methods are defined on them; this is the case because in the definition of the string class, lower() is defined.

```
>>> month = 'September'
>>> print(month.upper())
SEPTEMBER
>>> #now print the lower case of month
>>> print(month.lower())
september
>>>
```

To view all the methods that are defined on an object, we use the dir() method.

```
>>> dir(month)
['_add_', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__hash__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rstrip', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'title', 'translate', 'upper', 'zfill']
>>>
```

Notice that all of these methods are defined on all strings.

8.4 Defining a Class

We use the class keyword followed by the name to define a class.

Syntax:

```
class <class_name>:
    <class_docstring>
    <class_suite>
```

<class_name> is the name of the class following the same conventions as variable names in Chapter 1.

<class_docstring> or documentation string explains what the class is and provides the user of the class enough information about the class being defined.

<class_suite> is the statement that defines the class. This will contain the code that defines the class and are invoked whenever we have instances of the class.

**COPYRIGHT
PROTECTED**



Example:

```

class Student:
    """ The Student class defines a student with
    attributes: name, age, form
    """
    #Attributes
    name = "not set"
    age = 1
    form = "7"

    #Functions
    def get_name(self):
        return self.name

    def get_age(self):
        return self.age

    def get_form(self):
        return self.form

    def set_name(self, new_name):
        self.name = new_name

    def set_age(self, new_age):
        self.age=new_age

    def set_form(self, new_form):
        self.form = new_form

    def say_hello(self):
        print("hello my name is", self.name, " I'm in

```

The code snippet above will define the student class. The different sections

The class and docstring:

```

class Student:
    """ The Student class defines a student with
    attributes: name, age, form
    """

```

The keyword class is used, followed by the name of the class and colon.

The docstring or documentation string is used to build the documentation using the class. This information also appears when we use the help facility student class.

The attributes:

```

#Attributes
name = "not set"
age = 1
form = "7"

```

**COPYRIGHT
PROTECTED**



This is the section to list the attributes on the class and initialise the default

The class functions:

```
#Functions
def get_name(self):
    return self.name

def get_age(self):
    return self.age

def get_form(self):
    return self.form

def set_name(self, new_name):
    self.name = new_name

def set_age(self, new_age):
    self.age=new_age

def set_form(self, new_form):
    self.form = new_form

def say_hello(self):
    print("hello my name is", self.name, " I'm in
```

The class functions are used to access and change the attributes of the class self. It turns out that we use self with the dot operator to refer to attributes of the class.

8.4.1 Using the Class

When we create a class, this is called an instance of the class. The following code creates an instance of student1 and use some of its functions.

```
#create a new Student
student1 = Student()

#Set the name of Student1 to John Doe
student1.set_name("John Doe")

#Use the accessor method to print the name
print(student1.get_name())

#Say hello from student1
print(student1.say_hello())
```

A few things to note above:

- We declare a variable and assign the name of the class as if it were a function.
- Once we create an instance of the class we can now use the class methods.

**COPYRIGHT
PROTECTED**



This is the output that we would get.

```
>>> ===== RESTART =====  
>>>  
John Doe  
hello my name is John Doe I'm in form 7  
None  
>>>
```

Notice that if we create a new instance of the student class and say "hello" message.

```
>>> student2 = Student()  
>>> print(student2.say_hello())  
hello my name is not set I'm in form 7  
None  
>>>
```

When we say "hello", the default values are displayed.

**COPYRIGHT
PROTECTED**



Practice Exercise 8

1. Create a class called “Time” with the following attributes: hour, min should initialise to (12:00:00), unless the user supplies another time with the following names and functionalities.
 - a. `get_hour()` – return the current hour
 - b. `get_minute()` – return the current minute
 - c. `get_second()` – return the current second
 - d. `print_time()` – print the current value of time

And the following mutator functions:

- e. `set_hour()` – takes a new hour and sets this as hour
 - f. `set_minute()` – takes a new minute and sets this as the min
 - g. `set_seconds()` – take a new second and sets this as the sec
 - h. `increment_second()` – increment second by one second, if incrementing will increment minute by one and turn seconds to 0
 - i. `increment_minute()` – increment minute by one, note that incrementing minute will result in incrementing hour by one.
 - j. `increment_hour()` – will result in incrementing hour by one and return to 1.
2. Write a Python program that inserts the time class and create two instances, one with the default time and the other a user-defined time. Use each of the mutator functions, printing the values before and after the mutator function.
3. Create a class called “Fraction”, with the following attributes: numerator, denominator. These should initialise to (0/1), unless the user supplies another fraction. Implement the following functions called:
 - a. `get_numerator()` – return the numerator
 - b. `get_denominator()` – return the denominator
 - c. `print_fraction()` – prints the fraction
 - d. `print_type()` – print the type of fraction (“proper” if the denominator is greater than the numerator, “improper” if the numerator is larger than the denominator)

And the following mutator function

- e. `set_numerator()` – takes a new numerator and set this as numerator
 - f. `set_denominator()` – takes a new denominator and set this as denominator
 - g. `inverse()` – change the numerator to the denominator and vice versa, also change if the numerator is 0
4. Write a Python program that inserts the “Fraction” class and create two instances, one with the default fraction and the other a user-defined fraction. Use each of the mutator functions, printing the values before and after the mutator function.

INSPECTION COPY

**COPYRIGHT
PROTECTED**



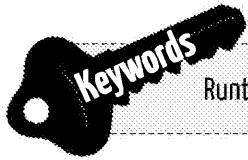
Chapter 9: Dealing with Errors

What we will learn:

- ✓ How to identify errors
- ✓ Categorising different types of error
- ✓ How to fix different errors
- ✓ Example of errors

What you need to know:

- ✓ Writing simple programs



Runtime errors

Syntax

Semantics

Recursive

Exceptions

9.1 Exceptions and Errors

There are different types of error that can occur when we are writing programs. We will discuss them and examine possible solutions.

Types of Errors

Type of Error	Meaning
Syntax	<p>A syntax error occurs when the programmer uses a token (letter, symbol and operator) somewhere that is not defined in the grammar of the language.</p> <p>Programming languages have rules, similar to other forms of communication. The interpreter takes the source code and parses the tokens, then sends this parsed code to the evaluator to get its meaning.</p>
Runtime	<p>A runtime error occurs during execution of a program. The programmer normally thinks about these but can do nothing about them. They are often dealt with by the exception handler.</p>
Semantic or logical	<p>Semantic refers to the meaning of things. These errors happen when there are errors with the logic of the program, which gives unexpected results. The interpreter will not raise an exception or give an error message for semantic errors, but the user will not get the expected result.</p> <p>In evaluating $\frac{15}{2\pi}$ in Python we would type <code>15/2*math.pi</code>. As division and multiplication have the same precedence we would get 23.56, instead of the correct answer 2.39. In order to get the correct answer we would have to use brackets; therefore, we would type <code>15/(2*math.pi)</code>.</p>

Table 7

INSPECTION COPY

COPYRIGHT
PROTECTED



In general, the term *exception* is used to describe when something goes wrong. It can include errors caused by the programmer and external factors, such as hardware failure. Some programming languages make explicit differentiation between an error and an exception. In Python, the difference is more subtle in that both errors and exceptions inherit from the `BaseException` class. Subclasses are then derived which are either errors or exceptions.

We will now turn our attention to dealing with these kinds of error and then exceptions.

9.2 Syntax Error

Syntax errors are quite easy to fix. The interpreter will raise this error where the error occurred. This is often at the exact location or sometimes in the previous line. For example, if a closed bracket or a colon is missing from the previous line, the syntax error will be highlighted in the following line. The code snippet below demonstrates this.

```
>>> print("I am leaving out the closed bracket here")
SyntaxError: invalid syntax
>>>
```

We will also get a syntax error if we group two operators together, for example:

```
>>> 5 + 4 + *
SyntaxError: invalid syntax
>>>
```

This will generate a syntax error, I'll use Backus–Naur Form to explain why in the example above.

The grammar of almost all programming languages is described in **Backus–Naur Form** (BNF), developed by John Backus in the 1950s; he was very influential in developing the ALGOL programming language. The purpose of BNF is to develop a concise way of describing the grammar of a language. The structure of BNF is:

<non-terminal> → replacement

The **replacement** can be replaced with zero or more non-terminal or terminal symbols. A **Terminal** terminates the statement; once it is used it cannot be replaced.

The grammar for mathematical statements can be expressed by the following BNF:

```
expression → expression operator expression
expression → number
operator → +, -, *, /, %, ...
number → 0, 1, 2, 3, 4, ...
```

You will notice that there are two lines for defining the expression. Together they form a recursive definition. A recursive definition is where we define something in terms of itself. The first line is a recursive definition called the base case. This qualifies the expression as a recursive definition. The expression can be replaced by an expression and also by number, which is a terminal. This is a common concept in computer science and is used to define and break down many problems.

**COPYRIGHT
PROTECTED**



How do we use this grammar?

Suppose we want to generate the expression $12 + 3 * 15$.

We can use the following steps:

1. Start out with an expression	<i>Expression</i>
2. Replace with expression \rightarrow expression	expression <i>operator</i>
3. Replace expression \rightarrow number	number <i>operator</i>
4. Replace number \rightarrow 12	12 <i>operator</i>
5. Replace operator \rightarrow +	12 + <i>expression</i>
6. Replace expression \rightarrow expression operator expression	12 + expression <i>operator</i>
7. Replace expression \rightarrow number	12 + number <i>operator</i>
8. Replace number \rightarrow 3	12 + 3 <i>operator</i>
9. Replace operator \rightarrow *	12 + 3 * <i>expression</i>
10. Replace expression \rightarrow number	12 + 3 * number
11. Replace number \rightarrow 15	12 + 3 * 15

The replaced is highlighted on the right.

Now let us try to generate $5 + 4 + *$.

1. Start out with an expression	<i>Expression</i>
2. Replace with expression \rightarrow expression	expression <i>operator</i>
3. Replace expression \rightarrow number	number <i>operator</i>
4. Replace number \rightarrow 5	5 <i>operator</i>
5. Replace operator \rightarrow +	5 + <i>expression</i>
6. Replace expression \rightarrow exp op exp	5 + expression <i>operator</i>
7. Replace expression \rightarrow number	5 + number <i>operator</i>
8. Replace number \rightarrow 4	5 + 4 <i>operator</i>
9. Replace operator \rightarrow +	5 + 4 + <i>expression</i>
10. Error we cannot replace expression \rightarrow operator	5 + 4 + Syntax

It turns out that it is impossible to generate $5 + 4 + *$ using the grammar. The interpreter will generate a syntax error when to try to parse the expression.

To reduce or eliminate syntax errors:

- Ensure that operators are surrounded by operands (Python uses whitespace) and therefore, the operator must be in the middle of the operands.
- Ensure that all opening brackets (, {, [have a matching closing bracket (, },]
- Ensure that all opening quotation marks (" or ') have a matching closing quotation mark.
- Ensure that you include a colon (:) at the end of your IF statements.
- Ensure that you are using the correct operator, for example, = means assignment.
- Ensure that you are not trying to use a keyword as a variable name.
- Ensure that you are using the correct indentation. Python is indented after a colon (:) and return to the original alignment afterwards.

**COPYRIGHT
PROTECTED**



9.3 Runtime Errors

Runtime errors are more difficult to spot, largely because the program will run to the user which gives the impression that everything is working well. Run the following code in a few ways:

- The program runs without an output. If this is the case, then check the output statement in the code, unless the code is a module that will not output.
- If the program hangs and become inactive. This normally happens in an indeterminate loop, without statements to make the condition false. For example, in the while loop. If this happens, the program will run forever. We will illustrate this.

```
number = 1
while number < 10:
    print("The value of number is ", number)
    # notice that there is no statement to make the condition false
```

Notice that in the body of the while loop there is no statement that will change the value of number, and therefore the number will always be less than 10, and the condition will always be true. If this code snippet was in your code, the error would be a runtime error. There is no statement in the body of the while loop that will be printing "The value of number is ". Therefore, we could identify where the error is. Now if there was a statement in the body of the while loop that would be updating a database or traversing a list, the error would become more difficult to identify.

- Runtime errors can happen when we have a recursive call that does not have a base case. Let us think about the recursive definition of a function. We defined expression as follows:

$$\text{expression} \rightarrow \text{expression operator expression}$$

$$\text{expression} \rightarrow \text{number}$$

If the second statement were absent ($\text{expression} \rightarrow \text{number}$), then the program would never get to a situation where we have expressions in all terminals (numbers).

Let us use a recursive definition to define a function called `power_number` that takes two values as input, namely a base and an index.

```
def power_number(base, index):
    if index == 0:
        return 1
    else:
        return base * power_number(base, index - 1)
```

The base condition in this situation is to return 1 whenever the index is 0. This is consistent with mathematics, in that any number raised to the power of 0 is 1. If this base condition was absent or impossible to get to, then after running the code the interpreter will return a runtime error, stating that maximum recursion depth exceeded.

**COPYRIGHT
PROTECTED**



```
def power_number(base, index):
    if index == 15:
        return 1
    else:
        return base * power_number(base, index-1)

print(power_number(4,12))
```

In the example above, the index will never be equal to 15, as the index statement and decreases in value.

```
ors and Exception/errors.py", line 23, in power_number
    return base * power_number(base, index-1)
File "C:/Users/Devz/Downloads/Python Book/Python for
ors and Exception/errors.py", line 23, in power_number
    return base * power_number(base, index-1)
File "C:/Users/Devz/Downloads/Python Book/Python for
ors and Exception/errors.py", line 20, in power_number
    if index == 15:
RuntimeError: maximum recursion depth exceeded in comparison
>>>
>>> .
```

Whenever you suspect that a recursive definition is not getting to the base case, you can use a trace table to go through a simple case, or using print statements and variables at strategic locations in the function call.

9.4 Semantic or Logical Error

Semantic refers to the meaning of a language or logic. Semantic errors are more subtle than runtime errors. Similar to runtime errors, the interpreter will not give you an error message. The program will even run without hanging, or giving a maximum recursion error. The output will not be the one that you are expecting.

Fortunately there are strategies available to the programmer to identify and fix semantic errors.

- Inserting `print()` statement at strategic locations in your code to debug is quite helpful.
- The use of trace tables to record the value of variables can be also helpful.
- Sometimes it is worth breaking down complex expressions into simpler ones. Having large complex statements.

If you define a function called `larger()` that takes two numbers and returns the larger of the two. It is possible to use `larger(num1, larger(num2, larger(num3, num4)))` to find the largest of three numbers. That is equivalent to:

```
temp_num = larger(num1, num2)
larger(temp_num, num3)
```

This may seem trivial, but it turns out that it is less likely to make mistakes with simple expressions than on complex ones.

**COPYRIGHT
PROTECTED**



- Semantic errors often arise from the use of incorrect operators, for example, `>` instead of `>=`.
- It is often worth using brackets to explicitly force the correct order of operations, for example in the table above:

In evaluating $\frac{15}{2\pi}$ in Python we would type `15/2*math.pi`

As division and multiplication have the same precedence, we would get the incorrect answer 7.5. In order to get the correct answer we would therefore, we would type `15/(2*math.pi)`

9.5 Dealing with Errors

Even with the best intentions, there will be situations in which errors occur that are not foreseen. Many programming languages provide error handling facilities for exceptions. Exceptions are instances of the `BaseException` class and are implemented in Python. A debugger tool in Python is called the Python debugger and is imported using `import pdb`. There are two versions of the try statement, namely `try-except` and `try-finally`.

9.5.1 The Try Statement

Syntax:

```
try:
    <block_to_execute>

except <exception>:
    <block_to_execute>
```

`<block_to_execute>` is the originally intended code.

`except` must include at least one exception, but can include as many as needed.

Example:

```
try:
    x=int(input("Enter a number to divide: "))
    y=int(input("Enter a number to divide by: "))
    print(x/y)
except ZeroDivisionError:
    print("Cannot divide by zero: ")
    y=1
```

This will catch the exception, print a more meaningful statement and continue the program.

In the example above, the interpreter will try to execute all the statements. If the user enters zero for the `y` value, the `ZeroDivisionError` exception will be raised. In this case, the interpreter will catch the exception in the "except" clause and recover by reassigning `y` to 1. We could have included other exceptions in the "except" clause. The interpreter would search until it finds a matching exception to enter into the "except" clause. If no exception is found, the program will halt/stop and print the exception to the user.

**COPYRIGHT
PROTECTED**



The example below shows how we can catch an ImportError (import a file/

```
>>> import unknown_module
Traceback (most recent call last):
  File "<pyshell#60>", line 1, in <module>
    import unknown_module
ImportError: No module named 'unknown_module'
```

This is the import statement without the exception handler. Note that this program.

```
>>> try:
    import unknow_module
except ImportError:
    print("Sorry module not found")

Sorry module not found
>>>
```

With the exception handler, the program will recover and continue to run.

9.5.2 The Try Statement with Multiple Exceptions

There are situations in which more than one exception can occur in a particular option of writing multiple except clauses, where you check each exception one except clause where the interpreter will check through a list of exceptions one way to deal with all exceptions, then the latter is a better option, but in a separate solution then having multiple except clauses is the better solution

If we are writing a program and we want the user to enter a number and then perform a mathematical operation on this number, we will use the int() function to convert the input to an integer. There are two possible exceptions that can occur in this situation, namely:

TypeError: when an operation or function is attempted that is invalid for the type of the object.
ValueError: values have the valid type of arguments, but the arguments have the wrong value.

The code below will catch both the ValueError and TypeError exception and print a message. In the situation where the course of action is dependent on the type of exception, the code below is applicable.

```
try:
    x=int(input("Enter a number: "))
except ValueError:
    print("You entered an incorrect value: ")
except TypeError:
    print("The data type is invalid ")

print("Continuing after the input ...")
```

**COPYRIGHT
PROTECTED**



9.5.3 The Optional Else Clause

The optional else clause is used when you only want to execute statements in the try clause.

```
try:
    x=int(input("Enter a number: "))
except (ValueError, TypeError):
    print("Something went wrong")
else:
    print("Nothing went wrong")
```

Notice that if the user's input is valid, then Python will execute the optional else clause. In the code snippet above, we have changed from using multiple except clauses to using a single except clause with multiple exceptions (ValueError and TypeError).

9.5.4 The Try-Finally Statement

If we have multiple statements in the try block and one of the statements raises an exception, then the rest of the statements in the try block will not be executed. If you want the entire try block to be executed, even if an exception was raised, the finally statement will resolve this.

Syntax:

```
try:
    <block_to_execute>
```

```
finally:
    <finally_block>
```

<block_to_execute> is the originally intended code.

finally must block that; it will be executed regardless of any exception raised.

Example:

If we are writing a program that opens a file and do some calculations then we may want to raise an exception after attempting to open the file. If an error occurs while opening the file, then the IOError exception is raised. Consider the code below.

```
try:
    test_file = open("example_file.txt", 'w')
    test_file.write("Some operations")
except IOError:
    print("File don't exist ...")
    test_file.close()
```

If we encounter an error while trying to write to the file, the IOError exception will handle the exception and close the file. If no exception is raised, then the program will continue to execute. This is not good for our program. It turns out that the "try-finally" can help us. Whenever we use the "try-finally" statement if an exception is raised, then the program will execute the finally clause and then to the exception handler in the "except" section.

**COPYRIGHT
PROTECTED**



The code snippet below gives this solution.

```
try:
    test_file = open("example_file.txt", 'w')
    try:
        test_file.write("Some operations")
    finally:
        test_file.close()
except IOError:
    print("File don't exist ...")
```

Notice that if the write statement fails then the file will be closed.

INSPECTION COPY

**COPYRIGHT
PROTECTED**



Practice Exercise 9

1. What is a syntax error?

.....

2. Why are semantic or logical errors hard to spot?

.....

.....

3. What kind of error is in the code below?

```
def factorial(num):  
    if num ==1:  
        return 1  
    else:  
        return num * factorial(num)
```

.....

4. Correct the error in the function above so that it calculates the factorial. Factorial of n is $n * (n-1) * (n-2) * (n-3) * \dots * 1$.

.....

.....

.....

.....

.....

5. What type of error is in the code below?

```
number = 1  
while number< 13:  
    print(number, " Squared is ", number
```

.....

6. Correct the error so that the code prints the square numbers from 1

.....

.....

.....

INSPECTION COPY

**COPYRIGHT
PROTECTED**



7. Use the grammar below to show how the interpreter parses the expression `1 + 2 * 3 + 4`.
- a. `expression` \rightarrow `expression operator expression`
 - b. `expression` \rightarrow `number`
 - c. `operator` \rightarrow `+`, `-`, `*`, `/`, `%`, ...
 - d. `number` \rightarrow `0`, `1`, `2`, `3`, `4`, ...

.....

.....

.....

.....

8. Rewrite the code below so that if there are no exceptions while opening a file, then the file should close after the update.

```
try:
    test_file = open("new_file.txt", 'w')
    test_file.write("Some operations")
except IOError:
    print("File don't exist ...")
test_file.close()
```

.....

.....

.....

.....

.....

.....

**COPYRIGHT
PROTECTED**



Project Task

Decorating Company

John, a local decorator in your community, has approached you to keep track of the time and cost of his jobs. He lays carpets for customers. He has to give his customers accurate quotes, after he has taken measurements. You are required to develop a computer system that will help him to do this.

Carpets

When a room measurement is taken, the length and the width are taken to calculate the area of carpet needed in square metres. He also supplies underlay to go under the carpet and gripper to run along the edge to hold the carpet in place. The underlay required is the same as the amount of carpet required. As the gripper goes around the edge of the carpet, the length of gripper required is the perimeter of the carpet.

Some rooms have bay windows, fireplaces and other obstacles; therefore John has to measure before laying. This does not reduce the price of the carpet. The table below shows the prices.

Item	Price
Carpets	£22.50 per square metre
Underlay – First Step	£5.99 per square metre
Underlay – Monarch	£7.99 per square metre
Underlay – Royal	£60 per square metre
Gripper	£1.10 per metre

In addition to the raw material, John charges £65 for each hour worked as an estimator. He estimates that it requires an hour for each 16 square metres; this is therefore £4.06 per square metre goes above 16 square metres.

The system should allow the user to:

1. Enter the customer's details
2. List all customers in the system – from this list you should have the option to add a new customer or view a quote linked to a certain customer
3. Add a new quote
4. List all quotations – with an itemised list of the material and labour
5. Save customers and quotation details to an external file

For each customer the system should save:

- First name and surname
- Town
- Telephone number

Each quote consists of:

- A customer – the system should be able to add a new customer or select an existing customer when creating a quote
- The dimensions of the room (length, width)
- Select the type of underlay
- Calculate the overall cost

You should develop the part of the system that allows the user to save a quote to a file.

INSPECTION COPY

**COPYRIGHT
PROTECTED**



Practice Exercises Solutions

INSPECTION COPY

Solutions for Practice Exercise 1

```
1)
print(30*24*60*60)

2)
print(365*24*60*60)

3) Use Python as a calculator.
print(84*45)
print(75/15)
print(90-34)
print(23+45)

4)
print( (414/150)*60)

5)
help("keyword")

6)
print(49/7)
print(8**2)
print(20%3)
print(17//3)
print(7**3)

7)
print(500*1.20)

b.
print(320/1.32)

8)
print((4/3)*22/7*10**3)

9) the expression 36/9-2 to get:
a. 2
print( (36/9)-2 )

b. 5.12
print( 36/(9-2) )
```

Solutions for Practice Exercise 2

```
1)
length = 18
width = 7
perimeter = (2*length+2*width)
print(perimeter)

2)
days_in_each_school_year = 365
print(5*days_in_each_school_year)

3) What value will be printed?
marks = 25
marks = marks + 10
print(marks)

4)
time_spent = 34
time_spent = time_spent + 10
print(time_spent)

5)
Error --> hours_1 is not a variable, therefore will generate an error.

6) 24

7) False: An expression cannot be assigned to a variable.
#--> A variable cannot be assigned to an expression, not the other way round.

b)23

8) #4gs      &item

9) By typing help()

10) c. days in a year
```

Solutions for Practice Exercise 3

```
1)
quote = "Your favourite quotation"
print(quote)

2)
name="Patrick"
print(name)

3)
number1= int(input("Please enter the first number: "))
number2= int(input("Please enter the second number: "))
number3= int(input("Please enter the third number: "))

total = number1+number2+number3
print("The sum is ", total)
```

**COPYRIGHT
PROTECTED**



```

4)
name = input("What is your name: ")
age = int( input("How old are you?: "))
print("Hello ", name, ", please accept this warm message")

5)
x=34.67
print(type(x))

6) h

7) a

8)
title = "Python for key stage 4"
print(title[11:14])

9)
Name = "Benjamin"
print("Hello " + Name + ", Python is fun")

10)
title= "Python for key stage 4"

for letter in title:

    print(letter)

```

Solutions for Practice Exercise 4

```

1)
def double(num):
    return num*2

number = int(input("Please enter a number to double: "))
print(number, "doubled is ", double(number))

2)
def perimeter_of_rectange(length, width):
    return (2*length)+(2*width)

length = float(input("Please enter a Length: "))
width = float(input("Please enter a width: "))

print(perimeter_of_rectange(length,width))

3)
def area_of_circle(r):
    return (22/7)*r**2

radius = float( input("Please enter the radius: "))
print("The area is ", area_of_circle(radius))

4)
def hello_three_times(name):
    print("hello ", name*3)

person = input("Please enter the person to say hello to: ")

print(hello_three_times(person))

```

**COPYRIGHT
PROTECTED**



```

5)
def head_name(word):
    return word[0]

name = input("Please enter your name: ")
print("The head of ", name, " is ", head_name(name))

```

```

6)
def tail_name(word):
    return word[1:len(word)]

name = input("Please enter your name: ")

print(tail_name(name))

```

7) Using functions makes our program code more readable. When we actually naming a certain set of instructions that are performed, therefore making our program code easier to read.

Functions can reduce our program code significantly by putting a function. Changes or adjustments can be made in one place.

Functions facilitate modular design. We can break down a large task into chunks, work on these chunks in functions, then put the entire code for a working solution.

Functions can be placed into modules so that they can be reused.

8) Parameters are place holders that are used inside the function. Arguments are the actual values that you send when calling the function.

```

9)
def power_value(base, index):
    return base**index

base = float(input("Enter the number for the base: "))
index = float(input("Enter the index: "))

print("The power is ", power_value(base, index))

```

```

10)
def absolute_value(num):
    if num<0:
        return -num
    else:
        return num

print(absolute_value(-5))
print(absolute_value(0))
print(-absolute_value(-5))

```

Solutions for Practice Exercise 5

```

1)
number1 = int(input("Please enter the first number: "))
number2 = int(input("Please enter the second number: "))

if number1 > number2:
    print("The Larger number is ", number1)
else:
    print("The Larger number is ", number2)

```

**COPYRIGHT
PROTECTED**



```

2)
score = int( input("Please enter your score: "))

if score>49:
    print("Pass")
else:
    print("Fail")

3)
score = int( input("Please enter your score: "))

if score>49:
    print("Pass")
    if score>79:
        print("Well Done")
else:
    print("Fail")
    if score <20:
        print("You need to try harder")

4)
score = int(input("Please enter your score: "))

if score> 79:
    print("Your grade is A")
else:
    if score>59:
        print("Your grade is B")
    else:
        if score>39:
            print("Your grade is C")
        else:
            if score>29:
                print("Your score is D")
            else:
                print("Your score is U")

5)
word = input("Please enter a word to find out the vowel worth: ")

def vowel_worth(word):
    value = 0
    for letter in word:
        if letter == 'a':
            value = value + 5
        if letter == 'e':
            value == value + 4
        if letter == 'i':
            value = value + 3
        if letter == 'o':
            value = value + 2
        if letter == 'u':
            value = value+ 1
    return value

print("The value worth of ", word, " is ", vowel_worth(word))

```

**COPYRIGHT
PROTECTED**




```
6)
def larger(first, second):
    if first > second:
        return first
    else:
        return second

number1 = int(input("Please enter the first number: "))
number2 = int(input("Please enter the second number: "))

print("The Larger is ",larger(number1, number2))

7)
def long_name(name):
    if len(name)>13:
        return True
    else:
        return False

test_name = input("Please enter the name to test: ")

print(long_name(test_name))

8)
def larger(first, second):
    if first > second:
        return first
    else:
        return second

def largest(first, second, third):
    return larger(larger(first, second), third)

print("The Largest of 12, 45, and 90 is ", largest(12,45, 90) )

9)
def print_upto(number):
    for val in range(1,number+1):
        print(val)

num = int(input("Enter the number to print up to: "))

print(print_upto(num))

10)
def print_even_upto(number):
    for val in range(1,number+1):
        if val%2 ==0:
            print(val)

num = int(input("Enter the number to print up to: "))

print(print_even_upto(num))
```

**COPYRIGHT
PROTECTED**



```

11)
def magic_number(guess):
    number = 7
    while guess != number:
        if guess == number:
            print("Well done")
        else:
            if guess > number:
                print("Too high")
                guess = int(input("Please enter a guess: "))
            else:
                print("Too Low")
                guess = int(input("Please enter a guess: "))

guess = int(input("Guess the number: "))

magic_number(guess)

```

```

12)
def magic_number(guess):
    number = 7
    count = 0
    while (guess != number) and (count < 5):
        if guess == number:
            print("Well done")
        else:
            if guess > number:
                count = count + 1
                print("Too high")
                guess = int(input("Please enter a guess: "))
            else:
                count = count + 1
                print("Too Low")
                guess = int(input("Please enter a guess: "))

guess = int(input("Guess the number: "))

magic_number(guess)

```

```

13)
def factorial(num):
    if num == 1:
        return 1
    else:
        return num * factorial(num-1)

number = int(input("Enter the number to find the factorial of: "))
print(factorial(number))

```

Solutions for Practice Exercise 6

```

1)
bank_holidays_in_month = [1, 0, 1, 1, 2, 0, 0, 1, 0, 0, 0, 2]

def bank_holiday(number):
    return bank_holidays_in_month[number-1]

month = int(input("Please enter the number of the month to return bank holidays: "))
print(bank_holiday(month))

```

**COPYRIGHT
PROTECTED**



```

2)
my_list = [56, 3, 89, 2, 34, 12]

def my_sort(l):
    sorted_list = False
    sorted_pos = len(l)-1

    print(l)
    while not sorted_list:
        sorted_list = True
        for index in range(sorted_pos):
            if my_list[index] > my_list[index +1]:
                sorted_list = False
                #Swap values
                my_list[index], my_list[index +1] = my_list[index
                print(l)

my_sort(my_list)
print(my_list)

3)
my_list = [1, 2, 6, 89, 90]

def add_hello(l):
    return l.append("heLlo")

add_hello(my_list)

print(my_list)

4)
my_list = [150, 46, 69, 120, 36]

def discount_ten(l):
    for index in range(len(l)):
        l[index] = l[index]*1.1

discount_ten(my_list)
print(my_list)

5)
my_list = [12, 4,5, 3, 5, 34]

def remove_five(l):
    while 5 in l:
        l.remove(5)

remove_five(my_list)

print(my_list)

6)
def is_palindrome(word):
    end_pos = -1
    for index in range(int(len(word)/2)):
        if word[index] != word[end_pos]:
            return False
        else:
            end_pos = end_pos -1
    return True

test_word=input("Enter a word to test: ")
decision = is_palindrome(test_word)

print(decision)

```

**COPYRIGHT
PROTECTED**



```
7)
y= [ 1,2,2,3,3,4]
def unique_elements(l):
    x=[]
    for index in range(len(l)):
        test_value = l.pop()
        if test_value not in x:
            x.append(test_value)
    print(x)

unique_elements(y)

8)
z = [45, 67, 23, 56]

def backways(l):
    l.reverse()

backways(z)
print(z)

9)
z = [1, 3, 5, 7, 9, 10]

def sum_list(l):
    total =0
    for element in l:
        total = total + element
    return total

print(sum_list(z))

10)
z = [1, 3, 5, 7, 9, 10]

def sum_list(l):
    total =0
    for element in l:
        total = total + element
    return total

print(sum_list(z))

def mean_list(l):
    return sum_list(l)/len(l)

print(mean_list(z))

11)
z = [1, 3, 5, 7, 9, 10]

def sum_list(l):
    total =0
    for element in l:
        total = total + element
    return total

def mean_list(l):
    return sum_list(l)/len(l)

mean=mean_list(z)

def list_of_deviation(l):
    for index in range(len(l)):
```

**COPYRIGHT
PROTECTED**



```

        z[index] = z[index]- mean

list_of_deviation(z)

print(z)

12)
import math

z = [1, 4, 5, 7, 9, 20]

def sum_list(l):
    total =0
    for element in l:
        total = total + element
    return total

def mean_list(l):
    return sum_list(l)/len(l)

mean=mean_list(z)
#List of deviations
def list_of_deviation(l):
    for index in range(len(l)):
        l[index] = l[index]- mean

#Now square of deviations
def square_list(l):
    for index in range(len(l)):
        l[index] = l[index]*l[index]

def standard_dev(l):
    #mean=mean_list(l)
    list_of_deviation(l)
    square_list(l)
    variance = sum_list(l)
    return math.sqrt(variance/(len(l)-1))

std_dev = standard_dev(z)
print(std_dev)

```

Solutions for Practice Exercise 7

```

1a)
#Create a text file that shows the last five prime ministers in the UK
print("Writing text file")

# first open a file for writing
text_file = open("UKPrimeMinister.txt", "w")
text_file.write("Last five UK Prime Minister and term of office \n")
text_file.write("David Cameron           2010 - incumbent\n")
text_file.write("Gordon Brown           2007 - 2010\n")
text_file.write("Tony Blair             1997 - 2007\n")
text_file.write("John Major             1990 - 1997\n")
text_file.write("Margaret Thatcher      1979 - 1990\n")

text_file.close()

#now open the file and print the contents to the screen
text_file = open("UKPrimeMinister.txt", "r")
print(text_file.read())

#Finally close the file
text_file.close()

```

**COPYRIGHT
PROTECTED**



```

1b)
#Now write a python program that will read the file called UKPrime
#that was created above and print the content to the screen.

print("Reading text file")

#now open the file and print the contents to the screen
text_file = open("UKPrimeMinister.txt", "r")
print(text_file.read())

#Finally close the file
text_file.close()

2)
a)Change the mode from 'w' to 'a'
    new_file = open(file_name, 'a')

b) update the range arguments to 0-3
for index in range(0,3):

3)
#we first import the sqlite3 library
import sqlite3

#Create a handler called new_db that connects to db on disk
new_db = sqlite3.connect('C:\\Users\\Devz\\PythonExercise\\Library.db')

#Create a new cursor object to manipulate the database
c=new_db.cursor()

#Now create a table called Students
c.execute('''CREATE TABLE Books
(book_isbn text,
book_title text,
book_type text,
book_author text,
publisher text)
''')

#insert data into our table
c.execute('''INSERT INTO Books
VALUES ('978-0-340-88851-3', 'A2 Pure Mathematics', 'Non
fiction', 'Catherine Berry', 'Hodder Education')''')

c.execute('''INSERT INTO Books
VALUES ('978-1-118-10227-5', 'Android 4 Application Dev
lopment', 'Reto Meier', 'Wiley')''')

c.execute('''INSERT INTO Books
VALUES ('0-596-00699-3', 'Programming C#', 'Non fiction',
'Jesse Liberty', 'O Reilly')''')

#Save changes using the commit() function
new_db.commit()

#Close the connection to the database
new_db.close()

#Re-connect to the database
new_db=sqlite3.connect('C:\\Users\\Devz\\PythonExercise\\Library.db')

#Create a cursor object
c=new_db.cursor()

```

**COPYRIGHT
PROTECTED**



```
#Use the SELECT statement with wildcard to select all columns
c.execute("SELECT * FROM Books")
```

```
#use the fetchone function to fetch one row of data
book_library=c.fetchall()
```

```
#print the row to the screen
for book in book_library:
    print(book)
```

```
#Finally close the connection
new_db.close()
```

4) Displays the name, quantity and price from a table called Product table, where the quantity is greater than 20.

Solutions for Practice Exercise 8

1)

```
class Time:
```

```
    """ The Time class defines a time with attributes:
    hour, minutes and second
    """
```

```
    def __init__(self, hour=12, minutes=0, seconds=0):
        self.hour = hour
        self.minutes = minutes
        self.seconds = seconds
```

```
    #Attributes
    hour = 12
    minutes = 0
    seconds = 0
```

```
    def get_hour(self):
        return self.hour
```

```
    def get_minutes(self):
        return self.minutes
```

```
    def get_seconds(self):
        return self.seconds
```

```
    def print_time(self):
        print(self.get_hour(),":",self.get_minutes(),":",self.get_seconds())
    #and the mutator functions
```

```
    def set_hour(self, new_hour):
        self.hour, = new_hour
```

```
    def set_minutes(self, new_minutes):
        self.minutes = new_minutes
```

```
    def set_seconds(self, new_seconds):
        self.seconds = new_seconds
```

```
    def increment_seconds(self):
        if self.seconds == 59:
            self.seconds = 0
            self.minutes= self.minutes + 1
        else:
            self.seconds = self.seconds + 1
```

```
    def increment_minutes(self):
        if self.minutes == 59:
```

**COPYRIGHT
PROTECTED**



```

        self.minutes = 0
        self.hour = self.hour + 1
    else:
        self.minutes = self.minutes + 1

def increment_hour(self):
    if self.hour == 12:
        self.hour = 1
    else:
        self.hour = self.hour + 1

2)
print("Creating two instances of time")
time1 = Time() #Created with default values
time2 = Time(12,45,12) #user defined time

print("Now print both times ")
print("Default time --> ", time1.print_time())
print("User defined time --> ", time2.print_time())

time1.increment_hour()
time1.increment_minutes()
time1.increment_hour()
print("After using increment functions")
time1.print_time()

time2.increment_hour()
time2.increment_minutes()
time2.increment_hour()
print("After using increment functions")
time2.print_time()

3) class fraction:
    """ The fraction class defines a fraction with attributes:
    numerator, and denominator
    """

    def __init__(self, numerator =0, denominator = 1):
        self.numerator = numerator
        self.denominator = denominator

    #Functions
    def get_numerator(self):
        return self.numerator

    def get_denominator(self):
        return self.denominator

    def print_fraction(self):
        print(self.get_numerator(), "/", self.get_denominator())

    def print_type(self):
        if self.get_numerator() > self.get_denominator():
            return "Improper"
        else:
            return "Proper"

    def set_numerator(self, new_numerator):
        self.numerator = new_numerator

    def set_denominator(self, new_denominator):
        self.denominator = new_denominator

```

**COPYRIGHT
PROTECTED**




```

def inverse(self):
    temp = self.get_numerator()
    self.set_numerator(self.get_denominator())
    self.set_denominator(temp)

4)
#first create two fractions one with default values
# and the other with user defined values
fraction1 = fraction()
fraction2 = fraction(2,5)

#Now print the two fractions
fraction1.print_fraction()
fraction2.print_fraction()
print("fraction two is ", fraction2.print_type())

fraction2.inverse()
print("After the inverse fraction 2 is ", fraction2.print_type())
fraction2.print_fraction()

```

Solutions for Practice Exercise 9

- 1) A syntax error is one that breaks the grammar of the language. programmer uses a token (letter, symbol and operator) somewhere in the grammar of the language.
- 2) Semantic or logical errors are difficult to spot because they do not cause a problem to the programmer. The programmer will know that there is a problem, but the result that they are getting is not what they are expecting. These errors only occur under a given situation, which may not even happen.
- 3) The error is a semantic error. The interpreter will run the program and find an error, but the function will never compute the factorial of the number. There is nothing in the function that is bringing the number close to 1 (e.g., ==1). As it is written factorial(number) will keep multiplying indefinitely.
- 4)

```

def factorial(num):
    if num == 1:
        return 1
    else:
        return num * factorial(num-1)

```
- 5) The error is a semantic error. There is no statement within the function that is terminating condition (number >= 13) to this condition, therefore it will run forever, unless interrupted by some external exception.
- 6)

```

number = 1
while number < 13:
    print(number, " Squared is ", number*number)
    number = number+1

```

**COPYRIGHT
PROTECTED**



```
7) 8*2/1
expression
expression operator expression
number operator expression
8 operator expression
8 * expression
8 * expression operator expression
8 * number operator expression
8 * 2 operator expression
8 * 2 / expression
8 * 2 / number
8 * 2 / 1

8)
try:
    test_file = open("new_file.txt", 'w')
    try:
        test_file.write("Some operations")
    finally:
        test_file.close()
except IOError:
    print("File don't exist ...")
```

**COPYRIGHT
PROTECTED**



Project Task Solution

The solution uses the list data structure extensively and uses classes to store details. For the controlled assessment, you will need to show more in your solution. You need to show:

- An explanation of the problem – to demonstrate that you understand the problem you are going to solve
- A plan
- Your proposed solution – pseudo code or flow chart
- Solution development, with annotated code

The solution below is a basic solution with some important features missing:

- There is no error detection and correction present. The solution assumes the user will enter the correct data at all times. In particular, when the user is prompted to enter a number, it will be entered and nothing else. This needs to be more controlled for the controlled assessment.
- There is no data validation, for example, in entering length and width, the data is not validated.
- Whereas a list is a fast, effective way of storing and accessing data in real life, it would be more realistic to save the details (customers, quantities) to a file for more persistent use of data.
- Whereas the solution is meeting the client's need, it would be a better solution to have the price as the price, in an external file so that the client can change the price.

The Customer Class

```
class customer():
    """
    The customer class, storing the information for each customer
    firstname, surname, town, address
    """

    def __init__(self, firstname="First name", surname="surname",
telephone="070 000 0000"):
        self.firstname=firstname
        self.surname = surname
        self.town = town
        self.telephone = telephone

    def get_firstname(self):
        return self.firstname

    def get_surname(self):
        return self.surname

    def get_town(self):
        return self.town

    def get_telephone(self):
        return self.telephone

    def print_customer(self):
        print("Name is : ", self.firstname, self.surname)
        print("Town is : ", self.town)
        print("Telephone: ", self.telephone)
```

INSPECTION COPY

COPYRIGHT
PROTECTED



```
def set_firstname(self, new_name):
    self.firstname = new_name

def set_surname(self, new_surname):
    self.surname = new_surname

def set_town(self, new_town):
    self.town = new_town

def set_telephone(self, new_telephone):
    self.telephone = new_telephone
```

The Quotation Class

```
from customer import customer
```

```
class quotation():
```

```
    """
    Quotation is class that stores the information for a quotation
    a customer, quotation date, which is input automatically on creation
    dimensions (length and width), underlay_name. Length of gripper
    underlay_price, which is calculated based on underlay selected
    the total price.
    """
```

```
def __init__(self, customer, date, length, width, uname, gripper):
```

```
    """
    Constructor for quote.
    """
```

```
    self.customer = customer
    self.quotation_date = date
    self.length=length
    self.width = width
    self.underlay_name = uname
    self.gripper = gripper
    self.underlay_price = self.set_underlay_price()
    self.total_price = self.set_total_price()
```

```
def get_customer(self):
    return self.customer
```

```
def get_quotation_date(self):
    return self.quotation_date
```

```
def get_length(self):
    return self.length
```

```
def get_width(self):
    return self.width
```

```
def get_underlay_name(self):
    return self.underlay_name
```

```
def get_underlay_price(self):
    return self.underlay_price
```

```
def get_total_price(self):
    return self.total_price
```

```
def get_gripper(self):
    return self.gripper
```

**COPYRIGHT
PROTECTED**



```

#the print_quotation is used extensively to display the content
def print_quotation(self):
    print("Quotation Details ")
    print(self.customer.print_customer())
    print("Quotation date", self.quotation_date.day, "-",
          self.quotation_date.month, "-", self.quotation_date.day)
    print("Room measurement--> Length:", self.get_length(), "m",
          "Area -->", self.get_length()*self.get_width(), "sq. mts")
    print("Underlay", self.get_underlay_name())
    print("Underlay price", self.get_underlay_price())
    print("Total Price ", self.get_total_price())

#Total price calculated to include cost for material, and labour
def set_total_price(self):
    area = self.length*self.width
    if area >=16:
        labour_cost = (area/16) *65
    else:
        labour_cost = 65
    return (area*22.5)+(area*self.get_underlay_price())+labour_cost

def set_underlay_price(self):
    if self.get_underlay_name() == "First Step":
        return 5.99
    else:
        if self.get_underlay_name() == "Monarch":
            return 7.99
        else:
            if self.get_underlay_name() == "Royal":
                return 60
            else:
                return 0

def set_customer(self, new_customer):
    self.customer = new_customer

def set_quotation_date(self, new_date):
    self.quotation_date = new_date

def set_length(self, new_length):
    self.length = new_length

def set_width(self, new_width):
    self.width = new_width

def set_underlay_name(self, new_underlay_name):
    self.underlay_name = new_underlay_name

```

The Decorator's Driver File

```

from customer import customer
from quote import quotation
from datetime import datetime
import sys

customer_list = []
quotation_list = []

# main menu
def main_menu():
    print("Welcome to John's Decorating")

```

**COPYRIGHT
PROTECTED**



```

print("You have the following options:\n")
print("1)    Add a new customer")
print("2)    List all customer")
print("3)    Add a new quote")
print("4)    List all Quotation")
print("5)    Save customer details")
print("6)    Save Quotation to file\n")
print("0)    Exit")

choice = int(input("Select one --> "))
if choice == 1:
    add_customer()
if choice ==2:
    list_all_customer()
if choice ==3:
    add_a_quote()
if choice ==4:
    list_all_quotations()
if choice ==5:
    save_customer_details()
if choice ==6:
    save_quotation()

if choice ==0:
    print("Good bye .....")
    sys.exit()

def save_quotation():

    print("Saving Quotations ...")
    file_name=input("Enter the file name to save in : ")
    file_name = file_name+"_quotations.txt"
    quote_file = open(file_name,'w')
    for quote in quotation_list:
        quote_file.write(quote.customer.get_firstname()+"\n")
        quote_file.write(str(quote.get_quotation_date())+ "\n")
        quote_file.write(str(quote.get_length())+ "\n")
        quote_file.write(str(quote.get_width())+ "\n")
        quote_file.write(quote.get_underlay_name()+ "\n")
        quote_file.write(str(quote.get_gripper())+ "\n")
        quote_file.write(str(quote.get_underlay_price())+ "\n")
        quote_file.write(str(quote.get_total_price())+ "\n")
    quote_file.close()

    print("File saved")
    main_menu()

#function used to save customers details
def save_customer_details():
    print("Saving Customers' Details ...")
    file_name=input("Enter the file name to save in : ")
    file_name = file_name+"_customers.txt"
    cust_file = open(file_name,'w')
    for cust in customer_list:
        cust_file.write(cust.get_firstname()+"\n")
        cust_file.write(cust.get_surname()+ "\n")
        cust_file.write(cust.get_town()+ "\n")
        cust_file.write(cust.get_telephone()+ "\n")
    cust_file.close()

    print("File saved")

```

**COPYRIGHT
PROTECTED**



```

main_menu()

#function used to add a new customer, once confirmed save the customer
#customer list
def add_customer():
    print("Add customer")
    firstname = input("Enter the customer's first name: ")
    surname = input("Enter the customer's surname: ")
    town=input("Enter the customer's town: ")
    telephone = input("Please enter the customer's telephone number: ")
    customer1 = customer(firstname, surname,town, telephone)
    customer1.print_customer()
    customer_list.append(customer1)
    print("Customer added successfully ...")
    another = input("Would you like to add another? yes [y] or no [n] ")
    if another=='y':
        add_customer()
    else:
        main_menu()

#list all the quotation in the system
def list_all_quotations():
    number =1
    for quote in quotation_list:
        print("Quotation ", str(number))
        quote.print_quotation()
        number = number +1
        print("\n")

    print("[A] Add new      [D] Delete      [B] Back")
    choice=input("-->")
    if choice=='a' or choice=='A':
        add_a_quote()
    else:
        if choice=='d' or choice=='D':
            print("Enter the quotation number to delete")
            quote_num = int(input("--> "))
            quote = quotation_list[quote_num-1]
            print("Are you sure you want to delete: ")
            quote.print_quotation()
            choice = input("[Y] Yes      [N] No")
            if choice == 'y' or choice == 'Y':
                quotation_list.remove(quote)
                print("Quotation removed ")
                main_menu()
            else:
                list_all_quotations()
        else:
            main_menu()

def list_all_customer():
    number =1
    for cust in customer_list:
        print("Customer" , str(number))
        cust.print_customer()
        number = number +1
        print("\n")

    print("[A] Add new      [D] Delete      [V] View Quote for customer")

    choice = input("What would you like to do --> ")
    if choice == 'a' or choice=='A':
        add_customer()

```

**COPYRIGHT
PROTECTED**



```

else:
    if choice == 'd' or choice == 'D':
        delete_customer()
    else:
        if choice == 'v' or choice == "V":
            view_quote()
        else:
            main_menu()

def add_a_quote():
    print("Add new Quote")
    print("[N] New customer      [E] Exiting customer")
    choice=input("-->")
    if choice == 'n' or choice == 'N':
        customer1 = add_customer_for_quote()
    else:
        customer1 = search_for_existing_customer()

    length = int(input("Enter the Length: "))
    width = int(input("Enter the width: "))
    print("UnderLay Choice")
    print("[F] First Step (5.99)    [M] Monarch (7.99)    [R] Royal")
    choice=input("-->")
    if choice == 'f' or choice == 'F':
        uname = "First Step"
    else:
        if choice == 'm' or choice == 'M':
            uname="Monarch"
        else:
            uname="Royal"
    gripper = 2*length + 2*width
    quotation1 = quotation(customer1, datetime.now(), length, width)
    quotation1.print_quotation()
    print("[A] Add      [D] Delete")
    choice=input("-->")
    if choice=='a' or choice == 'A':
        quotation_list.append(quotation1)
        print("Quote added successfully")
        main_menu()
    else:
        main_menu()

def add_customer_for_quote():
    print("Add customer For new Quote")
    firstname = input("Enter the customer's first name: ")
    surname = input("Enter the customer's surname: ")
    town=input("Enter the customer's town: ")
    telephone = input("Please enter the customer's telephone number: ")
    customer1 = customer(firstname, surname,town, telephone)
    #customer1.print_customer()
    customer_list.append(customer1)
    return customer1

def search_for_existing_customer():
    number =1
    print("Here is a List of all customers ")
    for cust in customer_list:
        print("Customer" , str(number))
        cust.print_customer()
        number = number +1
        print("\n")
    print("enter the customer number to add a quote for ")
    cust_number=int(input("-->"))
    customer1=customer_list[cust_number-1]

```

**COPYRIGHT
PROTECTED**




```

    return customer1

def delete_customer():
    print("Delete Customer")
    cust_number=int(input("Enter the customer number to delete --"))
    customer1 = customer_list[cust_number-1]
    print("Customer")
    customer1.print_customer()
    print("Are you sure you want to delete [Y] Yes      [N] No")
    choice = input("--> ")
    if choice == 'y' or choice=='Y':
        customer_list.remove(customer1)
        print("Customer information Deleted ...")
    main_menu()

def view_quote():
    print("View Quote")
    print("Enter the customer's number to view quote")
    choice = int(input("-->"))
    customer1=customer_list[choice-1]

    #Create a dummy quote, just in case the customer do not have a
    target_quote = quotation(customer1, datetime.now(), 0, 0, "dummy")
    #Now search for the quotation with the customer
    for quote in quotation_list:
        if quote.customer == customer1:
            target_quote = quote
    if target_quote.get_underlay_name() == "dummy":
        print("Sorry", customer1.get_firstname(), "does not have a quote")
        print("Here is a list of all your customers")
        list_all_customer()
    else:
        target_quote.print_quotation()
        choice=input("continue --> ")
        main_menu()

main_menu()

```

**COPYRIGHT
PROTECTED**



Glossary

Absolute path	A path that gives the specific location of a file system address is given from. This should contain the root subdirectories to get to a file.
Algorithm	A collection of unambiguous and executable operations in a finite amount of time.
Abstract data type (ADT)	These are new data types that numerous operations can be performed on.
Argument	These are the actual values that are supplied during function calls. Arguments are then matched with the parameter in the function definition.
Attributes	Data items that an object possesses.
Boolean	This is a data type having two values, usually denoted as True and False.
Built-in function	These are functions made within an application and are available to all users.
Calculation	Any mathematical operations done with numbers.
Class	It is a user-defined data type that is made up of attributes and methods (are known as ADT).
Compilers	These create object codes which are passed on to the interpreter.
Computer science	The study of the design of algorithms, their proper implementation, and their mechanical realisation.
Control structures	Blocks of code that dictate the flow of control within a program.
Database	This is a collection of tables that store data.
Data Encapsulation	This is information hiding. It involves reusing codes and data. The user does not have to know the attributes or methods of the object.
Data types	The range of values that can be stored and the kind of operations (addition, subtraction, comparison, concatenation, etc.) that can be performed on a variable.
DBMS	Database Management System – software used to manage the database in a systematic manner.
Directory structure	The way in which an operating system displays its files and folders.
Exception	This is when an error occurs in a program. This can be handled by the programmer.
Expressions	A combination of operator and operations that evaluate to a single value.
Float	A data type made up of numbers that have a decimal point.
Function	These are blocks of code that perform a specific task.
Integer	This is a data type made up of any positive or negative whole numbers.
Interactive mode	A mode in Python where instructions are typed at the prompt and each individual line of instruction is executed at a time.

INSPECTION COPY

**COPYRIGHT
PROTECTED**



Interpreters	These take each line, translate and execute the translation, then give their attention to the other line.
List	This is a mutable collection of data items which can contain different data types.
Object	This is a thing or an event in our application.
Object-oriented programming	A type of programming that defines the data type and the structure.
Parameter	These are the place holders that are used in defining a function.
Program code	The set of instructions normally written in high-level language that the computer follows to complete a task.
Programming paradigm	Different disciplines and principles involved in programming.
Python	Python is an interpreted, general purpose programming language that conforms to multiple ways of programming.
Query	A question that is asked of the database.
Quotient	This is the number of times one quantity can be divided by another to get the result of a division.
Recursive	A rule or a procedure that can be applied repeatedly to itself.
Relative path	This starts the path by using the current directory.
Runtime error	An error that occurs during the execution of a program.
Script	A program that is normally interpreted, stored as a text file.
Semantics	An error in the logics of the program. It encompasses the program and its output.
SQL	Structured Query Language – the universal language to query and manipulate databases.
Statements	This is used to store the value of an expression in a variable.
String	This is a data type made up of characters. These can be alphanumeric symbols.
Syntax	An error of language; usually the code does not conform to the language.
Values	Any literal that can be stored in a memory location.
Variables	A named memory location.

References

Aitel, David; Diamond, Jason; Foster-Johnson, Eric; Parker, Aleatha; Peter, Mark; Roberts, Michael; Samuel, Alex, *Beginning Python*, (Indianapolis: Wiley Publishing, 2002), Downey, Allen (2008), *Think Python: How to Think like a Computer Scientist*, (New York: No Starch Press, 2008), Hetland, Magnus Lie, *Beginning Python, From Novice to Professional* (Apress, 2005), Gupta, Rashi, *Making Use of Python*, (New York: Wiley Publishing Inc., 2002).

**COPYRIGHT
PROTECTED**

