­­­

A picture containing logo

Description automatically generated

**PYTHON LESSONS**

An introduction to Programming Version 3

By

G. Bradshaw

© 2022

Contents

[Lesson 1 – Console and Variables 8](#_Toc122366226)

[The Console & Basic operations 9](#_Toc122366227)

[Task 1.1 – Basic Commands 9](#_Toc122366228)

[Variables 10](#_Toc122366229)

[Task 1.2 – Creating Variables 10](#_Toc122366230)

[Task 1.3 – Different types 11](#_Toc122366231)

[Converting Variable Types 12](#_Toc122366232)

[Task 1.4 – Type Errors 12](#_Toc122366233)

[Task 1.5 – Type Conversions 13](#_Toc122366234)

[Task 1.6 – Type Corrections 13](#_Toc122366235)

[Output to screen 15](#_Toc122366236)

[Task 1.7 – Printing 15](#_Toc122366237)

[Writing your first program in IDLE 16](#_Toc122366238)

[Task 1.8 – Creating Program 16](#_Toc122366239)

[Input function 17](#_Toc122366240)

[Tasks – input and output 19](#_Toc122366241)

[Inputting Numbers 20](#_Toc122366242)

[Arithmetic operations 20](#_Toc122366243)

[Tasks 20](#_Toc122366244)

[Extension Task 22](#_Toc122366245)

[Good practice 23](#_Toc122366246)

[Tasks 24](#_Toc122366247)

[Lesson 2 - Selection 25](#_Toc122366248)

[Changing the flow of a program 26](#_Toc122366249)

[Comparison Operators 28](#_Toc122366250)

[Tasks 28](#_Toc122366251)

[Nested IF statements 30](#_Toc122366252)

[Tasks 30](#_Toc122366253)

[Logical Operators 32](#_Toc122366254)

[“IF”…“Else If” statements 33](#_Toc122366255)

[Tasks 34](#_Toc122366256)

[Lesson 3 - Iteration 35](#_Toc122366257)

[Fixed Loops– iterating by sequence 36](#_Toc122366258)

[Tasks 39](#_Toc122366259)

[Extension 39](#_Toc122366260)

[Loops within in loops 40](#_Toc122366261)

[Continue Nested Loop 40](#_Toc122366262)

[One-Line nested loops 40](#_Toc122366263)

[Tasks 40](#_Toc122366264)

[Extension 41](#_Toc122366265)

[Infinite Loops 42](#_Toc122366266)

[Conditional loops 43](#_Toc122366267)

[Tasks 45](#_Toc122366268)

[Validation 46](#_Toc122366269)

[Tasks 47](#_Toc122366270)

[Extension 47](#_Toc122366271)

[Lesson 4 - Functions, Strings and Dates 48](#_Toc122366272)

[Introduction to Functions 49](#_Toc122366273)

[More built-in functions 51](#_Toc122366274)

[Tasks 51](#_Toc122366275)

[Additional Non-core Functions - Modules 52](#_Toc122366276)

[RANDOM module 52](#_Toc122366277)

[Import methods 54](#_Toc122366278)

[Tasks 54](#_Toc122366279)

[Strings – a data structure 55](#_Toc122366280)

[String slicing 56](#_Toc122366281)

[Useful String recap 58](#_Toc122366282)

[TASK 58](#_Toc122366283)

[String Methods 59](#_Toc122366284)

[Tasks 60](#_Toc122366285)

[Formatting Strings 61](#_Toc122366286)

[Format operator (old style) 61](#_Toc122366287)

[Format method (new style) 64](#_Toc122366288)

[Tasks 66](#_Toc122366289)

[Dates 67](#_Toc122366290)

[Manipulating Dates 69](#_Toc122366291)

[Tasks 70](#_Toc122366292)

[Creating Functions 71](#_Toc122366293)

[Tasks 72](#_Toc122366294)

[Function Parameters 74](#_Toc122366295)

[Tasks 74](#_Toc122366296)

[Function Returns 75](#_Toc122366297)

[Tasks 75](#_Toc122366298)

[Lesson 5 - Lists, Tuples, and Dictionaries 76](#_Toc122366299)

[Introduction to Lists 77](#_Toc122366300)

[List Examples 78](#_Toc122366301)

[Tasks 79](#_Toc122366302)

[Iterating through LISTS 79](#_Toc122366303)

[Initialising Lists 80](#_Toc122366304)

[Tasks 81](#_Toc122366305)

[Processing Lists 82](#_Toc122366306)

[List indexes 84](#_Toc122366307)

[Tasks 85](#_Toc122366308)

[Searching – linear search 86](#_Toc122366309)

[TASKS 87](#_Toc122366310)

[Bubble Sort 88](#_Toc122366311)

[Tasks 88](#_Toc122366312)

[2-Dimensional Lists 89](#_Toc122366313)

[Tasks 90](#_Toc122366314)

[Further Processing 91](#_Toc122366315)

[Slicing lists 91](#_Toc122366316)

[Indexing through a list 92](#_Toc122366317)

[Common List Maths Functions 92](#_Toc122366318)

[Tasks 93](#_Toc122366319)

[Tuples 93](#_Toc122366320)

[DICTIONARIES 94](#_Toc122366321)

[Tasks 95](#_Toc122366322)

[Dictionary Methods 97](#_Toc122366323)

[Tasks 99](#_Toc122366324)

[Dictionary Lists 100](#_Toc122366325)

[Teacher Mark book Project 101](#_Toc122366326)

[Dictionaries within Dictionaries – Extension work 104](#_Toc122366327)

[Tasks 105](#_Toc122366328)

[Lesson 6 - Structured programming 106](#_Toc122366329)

[Introduction to Structured Programming 107](#_Toc122366330)

[Defining Functions 108](#_Toc122366331)

[Global and Local Variables 109](#_Toc122366332)

[Parameters 111](#_Toc122366333)

[Returning multiple values 112](#_Toc122366334)

[Tasks 112](#_Toc122366335)

[Pass by Value, Pass by Reference 114](#_Toc122366336)

[Tasks 118](#_Toc122366337)

[File Lesson 7 - Filing 120](#_Toc122366338)

[Introduction to Files 121](#_Toc122366339)

[Writing File 123](#_Toc122366340)

[Reading a file 124](#_Toc122366341)

[Tasks 127](#_Toc122366342)

[Comma Delimited Files 128](#_Toc122366343)

[Tasks 130](#_Toc122366344)

[Tasks 132](#_Toc122366345)

[Extension Task 132](#_Toc122366346)

[Other filing methods 133](#_Toc122366347)

[Tasks 134](#_Toc122366348)

[Direct Access Filing 135](#_Toc122366349)

[Tasks 136](#_Toc122366350)

[137](#_Toc122366351)

[Lesson 8 – Object Oriented Programing 138](#_Toc122366352)

[Introduction to object oriented 139](#_Toc122366353)

[Classes and Objects 139](#_Toc122366354)

[EMPLOYEE EXAMPLE 141](#_Toc122366355)

[Objects - Instance of a Class 142](#_Toc122366356)

[Private and Public methods/attributes 143](#_Toc122366357)

[Built-In Class Attributes 146](#_Toc122366358)

[Tasks 147](#_Toc122366359)

[Instance vs Class Variables 148](#_Toc122366360)

[Tasks 150](#_Toc122366361)

[Inheritance 151](#_Toc122366362)

[Tasks 153](#_Toc122366363)

[Inheritance – Super Method 154](#_Toc122366364)

[Tasks 156](#_Toc122366365)

[Association, Aggregation and Composition 157](#_Toc122366366)

[OO Project – Class Assignments 162](#_Toc122366367)

[Lesson 9 – SQL Databases 162](#_Toc122366368)

[Introduction 164](#_Toc122366369)

[Create Database 164](#_Toc122366370)

[Database Tables 166](#_Toc122366371)

[Field Data Types 166](#_Toc122366372)

[Create Table 167](#_Toc122366373)

[Task 168](#_Toc122366374)

[Amend Table 169](#_Toc122366375)

[Task 169](#_Toc122366376)

[Adding Data 170](#_Toc122366377)

[TASK 171](#_Toc122366378)

[Querying 171](#_Toc122366379)

[TASKS 172](#_Toc122366380)

[User Data Entry 172](#_Toc122366381)

[Data Updating 175](#_Toc122366382)

[TASKS 176](#_Toc122366383)

[TASKS - Books Sample Database 177](#_Toc122366384)

[Lesson 10 – TKinter 179](#_Toc122366385)

[Introduction 180](#_Toc122366386)

[Program Structure 180](#_Toc122366387)

[Adding Widgets 181](#_Toc122366388)

[Task 182](#_Toc122366389)

[Tasks 191](#_Toc122366390)

[Menus & Multiple Windows 194](#_Toc122366391)

[Task 194](#_Toc122366392)

[Data Entry and Processing – Student Marks System 195](#_Toc122366393)

[Tasks 199](#_Toc122366394)

# Lesson 1 – Console and Variables

|  |  |
| --- | --- |
| **Lesson Outcomes**  In this lesson you will learn:   * how to use the Python Console. * the use of variables and basic arithmetic operations. * about different data types. * how to get input from the user. * how to create your first program. | **C:\Users\Graham\AppData\Local\Microsoft\Windows\Temporary Internet Files\Content.IE5\X6CHINOH\MC900441498[1].png** |

## The Console & Basic operations

In Python the console is the command line interface where are all programs are executed. It is referred to as the SHELL and is used to either run programs or commands. So, for example, the most common command is “print” which you can type in the shell as follows:

Text

Description automatically generated

NOTE: Python is case sensitive e.g., type **prin**t as opposed to **Print**

The command **print** is followed by an opening bracket and then the text you want to display enclosed in quotes – you may use single or double quotes. You then finish the statement with a closed bracket. As well as print we may use a whole array of different expressions and commands.

### Task 1.1 – Basic Commands

Try typing the following commands and basic expressions (press return after each command):

>>> print (‘Hello world’)

>>> 1 + 5

>>> 10 -20

>>> 5 \* (10 + 30)

>>> 10 /2

>>> print(‘hello world’ + ‘, this is my first program’)

After each entry you should get the following:

Graphical user interface, text, application

Description automatically generated

The result of each command is displayed on the line below. You can use this shell to type in any command or expression and see the results. Try experimenting with your own expressions.

## Variables

**Variables** are a temporary storage of data. You can think of variables as like those used in algebra

x = 10, y = 20

Entering x \* y would output 200.

In programming variables can not only store numbers but a whole range of different data types. They are used extensively in programs to store data in memory while operations are performed upon them. Again, we can define variables within the console.

### Task 1.2 – Creating Variables

Try typing the following commands (press return after each command):

>>> x = (10 +2)

>>> y = x \* 2

>>> x

>>> y

You should get the following results:

In the above exercise you assign the results of calculations to different variables. In assigning a number to a variable the **interpreter** (Python) tells the computer that this variable is storing values of TYPE **integer** e.g., whole number. Obviously, if you were to type something other than a number, for example text, it would store it as a different type e.g., for text it would be a **STRING.**

### Task 1.3 – Different types

Try the following:

>>> x = “My name is John”

>>> a = 10

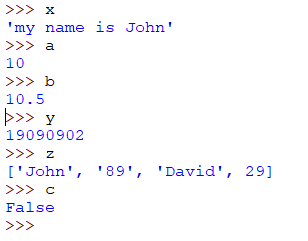
>>> b = 10.5

>>> y = 19090902

>>> z = [‘John’, ‘89’, ‘David’, 29]

>>> c = False

You can check the content of these variables by simple typing the variable name and pressing return:



Each one of the above variables have been allocated a **DATA TYPE.**

|  |  |  |
| --- | --- | --- |
| **Variable** | **Content** | **Data Type** |
| x | “My name is John” | This is what we call a **STRING** |
| a | 10 | This is a whole number referred to as an **INTEGER** |
| b | 10.5 | This is a decimal number and is what we call a **FLOAT**. |
| z | [ ‘john’, 89, ‘David’, 29] | This is a list of values and in Python is stored as type **LIST** (we will learn about these later in the course) |
| c | False | This is a **BOOLEAN** type, which can be one of two states: FALSE, TRUE |

## 

## Converting Variable Types

### Task 1.4 – Type Errors

When we create variables, Python assigns a type to that variable. So, as discussed before, if you assign a whole number to a variable then it becomes an **INTEGER** data type. In Python, and indeed in many languages, you can’t mix data types unless you tell the interpreter to convert from one to another. So, for example, if you try to add a STRING “Hello my name is John” to the number **9 (INTEGER)** then Python will give you an error – as you would expect.

Try the following:

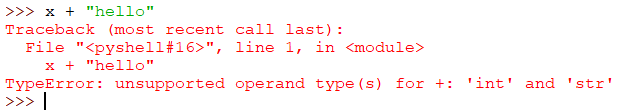
>>> x = 39033300000

>>> y = 10

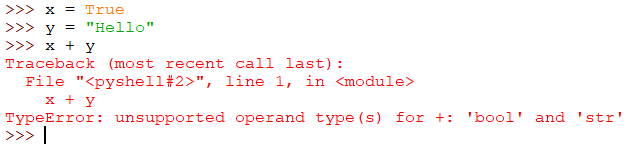
>>> x + y

>>> x + “Hello”

The last statement should create an **ERROR**. The reason for this is an **INTEGER** data type is being added to a **STRING data type.** Python doesn’t allow you to mix or add completely different **DATA TYPES** together without some conversion first.



Again, if you try MIX a **BOOLEAN** with a **STRING**:



To help us to mix and match different **DATA TYPES** we can use **type conversion functions.** A **function** is given a name which takes a **PARAMETER** (a user defined value) placed in brackets:

|  |  |
| --- | --- |
| **Function** | **Description** |
| **int (“10”)** | converts the TEXT “10” into a number |
| **str (1)** | converts the number 1 to a string i.e. text |
| **float (10)** | converts the integer 10 to a decimal |
| **float (“10.4”)** | converts the string “10.4” to a decimal |
| **int (10.6)** | converts the decimal 10.6 to an integer e.g. the number 10 (truncates rather than rounds) |

So, for example, if we want to add an **INTEGER** to a **STRING,** we must type the following:

A picture containing diagram

Description automatically generated

### Task 1.5 – Type Conversions

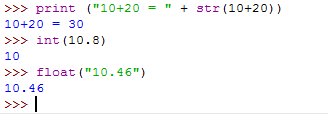
Type the following commands:

>>> print (“10 + 20 = “+ str (10+20))

>>> int (10.8)

>>> float (“10.46”)

You should get:



In the first print command we convert the result of the expression 10+20 into a STRING to display. If you didn’t have this conversion Python would display an error – you are trying to bring together to different data types. The second truncates the number 10.8 to the integer 10, and the final command converts the string “10.46” into a floating-point decimal number.

### Task 1.6 – Type Corrections

Use the Type Conversion Functions **int(), str(), float() CORRECT** the following statements (they have deliberate errors in them).

>>> 10 + “20”

>>> “One plus twenty-two equals “+ (1 + 22)

>>> 20.0 + 40

>>> “£s and pence: “+ (20 + 5.56) + “left”

>>> int (20 + 5.56)

>>> int (“25.56”)

For example, for the first one to work you would type:



This converts the **STRING “20”** into the **INTEGER 20.**

## Output to screen

One of the most useful functions in Python is print(). This is used to output content to the screen. The output can include a mixture of variables and data types, each separated by a “,”. You enter the print function and include in the brackets what you want to output, as follows:

### Task 1.7 – Printing

Type the following statements in Python Console:

>>> print (“hello world”

>>> print (“I am”,15, “years old”)

>>> items = 10

>>> print (“I have”, items, “items”)

Printing using multiple lines (“”” three quotation marks):

Text

Description automatically generated

## Writing your first program in IDLE

From the editor, Select **File à New File**

Graphical user interface, text, application

Description automatically generated

### Task 1.8 – Creating Program

Enter the following code in the new window:

Graphical user interface, text

Description automatically generated

Graphical user interface, text, application

Description automatically generated

When you have written your program save in an appropriate folder, making sure it has the extension **.py** :

**Run the program (on IDLE, select Run à Run Module, or press F5)**

This program should display the HELLO message and then prompt you for your name. Once you have entered your name the program should display your entry in a “HELLO” message.

Text

Description automatically generated with medium confidence

In this program, you will notice a new **function called input ().**

Text

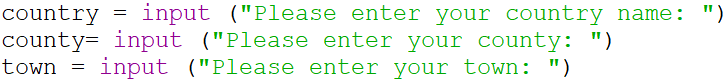
Description automatically generated with medium confidence

This creates a variable **called “name**” which is assigned the value of whatever the use inputs.

The ***input*** command prompts the user for data input.

## Input function

The input function is used to get to prompt the user for text input. This data is then stored in a variable. **By default, the input function always returns a STRING.**

****

Once you have assigned the variables, you may then printout the contents:



Notice you can mix the **STRINGS** with the **VARIABLES** by separating each by a comma. Running the complete program will output the following:

Text, table

Description automatically generated with medium confidence

Notice the spaces before the comma – you can remove this using the “+” to add the strings together. This only works if adding **TEXT**. This is called **CONCATENATION:**



Outputs:



### Tasks – input and output

1.9 Write a program that ask for the user’s surname and forename. Assign each to an appropriate variable. The program should then output:

Hello **surname, forename**  
 How are you?

1.10 Write a program that asks for your full address including postcode. After the user has entered the details display in following format:

Your address is:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

20 South Drive  
Allworth  
Winchester  
Hampshire  
S022 4DC

1.11 Using **STRING CONCATENATION** create a program that prompts the user for the following details and assigns to appropriate variables with types:

*Name, favourite colour, age, height in metres, gender, year group.*

Make sure you convert the TEXT input into the appropriate DATA TYPE e.g., age is an INTEGER, and you will need to use the INT function to convert.

Output the following format:

Text

Description automatically generated

## Inputting Numbers

As you have seen when using the input command Python prompts the user to enter **TEXT** which is then assigned to a variable. The problem with TEXT is you can’t perform any mathematical calculations on it without turning the DATA TYPE into a **NUMBER. As mentioned before, this can be achieved using the INT function.**

Graphical user interface, text, application

Description automatically generated

The above program asks the user for an input and then converts the **TEXT** input into an **INTEGER** number for the variable. The sum of the two numbers entered is then displayed.

## James Harley

## Arithmetic operations

It is important in programming to perform basic arithmetic operation. Python allows many different types of arithmetic operations:

|  |  |  |
| --- | --- | --- |
| **Arithmetic Operator** | **Operation** | **Example** |
| **+** | Addition | X + Y |
| **-** | Subtraction | Result – 1 |
| **\*** | Multiplication | P \* Interest Rate |
| **/** | Division | X / 6 |
| **//** | Integer Division | X // 6  i.e.  10 // 6 Returns 1 |
| **%** | Remainder | Y % 6  i.e.  10% 6 Returns Remainder 4 |
| **\*\*** | Performs power | X\*\*Y will give X to the power of Y  i.e.  10 \*\* 2 Returns 100 |

### Tasks

1.12 Write a program that prompts the user for two numbers and then displays the result of the first number entered multiplied by the second.

1.13 Write a program which prompts the user for 3 numbers and then displays the sum of those numbers.

1.14 Prompt the user to enter the length, width, and depth of a rectangular swimming pool. Assign each value to a variable. Using those variables calculate the volume of water required to fill the pool. Output the volume. For example:

Text

Description automatically generated

1.15 The formula for calculating the AREA of a triangle is:

Area = ½ \* base \* height

Write a program which prompts the user for the base and height in cm of a triangle as a decimal. The program should output the area.

1.16 Write a program to convert km into miles. The user should be prompted for the number of km, and the conversion into miles will be outputted. The conversion is 1km = 0.62137 miles.

1.17 Write a program that converts pounds to kilograms, using the rule 1 pound = 0.45 kilograms.

1.18 When doing calculations in Python standard mathematical order or precedence is applied i.e., BIDMAS.

Try the following:

Graphical user interface, text, application

Description automatically generated

1.19 Write a program which prompts the user for the length and width of a rectangle. The program should output the PERIMETER of the rectangle.

1.20 **Is it ODD or EVEN?**

Write a program that given a number will output 0 or 1 depending on if the number is ODD or EVEN. If EVEN the output should be 0 and if ODD, the output should be 1.

**HINT**: what number do we need to divide by to always get a remainder of either 1 or 0?

1.21 Write a program, using the “%” and “//” mathematical operators, that will read in two integers, number1 and number2, and display the whole number part and the remainder of dividing number1 by number2.For example:

A picture containing text

Description automatically generated

### Extension Task

1.22 Using the following arithmetic operators:

**%** - remainder  
 **//** - INTEGER division.

Write a program to enter an amount of money as a whole number, for example £78, and display the minimum number of £20, £10, £5 notes and £2 and £1 coins that make up this amount.

For example, the value £78 would give 3 twenty-pound notes, 1 ten-pound note, 1 five-pound note, 1 two-pound coin and 1 one-pound coin.

## Good practice

#### Naming variables

You have already started to use lots of variables in your programs, with each one given a different name e.g., number1 or x or y. In programming it is considered good practice to make sure that variables are named sensibly according to purpose and the nature of the data stored.

For example, in **Task 1.14** calculating volume of a swimming pool, you may have used a variable called “x” for the width and “y” for the length and “z” for the depth. Although this will work it doesn’t tell you much about what the variable is and what it stored. This becomes very important when creating large programs with lots of variables. Trying to debug a program where you don’t know what the variable is and what it does can be extremely frustrating! To resolve this problem, we simply try to give variables sensible names. So, in Task 1.14, good names for your variables could be ***length, width,*** and ***depth***. For the final calculation store the result in a variable called ***volume.***

***Good variable names:***

|  |  |
| --- | --- |
| ***name*** | (stores name) |
| ***rectangle\_volume*** | (using the “\_” underscore to illustrate a space) |
| ***pos\_x*** | (pos short for position at x-axis) |
| ***student\_mark*** | (again, using underscore for space. Stores a student mark) |

Note:

* A variable name must start with a letter or the underscore character.
* A variable name cannot start with a number.
* A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and \_ )
* Variable names are case-sensitive (score, Score and SCORE are three different variables)

#### Using Comments

When writing very complex programs it is a good idea to include comments in your code to help the reader understand how your program works. To add comments, we simply use the “#” symbol to indicate the following is a text comment and not to be executed or used in the program. Examples of comments:

Text

Description automatically generated

Adding COMMENTS in code using the “#” symbol

#### Appropriate use of spacing and layout

When creating a program, it is always good to include appropriate spacing and layout so that the program reads well, aiding debugging and testing.

Text, letter

Description automatically generated

### Tasks

1.23 The following program is poorly laid out, no commentary and inappropriate variable names. Add comments, layout, and change variables to have appropriate names.

Text

Description automatically generated

# Lesson 2 - Selection

|  |  |
| --- | --- |
| **Lesson Outcomes**  **C:\Users\Graham\AppData\Local\Microsoft\Windows\Temporary Internet Files\Content.IE5\X6CHINOH\MC900441498[1].png**  In this lesson you will learn:   * how to change the flow of a program. * the implementation of the IF… Else command. * comparison and logical operators. * and the more advanced multiple IF concept. |  |

## Changing the flow of a program

In programming one of the most powerful concepts is **SELECTION**. Selection allows us to change the natural flow of a program so that decision making can be included. You can represent this easily using program flowcharts:

A picture containing text, iPod

Description automatically generated

Figure 1 Flowchart

The above flowchart asks the user to enter two numbers which are stored in separate variables. These variables are then compared, if Number1 is bigger than Number2 then is output “Number 1 is the biggest” else output “Number2 is the largest value” is outputted. **Notice we aren’t checking if the two numbers are the same.**

This is a fairly simple way of changing the natural sequence of commands in a program. In Python this is implemented using the **IF** command. It is used to test if a condition is met and then change the program flow accordingly.

We can represent the flowchart in Figure 1 just using **IF** and **ELSE** in Python as:

A screenshot of a computer

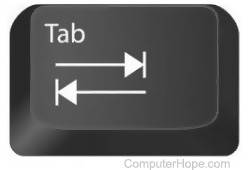
Description automatically generated with medium confidence

Statement executed if condition is **TRUE.**

The **ELSE** command is used when the CONDITION above is **FALSE.**

PYTHON uses **INDENTATION** to associate the PRINT as part of the **IF** statement.

In the above example the **CONDITION** is included after the **if** statement, with the colon representing the end. The print command is **indented** on the next line to show this is the part of the program executed if the condition is **TRUE**.



**NOTE: INDENTING** is very important in Python as it tells it how to **BLOCK** and execute the code**.** We do this using the **TAB key.**

**IF STATEMENT**

Indented code placed here. This code will run in the event IF statement is **TRUE.**

**ELSE**

Indented code placed here. This code will run in the event the IF statement is **FALSE**

The else command is used to tell Python the indented block code below is executed if the “IF” condition was FALSE.

## Comparison Operators

As well as equals you also have the following comparison operators:

|  |  |
| --- | --- |
| **Operator** | **Description** |
| **==** | Equals to |
| **>** | Greater than |
| **<** | Less than |
| **!=** | Not equal to |
| **>=** | Greater than or equal to |
| **<=** | Less than or equal to |

In the above table you will notice the **“==”** command has been used to check if the two numbers are equal. The reason why two equal signs have been used as opposed to one is so that PYTHON can tell the difference between **ASSIGNING** a value and **COMPARING** a value. To assign a value to a variable we use =, and to COMPARE we use.

### Tasks

2.1 Type the following program:

A picture containing chart

Description automatically generated

Change and add to the program so that it displays which out of the two numbers is smallest. Also, check if the numbers are the SAME, if they are output “they are the same”. Test the program works by changing the values assigned to number1 and number2.

2.2 Type the following program:

Text

Description automatically generated

Change the program so that if the number is greater or equal to 50 output they have passed otherwise output they have failed and will need to try again. Test the program works with different values assigned to student\_mark.

2.3 Type the following program:

A picture containing text

Description automatically generated

There is an error. Change the code so that it runs correctly.

Add to the code so that if the user selects no the program outputs *“Thank you for using this program. Have a good day!”.*

2.4 Enter the following program:

A picture containing text

Description automatically generated

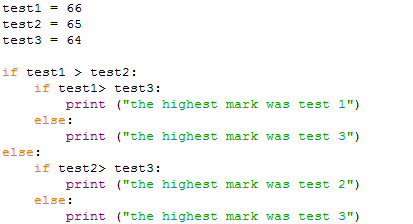
Adapt the above program to prompt for two numbers from the user and then displays a suitable message if the two numbers are the same (remember to convert user input to number using the int command).

2.5 Using an **ELSE** statement change the program in 2.1 so that it displays the largest of the two numbers (ignore if they are the same).

## Nested IF statements

So far you have learned how to create a simple **“IF..ELSE”** statement however there are times when more complex statements are needed ( for example, to fully implement flowchart in Figure 1 Flowchart). In Python, and indeed in most programming languages, it is possible to include **MULTIPLE IF** statements. For example, say we wanted to find the largest of three numbers:

**1st** IF Condition



**2nd** IF: if first condition **TRUE**

**ELSE** statement if **SECOND** IF **FALSE**

**INDENTING** if, else and the print statement

This program goes through and systematically compares each number until the largest one is found. You will notice that there is an IF within an IF from line 5 onwards – this is called a **Nested IF Statement.**

IF STATEMENT

Indented code placed here.

IF STATEMENT

Indented code placed here.

ELSE

Indented code placed here. This code will run in the event the first IF statement is **FALSE**

### Tasks

2.6 Write a program that asks the user to enter 3 numbers and displays the largest.

2.7 Write a program which asks their age and display whether they can drive or not.

2.8 Adapt the program in 2.6 to detect whether all 3 values are the same. Could we adapt to say what happens if 2 numbers are the same and the biggest values?

2.9 Write a program that lets the user enter a number between 1 and 12 and displays the month name for that month number. The input “3” would therefore display March.

2.10 Write a program that reads in the temperature of water in a container (in Centigrade) and displays a message stating whether the water is frozen, boiling or neither.

2.11 Write a simple quiz program which asks the user five questions, each providing a series of options to select as the correct answer. For example:

Chart, scatter chart

Description automatically generated

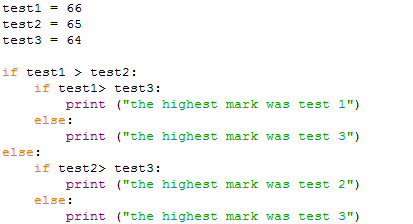
If the user chooses the correct option, the program outputs correct or otherwise incorrect. Create five questions.

Add to the program to keep a running score.

2.12 Write a program that asks the user for the number of hours worked this week and the hourly rate of pay. The program is to calculate the gross pay. If the number of hours worked is greater than 35, the extra hours are paid at 1.5 times the rate. The program should display an error message if the number of hours worked is not in the range 0 to 70.

## Logical Operators

In the above exercises you have written programs that use many **IF and ELSE** commands using a single **CONDITIONAL** statement. Using **nested IF** statements works well however it isn’t always the most efficient way of constructing the statement. Take for example the largest number program shown earlier:

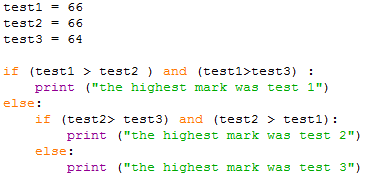


Although this program functions correctly there is a lot of duplication and unnecessary statements e.g., two print outputs are the same but copied twice for the different conditions. To tidy this, we can use **LOGICAL OPERATORS.** Python supports the following operators:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| and | Called Logical AND operator. If both the operands are true, then the condition becomes true. | (a and b) is true. |
| or | Called Logical OR Operator. If either or both true then the output is true. | (a or b) is true. |
| not | Called Logical NOT Operator. Use to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false. | not (a and b) is false. |

So, using these operators we can simplify the above program:

Two CONDTIONS joined by an **AND** operator. Both have to be **true** for the condition to return true.



We have replaced the two **nested IF** statements with one single **CONDTION** using the **AND** operator. Notice how each condition is put in brackets; although not required in Python it helps readability.

## “IF”…“Else If” statements

So far, we have looked at **IF, ELSE and LOGICAL** operators, which allows us to mix and match different approaches to testing conditions. One last approach is using a mixture of **IF, ELSE and ELIF statements.** In Python we can add what is known as an **“ELIF”** statement which is basically short for ELSE IF, that is instead of saying IF..ELSE we are saying IF condition above not met, then try another IF statement:

IF first condition **not met,** TRY another IF statement.

Text, letter

Description automatically generated

The above program will assign the integer value 20 to the variable test\_mark and then compare if that number is less than 20, if it isn’t then it will test if it is less than 30, and again less than 40, and if NONE of the conditions then print the last statement. So effectively we are going through each boundary until one is met or not at all.

This kind of layout is very useful if you have a sequence of values in **ORDER** that you wish to test. So, in the above we are working from values 20 and above. Notice there is NO ceiling in this program, which effectively means than **ANY** number 40 or above will give me the last statement.

### Tasks

2.13 Using logical operators (AND, OR) write a program that asks the user to enter 3 numbers and displays the largest.

2.14 Write a program which prompts the user for a percentage score and then displays the grade.

Consider using a mixture of IF…ELIF..ELSE statements for each grade boundary:

0 – 39 (E)

40-45 (D)

46-55 (C)

56-65 (B)

66-75 (A)

76- (A\*)

Ensure to check that the value is between 0 and 100 inclusive otherwise output an appropriate error message.

2.15 Write a program that asks the user for a month number and displays the number of days that month has.

2.16 Extend the program for task 2.12 to include leap years i.e., adapt to change days in the month for February. The program should prompt user for year and month. A year is a leap year if the year divides exactly by 4, but a century is not a leap year unless it is divisible by 400. For example, the year 2020 was a leap year, the year 1900 was not, and the year 2000 was a leap year.

2.17 Write a program that accepts a date as three separate integers such as 12, 5, 22 e.g. enter day, month and year.

The program should display the date in the form 12th May 2022.

For example, for the day the program will output either “st”, “nd”, “th”, “rd” after the number, the month number displayed as the full name, and the year outputted as a four-digit value.

No validation will be necessary e.g., the program won’t check if the number of days is allowed for the month specified.

# Lesson 3 - Iteration

|  |  |
| --- | --- |
| **Lesson Outcomes**  In this lesson you will learn:   * how to repeat parts of the same program. * to use fixed and conditional loops. * Using loops for validation. | **C:\Users\Graham\AppData\Local\Microsoft\Windows\Temporary Internet Files\Content.IE5\X6CHINOH\MC900441498[1].png** |

## Fixed Loops– iterating by sequence

In programming, loops are fundamental in producing solutions to problems. In tackling tasks loops provide the ability to repeat a sequence of commands repeatedly. A sequence is repeated either a fixed number of times or based on a condition. We are first going to look at fixed loops, or sometimes referred to as **definite loops**.

In Python you can create fixed loops in a variety of different ways. The most basic is looping through a sequence of values.

Let’s look at the following example:



The for command tells Python to create a **FIXED UNCONDITIONAL LOOP**. The range command tells Python to go through a sequence of numbers starting from the default 0 to the specified number (notice the program ends at number 9). The variable count is used to store the value in range as it repeats so in the example above (sometimes referred to as a **counter-controlled loop**):

Count = 0 : print (count) will output ‘0’

Count = 1 : print (count) will output ‘1’

Count = 2 : print (count) will output ‘2’

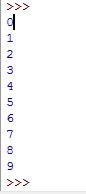
.

.

Count = 9 : print (count) will output ‘9’

Notice the countvalue stops at 9 – Python by default starts from 0, so there are 10 values.

Running this program, you should get the output:



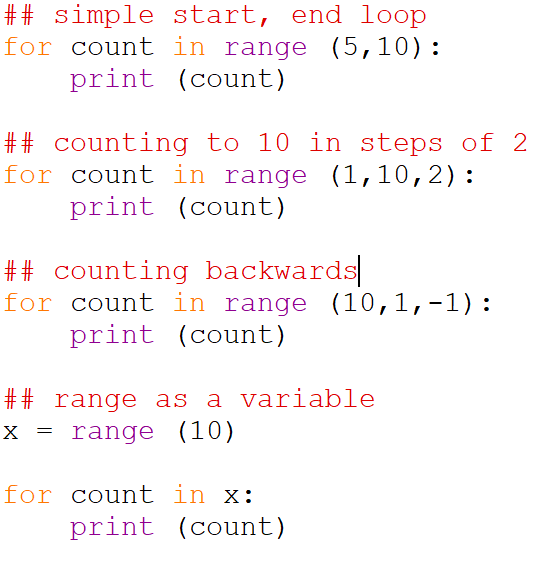
The range command is used to specify any range in the following format:

**range**([*start*], *stop* [, *step*])

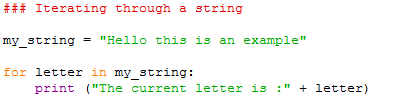
**[] brackets are optional**

Start is the staring value, stop is the end, and step is the incremental values which is 1 by default.

Examples:



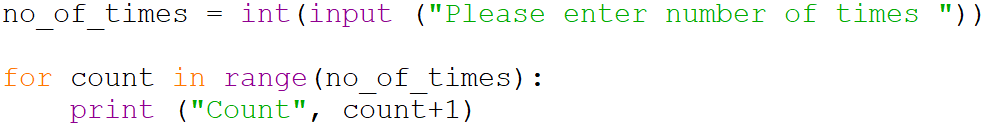
You can also use loop on lists and strings. For example, you may wish to iterate through a string in the following manner:



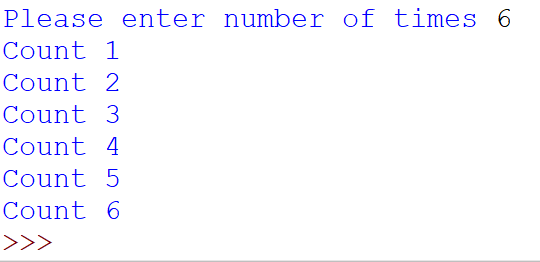
In the above program a loop is created to go through each character in the string. You should get the output:



You may also user input to specify the range:



Running will output:



### Tasks

3.1 Write a program that asks the user to enter a number. The program should output all the numbers starting from the user number to 0 e.g., count down from the user number entered.

3.2 Write a program to display an “n times table” for a given integer n. The user should enter a number, and the program will display the timetable from 1 to 12 for that given number. If the user enters 4, the output should be:

1 \* 4 = 4

2 \* 4 = 8

…

12 \* 4 = 48

3.3 Write a program that asks for a number and displays the squares of all the integers between 1 and this number inclusive. So, entering 4, the program would output 1,4,9,16 (12, 22, 32, 42).

### Extension

3.4 n factorial, usually written n!, is defined to be the product (multiplication) of all the integers in the range 1 to n:

N! = 1 \* 2 \* 3 \* 4 …………..\* n

Write a program that calculates n! for a given positive n.

## Loops within in loops

In programming it is often useful to have loops within loops, that is repeating a section of code n number of times by n2 number of times e.g.:

Text

Description automatically generated with low confidence

The above program will output all the times tables from one to twelve. A loop has been created at the beginning from one to 12 and then within that another loop from one to 12. Notice the indentation is used so Python knows which blocks of code are associated with which loops.

## Continue Nested Loop

The Continue statement forces the next iteration and ignores the remaining statements. For example:

A picture containing logo

Description automatically generated

Will produce the output:



The program will continue the loop each time the letter is equal to “j”, ignoring the print statement.

## One-Line nested loops

It is possible to create nested loops on the same line of code:

A picture containing diagram

Description automatically generated

Output creates a LIST (covered later):

### Tasks

3.5 Write a program that asks the user to enter the number of stars per row and the number of rows to be displayed. For example, entering 5 and 3 should display:

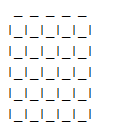
\*\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*

**HINT:** to print one character next to another without a new line is print(“\*”, end=””)

3.6 Write a program which programs the user for an x by grid. For example, given a 5 x 5 grid the program should display:



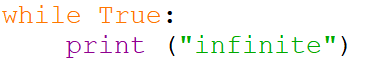
### Extension

3.7 Write a program to display all the prime numbers up to a user entered number. Remember a prime number is a number which can be divided evenly only by 1 or itself.

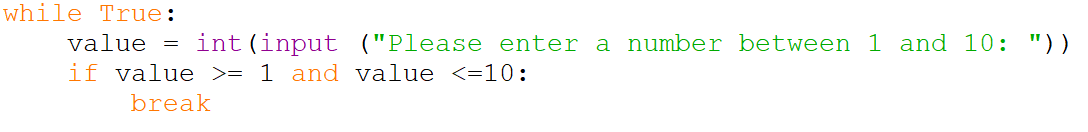
Infinite Loops

As the title suggest these loops which are defined as infinite i.e., they never end. These loops are normally intentional however there are occasions when they happen by accident causing the program to crash! Creating an infinite loop maybe necessary for certain types of control systems where constant monitoring is necessary and will only end through either an error, break or when program is terminated.

Example of infinite loop in Python:



The condition in the above while loop is set to True therefore will continually output the word “infinite”. We can use this approach to simulate a common repeat structure in many programming languages called **REPEAT … UNTIL**, where the condition for the loop is at the end as opposed to the beginning. This is a very useful approach when validating text input, as you would normally request input BEFORE testing the condition. For example:



Here we set up the infinite loop and we use the break command to exit the loop if the value is in the specified range.

## 

## Conditional loops

As the title suggest, a conditional loop repeats a sequence of commands until a condition is met, often referred to as **indefinite loops**. You can represent conditional iteration in a simple program flow chart:

A picture containing text, sign

Description automatically generated

The above program sets the variable **Count** to 1, outputs count, adds one and loops until Count is equal to 10. Notice how the Count variable is used to keep a track of the number of loops.

We can write this program in Python using the following code:

**Condition** e.g., keep looping while count is less than 10.

Text

Description automatically generated

Add one to count variable. Notice it is indented as part of the while statement

In the above program a while loop is used. The while loop will repeat all the indented block of code underneath until a **CONDITION** is met, in this case count < 10. This is called a **CONDTIONAL** loop i.e., the loop will continue until a specified condition is met.

Let’s look at another more complex example:

Text

Description automatically generated

**Line 1** takes a text input from the user converts it to a number using the int command and assigns the value to the variable no\_of\_times.

**Line 2** the variable counter is assigned to one.

**Line 3** sets up a while loop which will continue until counter is **less than or equal** to the variable no\_of\_times.

**Line 4** will print out the “Number” and add to it (concatenate) the contents of counter (using the **str** command to convert a number to TEXT)

**Line 5** Increment the counter by one.

You can also use a while loop with an **else** statement which will execute if the **WHILE** condition hasn’t been met. An example:

### Tasks

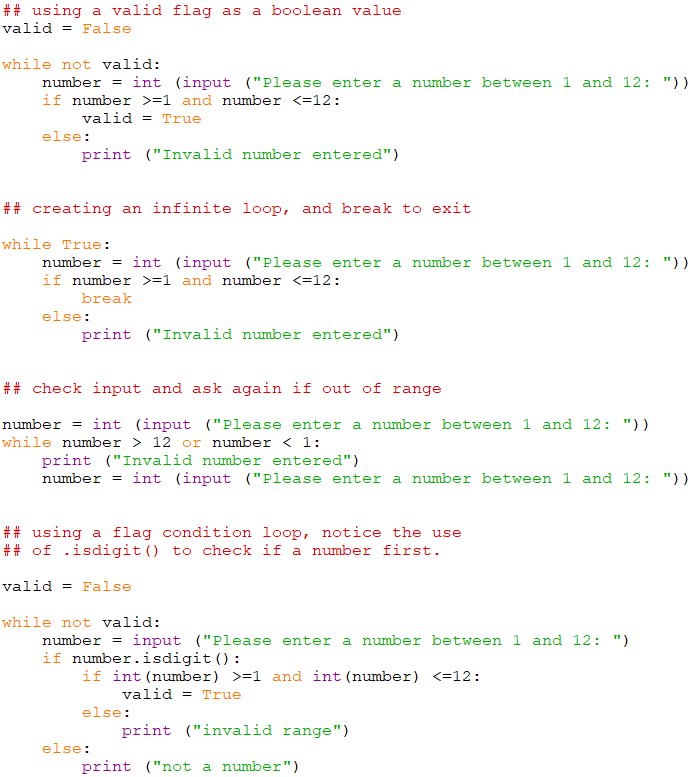
3.8 Write a program that displays the word “hello” on the screen 4 times using the while loop.

3.9 Write a program that prompts the user to enter a short message and the number of times it is to be displayed and then displays the message the required number of times.

3.10 Write a program that reads in a series of numbers and adds them up until the user enters zero.

## Validation

One of most common routines in programming is **validation**, that is checking user data entry is in a valid format. We can achieve this easily using conditional loops. There are many ways of doing this as demonstrated below:



All the above methods are an acceptable way of validating data inputted.

### Tasks

3.11 Write a program that asks the user for a number between 10 and 30 inclusive and will validate, that is test, the input. It should repeatedly ask the user for this number until the input is within the valid range.

**HINT** you will need to keep looping until a valid number is entered.

3.12 Expand your program from Exercise 3.4 to display the average as well as the sum of the numbers entered. Make sure you do not count the “0” value as entry.

3.13 Write a program that displays a conversion table for pounds to kilograms, ranging from 1 pound to a value the user enters (1 kg = 2.2 pounds). Include validation so the user has to enter a value greater than or equal to 10 and less than or equal to 100.

3.14 Write a program that asks the user to enter 8 integers and displays the largest. Include validation so the user must enter a positive integer (whole number), output an appropriate error depending on data entered.

**HINT:**

you will need to create a variable called LARGEST which you initialise to 0. This value will change according to the largest number entered. For example, if the user enters the value 9 and this is bigger than the LARGEST variable then you assign the value 9 to LARGEST. Use the .isdigit() to determine if a number has been entered.

3.15 Change the program in 3.14 to display the SMALLEST value.

### Extension

3.16 A Fibonacci sequence is the integer sequence of 0, 1, 1, 2, 3, 5, 8....

Write a program that will prompt the user for n terms e.g., up to what number in the sequence. The program should then output the Fibonacci sequence up to that number.

Include validation so the user cannot enter a number less than 2. An appropriate error message should be displayed, asking the user to re-enter the value.

# Lesson 4 - Functions, Strings and Dates

|  |  |
| --- | --- |
| **Lesson Outcomes**  In this lesson you will learn:   * how to use basic common functions. * Type Conversion functions. * to incorporate functions into your programs. * the format of STRINGS. * manipulating and using STRINGS. * how to use and manipulate dates using the date object. * Creating simple functions | **C:\Users\Graham\AppData\Local\Microsoft\Windows\Temporary Internet Files\Content.IE5\X6CHINOH\MC900441498[1].png** |

## 

## Introduction to Functions

When writing very large programs it is often necessary to split code into smaller chunks or what we call **subroutines**, which have a specific purpose. We can develop and test each sub-routine so that when the whole program is put together, we can be more confident it will work correctly. Also, we may call the subroutine multiple times within the same program without having to copy the same code.

Functions in any programming language are powerful tools and are a type of **subroutine**. Like most programming languages Python has many built-in functions for us to use. They allow us to perform common complex operations without having to write all the extra code. A function should have a **name** and with the option of adding **parameters** and a **return value**. For example, if we look at the common function **str** in Python:

**Output = str ( 1203 )**

The PARAMETER e.g. a number

The function name

The above function takes a number as an input and will return the number as a **STRING**. This returned value is assigned to the variable Output. Further examples of functions we use all the time:



Figure 1 function example

In Figure 1 function example the following functions are used: **float, input, print and str. “Float**” converts the text input to a decimal number, **“print”** outputs a string to the screen, **“str”** converts a number to a string and finally **“round”** takes a decimal number and rounds up to the given number of digits.

The common functions which you have already encountered include:

|  |  |
| --- | --- |
| float() | convert text into decimal numbers |
| int() | convert text into an integer |
| str() | converts a number into a string |
| input() | Return text input from the user |
| print() | output content in brackets to the screen. |
| range() | used to define a range of values. Mainly for looping. |

float, str and int are examples of **Type Conversion** functions or sometimes referred to as **Type Casting**. These functions convert one data type to another. Examples of their usage include:

Text

Description automatically generated

Figure 2 Type Conversion functions

## More built-in functions

Having now used many of the standard built-in functions, here are some other more commonly used functions which we are going to explore. Try typing them in Python console to see how they work:

|  |  |
| --- | --- |
| **Function** | **Description** |
| round() | round (value to round, [optional number of digits]) |
| len() | return the length of a string |
| reversed() |  |
| sum() | Adding values specified in brackets. For example, adding values in a list: |

### Tasks

4.1 Write a program that asks the user to type in a number with decimal places. The program should then display the rounded number to 2 decimal places.

4.2 Write a program that prompts the user for a sentence and then displays the number of characters.

## Additional Non-core Functions - Modules

Whilst Python has many useful built-in functions, sometimes it is useful to use other functions which aren’t part of the standard Python **modules**. What are **modules** you ask? **Modules** are Python files which contain additional functions or classes so that when imported allow your program to use. To import a module, you simply add the command **IMPORT** and the beginning of your program, stating the name of the module you want to import. Of course, there are many additional modules which you may use. Here are some of the more common ones:

* random – includes functions for generating a range of random numbers
* math – includes additional advanced maths functions
* turtle – uses TURTLE for drawing simple on-screen graphics
* numpy – you can download this and install in Python to give you extra functions for arrays / lists

As well as importing Python existing modules there also many which are often created on-line, available for downloading and using.

### RANDOM module

Looking at the **RANDOM** module we can create a whole range of different random numbers. When referring to a function within a module we normally type the module name followed by the function using the “.” character. For example, **random.randint(0,10)**

|  |  |
| --- | --- |
| **Function** | **Description** |
| .random()  and .randint (from, to) | Graphical user interface, text, application  Description automatically generated |

|  |  |
| --- | --- |
| .seed() | The seed() method is used to initialise the random number generator. Random programs require an initial number from which to generate a random value. This is normally by default the current system time. Using the seed you can specify the starting number. |
| .randrange() | The random range method returns a random selected element from the range specified (start, stop , step).    The above example will pick a random number from list (1,3,5,7, 9). |
| .choice() | This method will return a randomly selected element from a non-empty sequence. |

For a full reference of RANDOM functions (<https://docs.python.org/3/library/random.html>).

### Import methods

Sometimes when we import modules, we don’t necessarily need every function and class that comes with it. We can just import that single function using the from command, as follows:

A picture containing logo

Description automatically generated

We can now refer to this function as pi as opposed to math.pi. Alternatively, we could import all the methods and functions into our program, as opposed to the classes. This means you can refer to the functions by *name* (Figure 3) as opposed to the *module.name.*

Text

Description automatically generated

Figure 3 import ALL maths module functions

### Tasks

4.3 Write a program which outputs a random decimal between 0 and 1.

4.4 Write a program which outputs a random **odd number** between 50 and 100.

4.5 Write a program that will display random numbers between 1 and 6 until a six is generated **HINT:** you will need to use a conditional loop.

4.6 Write a program that will display six random numbers between 5 and 10.

4.7 Write a game in which the user guesses what number between 1 and 1000 the computer has “thought of”, until he or she has found the correct number. As the user enters a guess the program will display whether their guess is too high, too low or correct. If correct number chosen the program displays “You have guessed correctly” and finishes.

## Strings – a data structure

In the above exercises you used some useful functions for numbers and Strings. However, Python has some extra functions and methods which are unique to **Strings** and are used extensively for manipulating and extracting content. Before we can address these functions, we must understand the nature of a **String.**

You can imagine a string as a series of characters with each character given an **INDEX** number starting from 0. Given the example:



This would be represented in the computer’s memory as:

SPACE

Index Number

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| **G** | **r** | **a** | **h** | **a** | **m** |  | **B** | **r** | **a** | **d** | **s** | **h** | **a** | **w** |
| -15 | -14 | -13 | -12 | -11 | -10 | -9 | -8 | -7 | -6 | -5 | -4 | -3 | -2 | -1 |

Figure 4 Name String

Each character stored as an **ASCII value**, with each given an **INDEX** position (Python uses what we call **ZERO indexin**g i.e., the first character is index 0).

## String slicing

The position of a character within a string is referred to using the index number placed in **[] brackets** after the string variable name. We can also specify ranges and steps using the following format:

**String\_variable\_name[start: stop: step]**

Stop and Step values are optional. For example, if you just specify just one number then a single character at that index position is outputted.



Outputs:



If we wanted to extract the surname and first name from the string in **Figure 4**, we would have to perform a **SLICE**:

A picture containing graphical user interface

Description automatically generated

Here surname and first name are extracted from the name string. Surname starts at character **index position 6** to the end of the string (hence why left blank meaning got to the end of the string], first name **starts at 0** and **stops at 6** (Python does not include the last number e.g., stops at 6):

A picture containing graphical user interface

Description automatically generated

Alternatively, you can count from the end backwards to get the first name.



First and last characters from the string:

Text

Description automatically generated with low confidence

As mentioned above, not only can we specify the start and end of the string we can also specify the steps. This is particularly useful if you want to extract characters from a string at set intervals. For example, if we store a sequence of numbers in a string and wanted to extract all their odd positions:

Text

Description automatically generated with medium confidence

This will output each character at intervals of 2 starting from position 0 until the end (hence nothing between the first and second colon):



Of course, as, when entering a person’s name into a program then the number of characters won’t always be the same. For this we need to use the function split(), which simply splits the string based on a single character, in this case a SPACE, and inserted into a LIST (covered later). For example:

Text

Description automatically generated with medium confidence

**LINE 1:** split the string whenever a “ “ (SPACE) is found and create a LIST, with each element of the list indexed starting from 0.

**LINE 2:** print name[0], outputs first element

**LINE 3:** print name[1], outputs second element

## Useful String recap

Going back over what has been covered, we can review all the ways strings can be used:

Text

Description automatically generated

### TASK

4.8 Try typing and running the above programs to see what they do.

## String Methods

All strings in Python are **Objects** (refer to OO programming in later tutorials), and with Objects you have **METHODS** e.g., functions which perform operations on the object. Strings have many methods.

Here are some of the most useful methods to use, supported by examples:

|  |  |
| --- | --- |
| .upper() | This method will turn the given string into uppercase:    Outputs: |
| .lower() | Outputs: |
| .count(string to count) | Will return the value “2”, as the word “to” appears twice in the string. |
| .index() and .rindex() |  |
| .capitalize() | Will capitalize the first letter of the String. |

There are many METHODS, will the full index here:

<https://www.w3schools.com/python/python_ref_string.asp>

### Tasks

4.9 Write a program which asks the user for their surname and forename. The program will output their full name with both surname and forename in lower case and capitalised.

4.10 Write a program that reads in a date of birth in the format **dd/mm/yyyy** and displays the year of their birth. **HINT:** you need the last 4 digits

4.11 Write a program that asks the user for their first name and surname. The program then displays the full name - surname all uppercase followed by first name in lower case with first letter capitalised. Try to complete without using the .capitalize() method.

4.12 Write a program that asks the user to enter a sentence, terminated by a full stop. The program should count the number of words and display the result.

**HINT:** create a loop which looks at the string character by character, keeping a running total of words. The word count is incremented when a space is detected.

4.13 Write a palindrome tester. A palindrome is a word or sentence that reads the same backwards as forwards. The user should enter a string and the program should display whether the string is a palindrome or not. So, for example if you enter “RACECAR”, this is spelt the same backwards as forward so should output “is a palindrome”.

## Formatting Strings

Python provides two main methods to allow you to format the output of your string, this is particularly useful when you want to produce layouts or tables in your outputted data. One method is using the older formatting operator “%” (Python 2), or the other is to use the newer .format method (Python 3).

### Format operator (old style)

Works by replacing placeholders (where the % character appears in the text) within the string with values, for example:

A picture containing icon

Description automatically generated

The “%” is the format operator. In the above the place holders are “%d” i.e., replace these values with the integers 300 and 400.

The general structure is (after the % sign):

%[key][flags][width][.precision][length type]code

The [] indicate OPTIONAL values, followed by code. The optional values being:

|  |  |
| --- | --- |
| *Key* | Refers to a key in a dictionary “()”: |
| *flags* | Used to indicate zero fills **(0)** which fills preceding blank spaces within the field with 0 and optional justification within the given field width: **+** for right-justification or **-** for left-justification. Refer to ***width*** to compliment this.    Print “0” fill blanks for 10 digits of type (d) – integer/ decimal |

|  |  |
| --- | --- |
| *width* | An integer value indicating the number of spaces in the current field    10 characters, hello aligned by default to the right. |
| *precision* | The number of digits to be printed after the decimal place when printing a real value.    10 digits, blanks filled with 0, “d” for decimal/integer and “.” 2 decimal places. |
| *code* | One of the format codes (see previous examples of use):   |  |  | | --- | --- | | **Code** | **Description** | | %s | String (or any object) | | %c | Character (from an ASCII value) | | %d | Decimal or integer value | | %i | Integer value (same as %d) | | %u | Unsigned integer | | %o | Octal integer | | %x | Hexadecimal integer | | %X | Same as %x but uppercase | | %e | Floating-point with exponent (decimal numbers) | | %E | Same as %e but uppercase | | %f | Floating-point no exponent (decimal numbers) | | %g | Same as %e or %f | | %G | Same as %g but uppercase | | %% | Prints a literal % | |

Whilst this looks fairly complicated it is relatively straightforward. Just remember the placeholders are codes which can be any of the ones listed above.

Examples:

Graphical user interface, text, application

Description automatically generated

Using in a typical program:

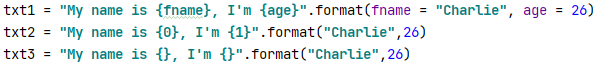
Text

Description automatically generated

The first placed holder is a simple string (%s) and the other values (%8.2f) are displayed in decimal, with 8-digits, fixed decimal (the ‘f’ indicating fixed decimal) of 2 decimal places

### Format method (new style)

Python also provides the more recent format method, which you may use on any STRING:



Outputting txt1, txt2, txt3:

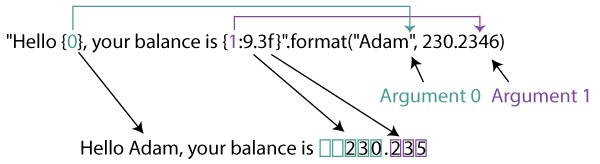
Text

Description automatically generated

Inside the placeholders “{}” you can add a formatting type after the first argument “:” to form the result. The “:” operator follows the sequence:

[[fill][align][sign][#][0][width][,][.precision][type]

**Example:**



So, in the above example:

:9.3f – 9 digits, 3 decimal places, fixed point number.

For a full reference of types (after “:”) refer to:

<https://python-reference.readthedocs.io/en/latest/docs/functions/format.html>

#### Formatting Summary

**[[fill][align][sign][#][0][width][,][.precision][type]**

|  |  |
| --- | --- |
| **Align** | |
| < | Left Align |
| > | Right Align |
| = | Forces padding to be placed after sign. Only available for numeric types |
| ^ | Forces the field to be centered within the available space |
| **Sign** | |
| + | indicates that a sign should be used for both positive as well as negative numbers. |
| - | indicates that a sign should be used only for negative numbers (this is the default behaviour). |
| “” (space) | indicates that a leading space should be used on positive numbers, and a minus sign on negative numbers. |
| # | The ‘#’ option is only valid for integers, and only for binary, octal, or hexadecimal output. If present, it specifies that the output will be prefixed by ‘0b’, ‘0o’, or ‘0x’, respectively. |
| , | The ‘,’ option signals the use of a comma for a thousands separator. For a locale aware separator, use the ‘n’ integer presentation type instead. |
| **Width** | |
| Decimal integer | *width* is a decimal integer defining the minimum field width. |
| **Precision** | |
| e.g “.2” , 2 decimal places | precision is a decimal number indicating how many digits should be displayed after the decimal point for a floating-point value formatted with ‘f’ and ‘F |
| **Type** |  |
| ‘e’ or ‘E’ | Scientific format (lower, upper case) |
| S or NONE | String / Character |
| ‘b’ | Binary format |
| ‘d’ | Decimal format. |
| ‘f’ or ‘F’ | Decimal number, with max 6 digits after the decimal point. Lower and Upper case |
| ‘x’, “X” | Hex format, lower and upper case. |
| ‘n’ | Number format. |
| ‘c’ | Converts the value into the corresponding Unicode character (ASCII). |
| ‘%’ | Percentage. Multiplies the number by 100 and displays in fixed (‘f’) format, followed by a percent sign. |

### Tasks

4.14 Write a program which prompts the user for a number between 1 and 12, and then displays the times table to 12 of that number e.g., user enters 2, so display the 2s times table. The table should be formatted using the appropriate formatting methods with one column for the times table numbers and the other column for the answers. Example:

Table

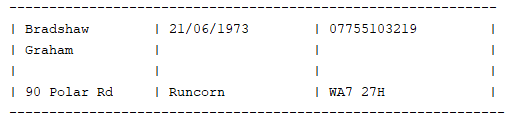
Description automatically generated

Notice the values in the first column are **ALIGNED** to the right, so when double digits the spacing doesn’t change.

|  |  |  |
| --- | --- | --- |
| Surname:  Forename:  Address: | Dob:  Town/ City: | Tel: No:  Postcode: |

4.15 Write a program which ask for their surname, first name, DOB, tel. no and address. Once the data has been entered display in the following format (us – and| for basic table formatting):

Example output:



## Dates

As you have seen from previous exercises, the ability to process strings is important. As well as these built-in functions sometimes, as mentioned, it is useful to import MODULES. One such module is **DateTime** for processing and manipulating dates. As part of the **DateTime** module there are different objects and methods:

* time – Time independent of day. *(Hour, Minute, Second, Microsecond)*
* datetime – Combination of date and time. (Month, Day, Year, Hour, Second, Microsecond)
* timedelta – A duration of time used for manipulating dates
* tzinfo – An abstract class for dealing with timezones

The following examples to display the current date and time:

Text, letter

Description automatically generated

This should produce the following outputs:

Text

Description automatically generated

To extract data and time you can use the ***strftime*** method as part of the time import. This is like formatting text using codes to display different parts of date/time.

Text

Description automatically generatedTo format the date output you can use the following codes:

|  |  |
| --- | --- |
| Directive | Meaning |
| %a | Weekday name. |
| %A | Full weekday name. |
| %b | Abbreviated month name. |
| %B | Full month name. |
| %c | Appropriate date and time representation. |
| %d | Day of the month as a decimal number [01,31]. |
| %H | Hour (24-hour clock) as a decimal number [00,23]. |
| %I | Hour (12-hour clock) as a decimal number [01,12]. |
| %j | Day of the year as a decimal number [001,366]. |
| %m | Month as a decimal number [01,12]. |
| %M | Minute as a decimal number [00,59]. |
| %p | Equivalent of either AM or PM. |
| %S | Second as a decimal number [00,61]. |
| %U | Week number of the year (Sunday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Sunday are considered to be in week 0. |
| %w | Weekday as a decimal number [0(Sunday),6]. |
| %W | Week number of the year (Monday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Monday are considered to be in week 0. |
| %x | Appropriate date representation. |
| %X | Apropriate time representation. |
| %y | Year without century as a decimal number [00,99]. |
| %Y | Year with century as a decimal number. |
| %Z | Time zone name (no characters if no time zone exists). |
| %% | A literal '%' character. |

Sometimes it is useful to store parts of the Date/ Time into variables so they can be used in calculations or for formatting purposes:

A picture containing text

Description automatically generated

## Manipulating Dates

You can also use the date/time object to create your own dates e.g.:

Logo, company name

Description automatically generated

Using this method allows you to manipulate dates using simple formula e.g.,

Text

Description automatically generated

Should output:

Let’s add 3 days to a date object.

|  |  |
| --- | --- |
|  |  |

This will output:

|  |  |
| --- | --- |
|  |  |

### Tasks

4.16 Write a program that displays today’s date in the format dd/mm/yyyy.

4.17 Write a program that reads in a date of birth in the format dd/mm/yyyy and displays the user’s age.

4.18 Write a program that reads in a date, converts into date format, adds a day, and then displays the next day’s date.

## Creating Functions

So far you have covered Python in-built functions. For programs to be efficient and structured it is essential to create and use your own Functions. Using functions allows re-usability of code, the grouping of tasks and provides program structure to help with debugging and testing.

To define a function, we use the def command following by the name and parenthesis:

A picture containing logo

Description automatically generated

We can run this function by simply calling my\_function():

A picture containing graphical user interface

Description automatically generated

You can call this function as many times as you like in the code; in other words, the same code is reusable. You must note to call a function you have to define it first before the call. As Python is an INTERPRETED language, it runs one line of code at a time, and if you call it before the function has been defined you will get an error:

Logo

Description automatically generated with medium confidence

That why it is good practice to define all your functions at the beginning of the program, and then call them from the bottom of the code.

### Tasks

4.19 Create a function called today() which will display the current date and time in the format dd/mm/yyyy, followed by hh:mm:ss. Call the function in the main body of the program.

4.20 Create a program which allows the user to select either multiply numbers or add from a menu. If they select **“add”** they are prompted for 5 numbers and the sum of those numbers is display, the user is returned to the main menu. If they select **“multiply”,** they are prompted to enter two numbers and the program display the multiplication result, they are then returned to the main menu. They may repeat this process until they select “x” to end the program

Example of menu structure:

Text

Description automatically generated

To construct this program, create three functions with the appropriate code:

Graphical user interface, text, application

Description automatically generated

In the main part of the program call the menu function to display the menu.

Prompt the user to select 1, 2 or 3. Call the appropriate function after selection e.g., selecting 1 will call the function add\_numbers(). Validate user menu selection, displaying an appropriate error for incorrect selection.

Example pseudo code:

selection = “”

WHILE selection NOT EQUAL “3” THEN

CALL menu()

selection = INPUT(“Please enter a value 1 to 3: “)

IF selection = “1” THEN

CALL add\_numbers()

ELSE IF selection = “2” THEN

CALL multiply\_numbers()

ELSE IF selection = “3” THEN

OUTPUT “End of program”

ELSE

OUTPUT “Incorrect entry”

## Function Parameters

When calling functions, we can also specify parameters which are values, or the contents of variables passed to the function. This can be illustrated creating a simple division function which will take two values and output the result:

A picture containing chart

Description automatically generated

Figure 5 division function

As demonstrated in figure 5 we have created a common function which can be called many times within our program using different values or variables each time. We could also adapt the program to display an error if we attempt to divide by 0:

Graphical user interface, text

Description automatically generated

Figure 6 divide function with error checking

### Tasks

4.21 Create a multiplication function which given three values, will display the multiplied results. For example, multiply(3,4,2) will output the result 24.

4.22 Prompt the user to enter two numbers. Create a function called biggest(val1, val2) that given two numbers will display the largest, or if equal output “they are equal”. Use the numbers the user has just entered as parameters to the function.

4.23 Create an is\_valid() function which give 3 parameters, number, min, max values will out “valid” or “invalid”. So for example if you call is\_valid(20,0,10) “invalid” is outputted as the value 20 is not between the values 0 and 10.

4.24 Use the function created in 4.23 in a program which outputs “invalid” or “invalid” when the user is prompted to enter a number between the values 50 and 100.

## Function Returns

When you call a function, a value is always returned even if null/ contains nothing. When we say returns a value, we mean a value is returned to where the function was called. To illustrate this, take the following example using a minus function:

Text, letter

Description automatically generated

Value is returned back to expression

To return a value we have used the command return followed by the value or variable we want to return.

### Tasks

4.25 Adapt the program in task 4.20 so the multiplication function returns the result. Use this function in a main program which asks the user for three numbers and displays the multiplication result.

4.26 Adapt the program in task 4.23 so rather than outputting “VALID” or “INVALID” the function returns a BOOLEAN value based on if the value given is valid.

Create a main program which prompts a user to enter a number between 0 and 10. Validate the entry using the adapted function above, so the user is prompted again if an incorrect value entered.

4.27 Write a function odd(num) which given a value will return whether the number parameter is odd returning TRUE if odd. Use the function in a main program to output whether an integer value entered by a user is an ODD or EVEN number.

# Lesson 5 - Lists, Tuples, and Dictionaries

|  |  |
| --- | --- |
| **Lesson Outcomes**  In this lesson you will learn:   * how to use basic Lists. * process and manage lists. * how to incorporate Lists into your programs. * what a TUPLE is. * how to initialise and use dictionaries * the setting up and processing of SETS. | **C:\Users\Graham\AppData\Local\Microsoft\Windows\Temporary Internet Files\Content.IE5\X6CHINOH\MC900441498[1].png** |

## Introduction to Lists

Lists are what we call **DATA STRUCTURES**. A data structure is way of storing data in some ordered fashion so that we may extract and process it easily. The most common type of data structures is an **ARRAY**, which is implemented in many different languages. These are probably one of the most useful data structures used in programming.

In Python Arrays don’t exist in the core modules, instead **LISTS** are used. They are also **Mutable**, meaning once defined they can be changed, hence why they are often referred to as **dynamic data structures**. They can expand or decrease in size throughout the life of a program. To create a simple list of numbers stored in sequence you would create the list and assign to a variable name:



The brackets **[]** are used to define the List, with each value inserted separated by a comma. The List is assigned to the variable sampleList. Once the list has been created, we can process it in a variety of different ways. For example, we may extract single elements from the list by referring to the list index number.



So, in the above example, by printing sampleList at **index position [3]** we get the number **4** (the index starts from 0). As data is now sequenced and indexed, we can refer to elements easily.

Another example could be storing months of the year. As we know there is an order sequence for each month:

INDEX

DATA

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| Jan | Feb | Mar | Apr | May | Jun | Jul | Aug | Sep | Oct | Nov | Dec |

In Python this is initialised as:



The above stores all the months in the year, starting with index **0** as “Jan”. You will also notice that each element is this case a **STRING.** You can now find out what the month name is by simply referring to the index number:



Will output the month **“Jul”**, as the index **starts at 0.**

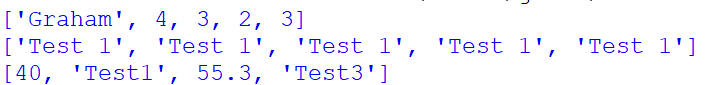
## List Examples

Lists can be used to store a whole array of either objects or values. Here are some examples:

Text, letter

Description automatically generated

Outputs:



Notice in list 2 we have used the variable x which stores a String, this is then created multiple times using the arithmetic operator “\*”, creating a list of five string values.

### Tasks

5.1 Write program that stores all the months of the year in a List. The program should prompt the user for a month between 1 and 12, and then output the month name.

5.2 Write a program that stores the first 10 numbers in the Fibonacci sequence in a LIST. The program should test the user’s knowledge of the sequence by prompting them to enter a random nth number (1 to 10) in the sequence and display whether they are correct or not (the program will need to refer to the List).

## Iterating through LISTS

One of the most useful aspects of LISTS is the ability to Iterate, that is loop, through each element. This could be for used for several purposes such as searching for a data item, totaling each value, or finding the position of an item. Python provides us with many methods in which to easily do this (covered later), but for now the focus is on how LISTS are structured and processed using a **For..Loop**:

A picture containing chart

Description automatically generated

**LINE 1 &2:** Initialise months LIST

**LINE 3:** Go through each element in the list and assign value to the variable month.

**LINE 4:** Output the contents of variable month.

Running the above program will produce the below output:



#### Alternative methods:

Some programmers prefer to use more accepted cross-language methods for iterating through a list by creating an INDEX value which is incremented after each iteration. This method is most commonly represented in flowcharts and pseudo code. The advantage of which allows you not only to examine each element of the list but also keep a track of the index value.

Text

Description automatically generated

**LINE 3 & 4:** Initialise months LIST

**LINE 3:** Create a fixed loop starting from 0 to 11. len functions returns number of elements in a LIST, in this case 12.

**LINE 4:** output the contents of variable month

Again, we could perform the same using a **while..loop:**

Text

Description automatically generated

## Initialising Lists

When creating lists sometimes you want to create default values for a fixed number of elements e.g., keeping a tally vote for candidates in an election.

Candidates in an election:

A picture containing diagram

Description automatically generated

Output (the initialised votes list):



**LINE 2:** here we create a fixed loop going through each element in candidates list.



Each iteration assigning the **value 0** to the list.

### Tasks

5.3 Using the example program above, add functionality to allow 20 votes to be cast, increasing the tally as a candidate receives a vote. Display the list of candidates, numbering each 1 to 5, and prompt the user to enter a valid range (1 to 5) to cast their vote. The program should then output who wins the election.

**HINT:** To output the election winner, you may remember in earlier exercises finding the largest value in a set of values.

5.4 Using the example code below:



Change for EVEN numbers. Prompt the user to enter a number between 1 and a 100, and using the LIST, to determine if the number entered is ODD or EVEN.

Text

Description automatically generated **HINT:** you can check if a number exists in a LIST using the IF statement in the following manner:

**Extension**

Could you adapt the above program to store a list of PRIME numbers and then check if the number entered in by the user is a PRIME number?

## Processing Lists

One of the most useful things about Python Lists is the common methods provided which allow you to easily process and manipulate the stored data. Here are some commonly used methods:

|  |  |
| --- | --- |
| **Method** | **Description** |
| append (value) | adds elements to end of the list |
| count(“x”) | counts the number of occurrences of x |
| index(“x”) | returns the index of “x” in the list |
| insert(y,”x”) | inserts “x” at location y |
| pop() | returns last element then removes it |
| remove(‘X’) | finds and removes first ‘x’ from list |
| sort() | sorts the list in ascending order. |
| copy() | Returns a copy of the LIST |
| extend() | Add elements of a list, the end of current list. |
| clear() | removes all the elements from the list |

**EXAMPLES:**



A picture containing diagram

Description automatically generated

Output:











A picture containing diagram

Description automatically generated



A picture containing chart

Description automatically generated



A picture containing text

Description automatically generated



Company name

Description automatically generated with medium confidence



Text

Description automatically generated



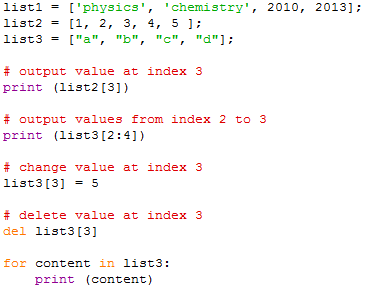
Text

Description automatically generatedText

Description automatically generated with low confidence



## List indexes

As well as using the above Lists methods you can also refer to list values through the indexes. For example:

Using the optional **“:”** in the format **from : to : steps**

Text, letter

Description automatically generated

Outputs:

A picture containing text

Description automatically generated

More on lists, visit <http://www.tutorialspoint.com/python/python_lists.htm>

### Tasks

5.5 Write a program that reads 6 names into a List. The program must display the names in the same order that they were entered and then in reverse order.

**HINT:** you can use the list method “.reverse()”

5.6 We want to simulate throwing a die 30 times and record the scores. If we did this “manually” we would end up with a tally chart. If we use a computer to keep a count of how many times each number was thrown, we could use an integer list:

tallychart = [0,0,0,0,0,0]

In general, a die throw will give a scorei, and we want to increment the count in the ith element.

talllychart[i] = tallychart[i] + 1

Write a program to simulate the throwing of a die 30 times. The results of the simulation should be printed as a table of scores (1 to 6) and frequencies.

5.7 **National Lottery Numbers**

We wish to select six random numbers between 1 and 49 with the condition that all numbers are different. One possible strategy, or algorithm, is:

*Initialise an empty lottery list ready to store six random numbers*

*Generate a random number between 1 and 49*

*If the number has been chosen before in the lottery list then generate again.*

*Display this value*

*Add number to the list*

*Repeat the above four steps until six numbers have been selected.*

Write a program to select six unique random numbers between 1 and 49.

## Searching – linear search

Often when we create a list, we need to be able to search items within it. One of the most common methods of searching is called a linear search. The principal idea of a linear search is from the beginning of the list, take each item in turn and compare with the value you are looking for, keep going until found or you reach the end of the list.

In Python, a linear search can be achieved using a while loop:

Graphical user interface, text, application, email

Description automatically generated

### TASKS

5.8 Create two lists and to store the name of students and their dates of birth. For example if Fred is born on 22/12/2004, then we could store “Fred” in Student [0] and “22/12/2004” in Do[0]. To find a particular student we can use the “.index” method to search a name.

Use the two lists to store 6 names with their corresponding DOB.

Write a program that asks the user for a NAME to search in the list and return their DOB.

5.9 Change the program in 5.8 to use a **linear search** as opposed to the “.index” method.

## Bubble Sort

Text

Description automatically generated

Notice the random module has been **imported** to use the shuffle method for the list of numbers. A swap flag has been used to determine when the loop ends e.g., when no more swaps have occurred. When iterating through the list, we have used the TEMP variable to temporarily store the swapped value.

This algorithm could further be improved by reducing the for..loop after each run-through e.g. no need to sort to the end each time as the next highest number has already been shuffled to the end.

After program executed:

Icon, polygon

Description automatically generated

### Tasks

5.10 Type in and run the above BUBBLE sort program. Change the program so the user can enter 10 numbers into the list. The list should then be sorted and outputted in descending order e.g., from highest to lowest.

5.11 Change the program in 5.14 so that the user enters 10 random words which are then displayed in Alphabetical order.

## 2-Dimensional Lists

Sometimes in programming it is necessary to store data in 2-dimensional lists i.e., Lists within lists. This is very useful when having many different sets of related data. So, for example, say we wished to store student marks in a series of tests.

|  |  |  |  |
| --- | --- | --- | --- |
| **Student** | **Test 1** | **Test 2** | **Test 3** |
| Li | 89 | 94 | 75 |
| Marcus | 50 | 65 | 48 |
| Jane | 23 | 65 | 53 |
| Beatha | 78 | 67 | 89 |
| Ciarrai | 67 | 58 | 98 |

Rather storing this data in one LIST, we can create SUB Lists in the following manner:

Text

Description automatically generated

You can see from the above each student has their own sub-list, with each Test mark in order of sequence. Using this List we could display the data similar to the table above:

Text

Description automatically generated

Outputs:

A picture containing table

Description automatically generated

### Tasks

5.12 Change the program you created in task 5.8, so that the student’s name and DOB are stored in a 2-dimensional list. For example, you can create a list of 6 elements, with each element containing a sub list of name and DOB e.g., storing 2 students:



You will need to amend how you locate the name in sub-List, to get the corresponding index value for the date of birth.

**HINT:** use a linear search like in 5.6 to find the name in each sub-list.

5.13Create a 2-D list containing a list of football team and their corresponding wins, draws, losses, goals scored, goals against (you can make up a league of 6 teams).

Display the league table with teams in order of points (descending order), showing total points and goal difference (don’t worry if two teams have the same points, although this could be an extension task to sort on goal difference).

Points calculates as follows: 3 for a Win, 1 for a draw.

**HINT: Refer back to a bubble sort algorithm.**

## Further Processing

Whilst we have discussed lists and worked through some examples, there are still little routines or advance concepts we can use to fully utilise lists.

### Slicing lists

One useful thing we can do in lists is slicing, which entails creating slice references to an existing list.

In this example we have a small deck of cards represented in a list with first digit as the card number followed by the suite e.g., “D” for diamond, “C” for clubs, “S” for spades, “H” for hearts:



We can use the slice class (refer to OO in later exercises) to create an object which can store the start, stop, and stop points which can be used when referring to a list. For example:

Text

Description automatically generated with low confidence

Outputs:

Text

Description automatically generated with low confidence

Notice, deck1 and deck2 aren’t lists but merely storing the start, stop and step points when referencing a list, in this case the deck of cards. Notice the original list is still intact. We have now created two list outputs for two decks.

### Indexing through a list

Sometimes it is necessary when iterating through a list to keep track of the index, that is the index of where we are in the list. So, for example, say we have a list of numbers and we wanted to know the index number of that list:

A picture containing text

Description automatically generated

This will produce the following output:

Icon

Description automatically generated

First number is the index position, followed by the number stored in the list.

### Common List Maths Functions

To help reduce complex code, Python includes some nice little mathematical functions which can be applied to lists:



|  |  |
| --- | --- |
| Function | Example |
| min | 2 |
| sum | 74 |
| max | 21 |

### Tasks

5.14 Write a program which creates a LIST which stores ten randomly generated numbers between 0 and 100. Using SLICING output all the numbers in **ODD** index positions, followed by another output of all the numbers in **EVEN** index positions.

5.15 Write a program which prompts the user for five student names. Record each name in a list along with a three-digit identifier. The program should then output the names along with a three-digit identifier.

The three-digit id should be created using the 1st digit to store the index position of the name, and the other 2 digits a random number between 0-9. For example, using the stored lists, the final output could be:

089 John Stevens

129 Aiguo Freeman

230 Claire Hanson

353 Xia Jones

449 Karen Moores

## Tuples

Tuples are like lists in that they can store a collection of ordered items in a single variable. They are defined using round brackets and, unlike lists, are unchangeable i.e., **immutable.** Why would you use a TUPLE? Well, sometimes you may need to create a list which is unchangeable, and when using DICTIONAIRIES (covered later) it is requirement that a component is a TUPLE instead of a list. Also, PYTHON will execute TUPLES much faster than normal LISTS.

Example of a TUPLE:

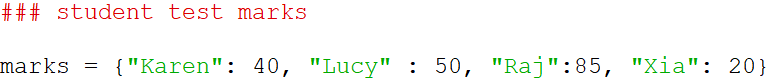
Text, letter

Description automatically generated

In this example, seasons lists have been defined as a TUPLE as this does not change. Whilst, once defined, we can’t change a TUPLE, we can add to other tuples to form one, in this example the YEAR.

## DICTIONARIES

Dictionaries are a very powerful ways of storing large data files so that it can easily be searched using a “KEY” or INDEX to retrieve the date. Extracting data from a dictionary is far faster and direct than Lists. Like an actual dictionary, you look up the word and find the corresponding definition. To setup up a dictionary in Python we use curly brackets **{}**, and insert a **key** followed by a **value**:



In this example we are storing student marks in a test. They **key value** is their name, followed by the **data** which is the mark. Notice we separate the KEY from the DATA using a colon “:”. Once setup, it is easy to retrieve data simply by referring to the key:

A picture containing text

Description automatically generated

The above code will search the DICTIONARY using the key name as specified by the user – notice the key name specified in between square brackets []. If the name doesn’t exist in the dictionary, then an error is returned. You may also use the **.get** method, as follows in this acronym lookup program:

Text

Description automatically generated with medium confidence

Another example here demonstrates how we can use a dictionary to store the availability of rooms in a in a hotel:

Text

Description automatically generated

Here we setup a dictionary with room numbers as the **key**. For each room we set to a **Boolean** value as either **True** for available or **False** for not available. The user requests a room no and the program will check whether the room exists and then if it is available or not. Notice we use the **in** command to check if the number exists in the set of dictionary keys.

### Tasks

5.16 Create a dictionary of student birthdays, so that given a student name the program will return their birth date.

5.17 Create a dictionary which given a month will tell you how many days in that month. Include validation

5.18 Write a program which ask the user for a text date in the format “dd/mm/yyyy” i.e., two-digit day, followed by a month, followed by four-digit year. You program should first extract the relevant day, month and year from the STRING and then validate. The basic validation should check:

* correct month number
* correct day range in given month
* correct year range 1900 onwards

If invalid date the program should output an appropriate error message and prompt the user to enter again. (Ignore the other validations you can do like checking the correct format, future date etc. – could be an extension task)

5.19 Extend your program in 5.18 to check date is valid in a given leap year.

To check for leap year create a function (refer back to functions in previous lesson) called leap\_check() which give the parameter year will return True if a leap year. Include the function call in your main program.

To check if a year is a leap year, **divide the year by 4**. If it is fully divisible by 4, it is a leap year. For example, the year 2016 is divisible 4, so it is a leap year, whereas 2015 is not. However, Century years like 300, 700, 1900, 2000 need to be divided by 400 to check whether they are leap years or not.

### Dictionary Methods

As well as storing and retrieving data from dictionaries using KEY-Pairs, Python also provides several methods, similar to Lists, to manipulate and edit set dictionaries.

|  |  |
| --- | --- |
| **Method** | **Description** |
| get() | Returns the value of the specified key |
| clear() | Removes all contents from a dictionary |
| copy() | Returns a copy of the dictionary |
| fromkeys() | Returns a dictionary with the specified keys and value |
| items() | Returns a list containing a tuple for each key value pair. |
| keys() | Returns a list containing dictionary’s keys. |
| pop() | Remove last insert item. |
| popitem() | Removes a specified key/value pair. |
| setdefault() | Returns the value of the specified key. If the key does not exist: insert the key, with the specified value |
| update() | Updates with specified key-value pair e.g. car.update({“color”: “Blue”}) |
| values() | Returns a list of values in the dictionary. |

Source: <https://www.w3schools.com/python/python_ref_dictionary.asp>

Here is a sample program which stores a series of recorded outside temperatures. These temperatures are recorded in a Dictionary every 30 minutes.

The user may extract a reading by inputting the time, which is rounded to the nearest 30 minutes.

Graphical user interface, text, application, email

Description automatically generated

This program could be expanded to allow the user to add further readings:

Text

Description automatically generated

Here you will see the code TemReadings.update({readingTime:readintTemp}) which updates the dictionary with the key pair time and temperature.

### Tasks

5.20 Change the die roll TALLY program you wrote for TASK 5.6 so that rather than storing the tally of each die roll in a list, store in a DICTIONARY with each die number acting as the key.

5.21 Write a program which stores a student mark for a test in a Dictionary. Create a dictionary with 5 student names as the key and the corresponding mark. The program will ask for a student name, and it will promptly output their mark.

### Dictionary Lists

Dictionaries also allow us not only to store a key pair, but also a LIST within the key pair. Let’s say we need to store a series of values associated with a key. Take the following example:

*A charity store is collecting money locally and has distributed several buckets to volunteers around the town. The buckets are collected at the end of every week. The money collected in each bucket is totalled up and recorded against each week. The buckets are redistributed again for the following week. Some volunteers either haven’t collected any money or were not available so no money was recorded for that bucket, hence different number of buckets each week. Some weeks may have no takings, so not recorded.*

We need to store the totals for each bucket and for each week. Assuming we don’t need to label each or identify buckets we can just record the totals for each week using the following structure:

Text

Description automatically generated

Here you have a list associated to a key, with the key being the week no. When the user enters a week no, this value is used to find the key value pair, and then using the SUM function to add up all the values in the associated list.

We can add further values to the week, by simply adding to the list in the associated key.

Graphical user interface, text, application

Description automatically generated

Would output:

Text

Description automatically generated

To output and format the values you would iterate through the dictionary list as follow:

Text

Description automatically generated

Would output:

Text

Description automatically generated

For further processing you could add totals and summaries for each week. Using the SUM function to add up the values in the list.



Note formatting added for the output:



Teacher Mark book Project

5.22 Expand the program you wrote for task 5.21, allowing the dictionary to store additional test marks for a given student e.g., changing the program to have a List of marks matched to the key in the dictionary. Test the program so that given a student’s name it will display all their test marks.

5.23 Write a program which expands on 5.22 to stores a list of student’s names for a given group set. Set up two dictionaries, one for SET details and another so that you can record students in sets:

**Student in Sets**

**Set Name** **Students**

M1 Harry Bradshaw  
 Shelly Thompson  
 Isma Dupont

M2 Ashraf Odom

Lexi Seymour

Murphy Lynch

Jameson Shannon

**Sets**

M1 Mathematics Group Top set

M2 Mathematics middle set

M3 Mathematics set 3

Add to the program, so the user can enter a Set Name and it will display the list of students and a description of the set.

5.24 Extend the program in 5.23 so that the user may add students to a given set. The user will be continually prompted for additional students until they enter nothing. They select set number before adding students.

5.25 Extend the program in 5.24 so the user has a menu system:

* 1. View students in a selected set
  2. Add students to a set
  3. Remove a student from a set
  4. View Student Marks
  5. Exit program

When an option is selected the program will perform what you have already written for tasks 5.20, 5.21 and 5.22, with the addition of removing students.

5.26 **Extension to 5.25 adding functions**

To help structure the program try to incorporate your code into Functions for each menu option, including the menu itself e.g.

* Get\_set() // returns set\_no
* view\_student\_set(set\_no)
* add\_student(set\_no)
* remove\_student ( setno)
* view\_student\_results()
* display\_menu()

When the user selects and option the appropriate function is called along with the parameters. In this case you don’t need to return any values.

5.27 **Extension - Validation / Error checking / Formatting**

Extend program in 5.24 So that validation and error checking is included. Where possible add validation and appropriate feedback e.g. picking a valid menu option, ensuring when viewing or adding students the set selected exists, when adding as student that a name is entered, the student entered must exist before removing.

Format outputted data so it looks tabular and well laid out.

### Dictionaries within Dictionaries – Extension work

As well as creating dictionaries with lists, it is also possible to have dictionaries within dictionaries!

A picture containing map

Description automatically generated

Figure 1 Dictionary within Dictionary

This will output:

A picture containing chart

Description automatically generated

Creating more categories within the family, including a slightly different structure, allowing to search on name

A picture containing graphical user interface

Description automatically generated

Figure 2 Multiple dictionaries within Dictionary

### Tasks

5.28 Create a dictionary using the Figure 1 structure on your family, adding children, and guardians. Add the facility to extract data by typing the relevant key e.g., father, child1, child2.

5.29 Change the structed to Figure 2. In the code allow the prompt the user to add another child, inputting the child’s name and DOB before adding to the dictionary.

5.30 Add to task 5.25 that given a child number within the family will return their DOB.

# Lesson 6 - Structured programming

|  |  |
| --- | --- |
| **Lesson Outcomes**  In this lesson you will learn:   * how to structure a program; * how to create user-defined functions; * the use of parameters in functions; * the difference between global and local variables. * Ho to return values from functions. | **C:\Users\Graham\AppData\Local\Microsoft\Windows\Temporary Internet Files\Content.IE5\X6CHINOH\MC900441498[1].png** |

## 

## Introduction to Structured Programming

When creating a program, it is important that some sort of structure in the design and approach is considered. For a program to be maintainable it must be easy to follow and debug; and this is where the layout and structure is the key.

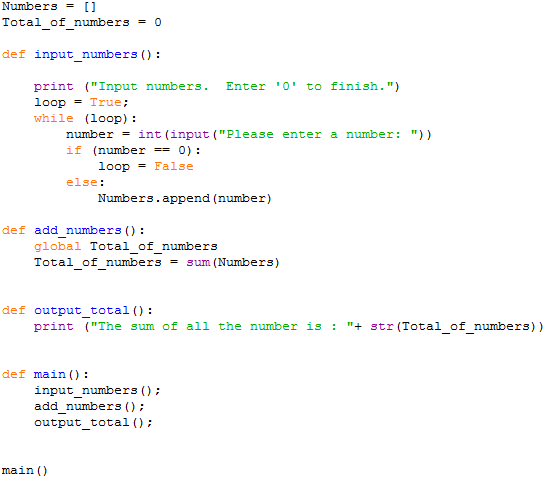
To help us design a structured a program we can use charts as visual aid in identifying the key processes and sequence of events. For example, let’s look at a simple program which prompts the user to enter a series of numbers, adds them together, and outputs the result:

Figure 1 Structured Chart

The chart, starting from left to right, splits the program up into three main parts. We could break down the chart further by adding sub functions to “input numbers” e.g., input a number, store in a list. Obviously, for a more complex program, you would have many sub-components potentially with 4 or 5 levels.

Once the structure has been designed the next stage is to implement it in Python.

## Defining Functions

To implement structure into our program we have to use functions. Functions are sub-routines or blocks of code which perform a specific operation. If we take the structured chart in **Figure 1**, there are three separate functions which we need to include in our Python program:

Main Function which calls all the other functions in order.

When the program starts this is the first command to be run. This calls the “main” function.

Simply applies the “sum” function to the list

Initialise global variables

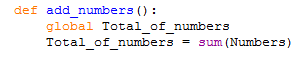
Asks the user for a series of numbers and stores in a list.

Figure 2 Add Numbers Program

Although not the same as our structured chart, the three main functions are there – input\_ numbers, add\_numbers and output\_total. Each of these functions is called from the main() function which represents the first box in the **Figure 1 Structured Chart.**

Each function is **self-contained** e.g., performs a single operation. You will notice the layout for a defined function is as follows:

Function name. Brackets contents are left empty as no parameters needed.



Global variable used. This will be discussed later.

Figure 3 function add numbers

## Global and Local Variables

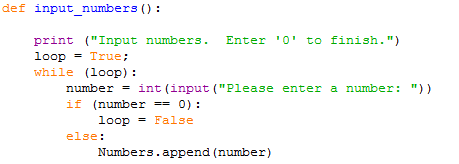
When creating functions, it is very important you understand the concept of **GLOBAL** and **LOCAL** variables. Without knowing the difference, you could find yourself defining variables which are inaccessible in other functions which may return some very strange results.

A **Global variable** is a variable which is accessible and defined for use across **the entire program**. If we look at the above program (**Figure 3**) you will notice two variables are defined at the beginning:

Numbers starts off as an empty List which is used throughout the different functions. We define it **OUTSIDE** a function at the beginning of the program to make it **GLOBAL**. The next variable keeps a TOTAL value, which again is accessible by all the functions in the program i.e., a **GLOBAL variable**.

When accessing a global variable within a function you must use the command **global:**

In this function the command **global** is used to reference the Total\_of\_numbers variable outside the function. If this command wasn’t specified then **a new LOCAL variable** Total\_of\_numbers would be initialised i.e., two different variables with the same name, one local the other global. You can see how this may get confusing.

**Local variables** are variables defined within the function – the contents can only be accessed and used within that function. We call this access the **SCOPE** of the variable i.e., only accessible within the function. Whereas the **SCOPE** for a global variable is the whole program. For example:

LOCAL VARIABLE called “loop” defined within the function. Set as a BOOLEAN value e.g. True or False

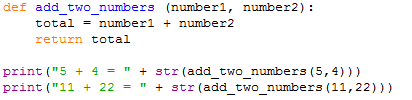
GLOBAL variable as defined outside the function.

Figure 4 input numbers function

## Parameters

As you can see when mixing global and local variables across functions confusion can arise. In the **Figure 2 Add Numbers Program** example you have two variables called **Total\_of\_numbers** with one as a global and the other local. Having to refer to one as global and the other local could easily get mixed up. If this was repeated in a large and complicated program, then the potential for bugs is a likely possibility. Without the use of **parameters** this is the only way the program can function. In programming this is considered bad practice.

When creating a good, structured program each function should be self-contained, using only local variables. To pass local variable contents from one function to another we need **parameters**.

Parameters are a great way of passing and return values from functions. It allows us much more flexibility in the **re-usability of code**. For example, I could write a simple function to add two numbers together:

Function parameters

Figure 5 add\_two\_numbers function

In **Figure 5** we have the function add\_two\_numbers(number1, number2). The required parameters are specified in the brackets as number1 and number2. The function is called with the values 5 and 4, with 5 assigned to number1, and the value 4 assigned to number2.

To return a value or series of values from a function we use the **return** command followed by the variables we want to return e.g., in the **Figure 5 add\_two\_numbers function** we return the contents of the ***total*** variable.

The returned value is then used in the expression where the function was originally called, in this case the two print commands at the end of the program. You will notice the **str** function is used to convert the returned value from an **integer** to a **string (casting).**

## Returning multiple values

In Python sometimes it is necessary to return multiple values from a function. For example, you may have the following statistics function:

Graphical user interface, text, application

Description automatically generated

Return values are assigned to the variable(s).

Here you are passing as a parameter a LIST of values. From this LIST you are calculating the average, maximum and minimum values which you want you want to return. In this program all three values are returned and can either be assigned to individual variables or a Tuple (essentially a LIST which you cannot update).



Printing the contents of those variables would output the following:



A picture containing text

Description automatically generatedReturning multiple values is also very useful when getting multiple inputs from a user in a separate function:

Figure 6 returning values from a function

Two variables returned

Two Variables assigned to returned values

### Tasks

6.1 Write a function that will take three numbers as input parameters and return the largest number. Incorporate this function into a simple program which ask the user for three numbers and displays the largest.

6.2 Write a function which given any LIST of numbers will return the average number.

6.3 Using structured programming techniques (think about structure first) construct a program which prompts the user for a series of numbers and then outputs the average of those numbers. You can use the function created in task 6.2 as part of the program.

6.4 Using functions, write a program that asks the user to enter an odd number, validates the number and then prints an inverted pyramid of stars based on that number. For example, entering the value 5 will produce:

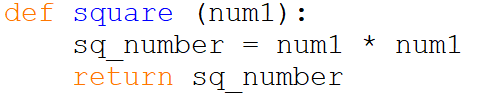
\*\*\*\*\*  
 \*\*\*  
 \*

6.5 Write a **structured program**, using functions, to let the computer guess a number the user has thought of, with a range specified by the user. After each guess the user will tell the computer if it higher or lower and continue until the computer guesses the correct number. The program should also output the number of attempts. Ignore validation for incorrect higher and lower user entry.

Pass by Value, Pass by Reference

When passing parameters from one function to another you will often hear the phrase “Pass By Reference”, and “Pass by Value”. These are terms which have existed in programming for many years and refer to the way values are transferred from one function or subroutine to another. As technology and practices have changed over the years, so too are the ways in which these methods are employed in various languages.

Pass by Value simply refers to the value being passed from one function to another. For example, let’s take a square function which has as its input a number and returns the square of that number:

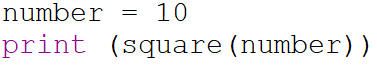


You can call it anywhere in the program. One typical way might be using the print function:



This print calls the square function with the value 7, outputting the result 49.

Here, you are transferring a parameter VALUE ‘7’ to the square function, which in turn will return the contents of a **local variable** sq\_number (assigned inside the square function), which contains the value 49 (7\*7). Only the value is being transferred to the function. In fact, a copy of the value is made in memory with a new variable identifier (name) as specified in the function parameter (in this example num1). Hence the term **“Pass by Value”.** Equally, you could also pass the contents of a variable, which would again be copied and assigned a new variable name (num1) in the function. Example:



#### Pass by Reference

This traditionally means passing a value using a reference point. When you create a variable in any programming language, a location in memory is reserved and assigned an **IDENTIFIER,** this being the name. So, for example:

value = 10

Here a space is reserved in memory, with the variable identifier “value” assigned, storing the assigned integer number 10 as illustrated below:

|  |  |  |
| --- | --- | --- |
| Identifier | Memory Location | Data |
| **value** | **1020** | **10** |

When passing values by reference you simply pass the **memory reference** as opposed to the value, so for example in the following pseudo code you would write:

Subroutine square (By Ref number)

number = number \* number

value = 12

print square(value)

You will notice that the subroutine square doesn’t return a value. Rather than passing the value to the subroutine the memory location is passed and given a new local variable identifier, therefore any changes to this variable will change contents of where it is pointing to.

Looking at the above program in detail:

The variable value is assigned the value 12:

|  |  |  |
| --- | --- | --- |
| Identifier | Memory Location | Data |
| **value** | **1020** | **12** |

A space is reserved at memory location 1020, assigned the identifier “value”, with the integer 12 as contents. This variable is then passed by **REFERENCE** (memory location 1020) to the subroutine square:

Memory location 1020 passed to subroutine square and given the identifier **number.**

print square(**value**)

|  |  |
| --- | --- |
| Memory Location | Data |
| **1020** | **12** |

New identifier “**number**” assigned to memory location 1020.

Subroutine square (ByRef number)

number = number \* number

Contents of memory location changed.

|  |  |
| --- | --- |
| Memory Location | Data |
| **1020** | **144** |

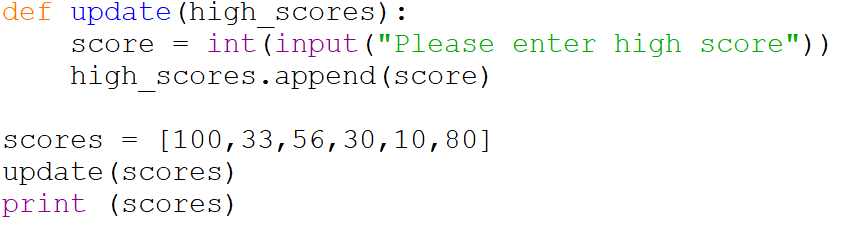
The following command:

Call output square(value)

Outputs contents of variable value which currently is assigned to memory location **1020** therefore outputting 144.

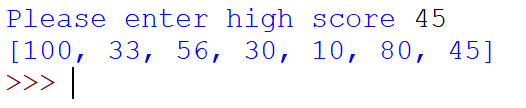
On a technical level Python neither uses pass by reference or value but pass by **Object Reference**. With object reference changing for immutable objects (Boolean, int, float, tuple, str), this is effectively **pass by value** and mutable objects (list, set, dictionaries), **pass by reference**. Technicalities aside, you can achieve pass by reference in Python using **Lists** and **Dictionaries**.

In Python to achieve **pass by reference**, you define the LIST as normal outside the function, and then pass as a parameter. The LIST can be updated without having to return. For example:

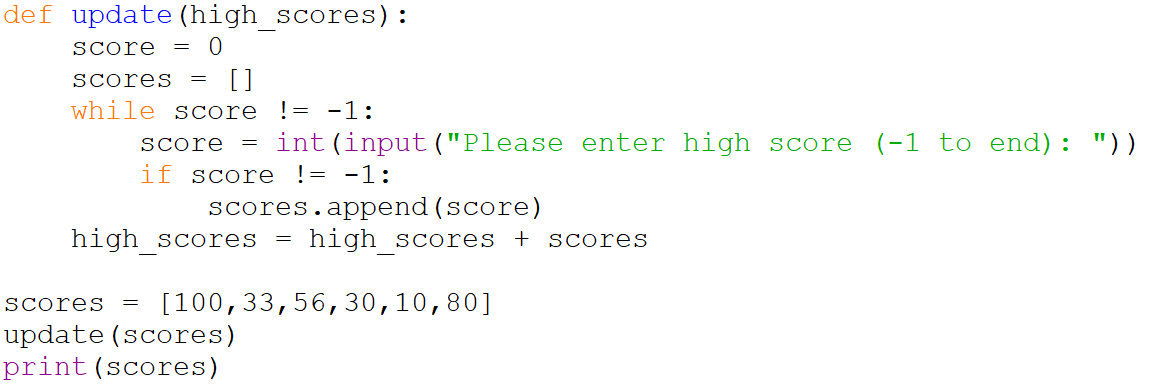


In this example you are passing the scores LIST to the function update. The code inside the function asks the user for a new high score which is appended to the high\_scores parameter. There is no need to return a value.

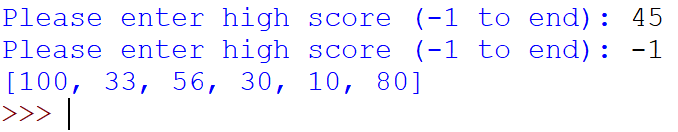
Running the program:



**Notice** if you were to redefine high\_scores, then the LIST wouldn’t update as you have created a new LOCAL List.



The above wouldn’t work, as high\_scores redefined locally in the function.



As you can see, the LIST hasn’t been updated with a new score.

Looking back at the **Figure 2** program we can amend to include **NO global variables.**

Text

Description automatically generated

Figure 7 program with parameters example

You will notice that each function has a parameter, with one returning a value.

### Tasks

6.5 The following function has been created to update a student class list:

function enter\_classlist()

classlist = []

complete = False

while NOT complete:

student\_name = input (“Please enter Student Name (enter X to exit)”)

if student\_name = “X” then

complete = True

else

Update\_classlist (classlist, student\_name)

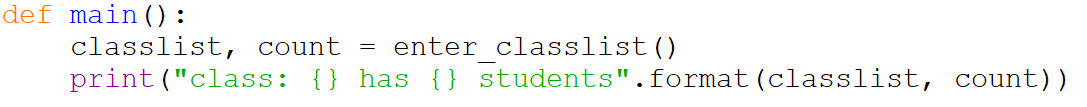
end if

end while

print (classlist)

Create the above program, adding a new function Update\_classlist (student\_name) which will update the classlist with the inputted name. For it to run, you will need to call the function enter\_classlist() at the bottom of your program.

6.6 Add to the program in 6.5, adding the following function:



Change the enter\_classlist function accordingly to calculate and return the correct values. At the bottom of the code add a call to the function main() as opposed to enter\_checklist().

6.7 Write a function called sort(numbers), which given a LIST of numbers will return the LIST ordered, and with the number of items in the list along with the average number. For example, after the function call:

A picture containing text

Description automatically generated

Will output:

A picture containing text

Description automatically generated

# File Lesson 7 - Filing

|  |  |
| --- | --- |
| **Lesson Outcomes**  In this lesson you will learn:   * the difference between text and byte files. * how to create, read and write files. * to write and read date into variables and Lists * how to use direct file access using the SEEK command. | **C:\Users\Graham\AppData\Local\Microsoft\Windows\Temporary Internet Files\Content.IE5\X6CHINOH\MC900441498[1].png** |

## Introduction to Files

Files are very important and allow us to store data from a program to backing storage e.g., a hard drive, USB or disk. In most programming languages files come as two main types: **binary** and **text.** Binary files are used to store **raw data** such as images, sounds or other non-text information (<https://youtu.be/ylUYkRXPAqY>).

Text files are different, as the data is stored as **CHARACTERS** with each block of data either stored as a complete line of text or individual characters on the page. The **TEXT** file is made up of **ASCII** characters and saved with the extension “.txt”. You may also include basic text formatting such as bold, italic, carriage return, line spaces and tabs using **special control characters**.

In Python you can create, edit, and read a file using the file object as follows:

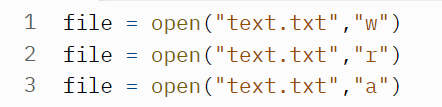
**file object = open(file\_name [, access\_mode][, buffering])**

**filename**: the name of the file you are going to create

**access mode:** determines what mode the file is going to be open in e.g. reading, writing or append.

**buffering:** allows the computer to read data into a buffer to speed up access. In this lesson we will be ignoring this option.

The access mode option when opening the file can be set to any of the following:



|  |  |
| --- | --- |
| **Modes** | **Description** |
| r | Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode. |
| rb | Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode. |
| r+ | Opens a file for both reading and writing. The file pointer will be at the beginning of the file. |
| rb+ | Opens a file for both reading and writing in binary format. The file pointer will be at the beginning of the file. |
| w | Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing. |
| wb | Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing. |
| w+ | Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing. |
| wb+ | Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing. |
| a | Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing. |
| ab | Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing. |
| a+ | Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing. |
| ab+ | Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing. |

## Writing File

#### Writing string text

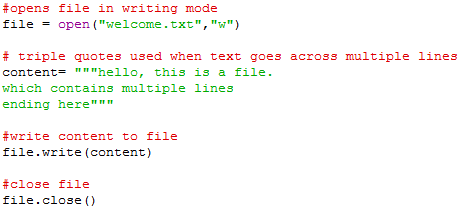
To create a basic file, you need to open the file, write the text content, and then close the file:

Figure 1 Writing to a file

The above program will create a file called “welcome.txt” in write mode. The file should appear in your default directory (where you saved the program), however you can specify the directory e.g., “F:\my\_python\_work\welcome.txt”.

The content variable contains the text to be saved to the file. You will notice **treble quotes** have been used to tell Python that this string is across multiple lines.

The file.write(content) command takes the contents of variable content and writes to the opened file.

The last line closes the file. It is **important you close the file** otherwise the data won’t be written. If you open the file in note or word pad you should see the following:

Graphical user interface, text, application

Description automatically generated

Figure 2 Notepad file

#### Writing a List

Alternatively, rather than writing entire pieces of text you might want to store a list of individual values or sets of values. The way in which you write to a file is dictated by how you want to structure your data, so that when read back it and store and processed in memory appropriately.

For example, me might want to save the contents of a LIST. To do this we would need to iterate through the list and store the value on **each line** on the TEXT file:

Text

Description automatically generated with medium confidence

Figure 3 Saving a list

We create the list first, open the file and then loop through each item in the list, assigning the value to the variable name and then writing the contents of this variable to the file. You will notice a **“\n”** attached to the end of the string. The **“\n”** is a **control character**, which when displayed tells the text editor or print output to display a **NEW LINE**. In this example, this prevents all your names appearing on one line:

Graphical user interface, application, Word

Description automatically generated

Figure 4 Text File list

You may also use the writelines method which given a list will automatically iterate through and write to a file:



Figure 5 writelines

To write each item to a new line you will note that each item in the list (line 1) has “\n” added.

## Reading a file

Having created a text file, reading it is simply the reverse. An example of how to read from the above file:

Text

Description automatically generated with medium confidence

Opened in READ mode.

Figure 6 reading a file

This program opens the file we created earlier in **read mode.** The file.read() function reads the entire file and returns as a string which is then printed out to the screen.

Alternatively, rather than reading the whole file you might want to just read the first line:



Figure 7 Read line of a text file

Note, when reading a file, you start from the beginning and work your way through (**serial access**). Using the above program (Figure 7), if I inputted the command file.readline() again it would give me the next line (line 2). You must also remember that the readline() method also returns the carriage return (last character, ‘\n’) within the string, which you won’t see unless assigned to a list or variable.

#### Iterating through text file

It is possible to create a simple for..loop to iterate through each line in the text file:

**Welcome.txt** file

Text

Description automatically generated

**Code:**

Graphical user interface, text, application

Description automatically generated

#### Reading a text file into a LIST

Suppose we wanted to extract a list of names in a TEXT file into a Python List, ready to process in our program. This can simply be done by reading each line of the text file and storing as an element in the list.

The following code will read the text file line by line and insert into an empty List:

Create Empty List called names.

Graphical user interface, text, application

Description automatically generated

Take each text line from file and assign to name. Append name to List names.

Figure 8 reading text into a list

You will notice the output:



Notice, as outlined by the red boxes, that character carriage return ‘\n’ is stored in the List, which you don’t really need. To avoid this, you need to use the **strip(‘\n’)** method to remove string ‘\n’ before storing in the list. Changes to the code at line 7, are needed:

Text

Description automatically generated

Figure 9 Append stripped string to list

The strip() method will remove whatever is specified in the parameter from the string in this case the control character **‘\n’**:



### Tasks

7.1 Write a program which asks the user for 5 names; each time the name will be saved to a text file. The program should output the contents of the file after the names have been entered.

7.2 Create a file (in notepad) with a series of usernames and passwords. Format the text so that there is a line for the username and a line for the password. Repeat this process for each user. For example, the password text file could contain:

Graphical user interface, text, application

Description automatically generated

Write a **structured program**, using functions, which will prompt the user for their username and password and display appropriate error messages if incorrect username and password provided. The user is allowed three incorrect attempts before the program quits.

The program should read the above file to check:

1. The username exists
2. The password entered equals the password stored for the username given

## Comma Delimited Files

#### Creating a file

One very common method of storing and retrieving structured data from a text file is using comma delimited files. A comma delimited file is simply a way of storing data such that each **field** or **item of data** is separated by a **“,”** on the same line, thus making up the **record**. For example, say we wanted to store a list of student names and their marks in a recent test.

|  |  |  |
| --- | --- | --- |
| **Surname** | **Forename** | **Mark** |
| Smith | Jane | 56 |
| Claes | Yakub | 89 |
| Imran | Kyle | 34 |
| Miles | Kate | 76 |

We can take each row of data (record) and separate each column (field) using the “,”, storing in turn line by line in the text file. For efficiency and ease of processing, before writing to the text file, we should store the data in a LIST.

Here the list comprises four sub-lists, effectively four records from the table.



To save, we need to iterate through each sub-list (record) and then save to the text file (the ‘\’ is used to fit the list across two lines but treated as same line).

Text, letter

Description automatically generated

Figure 10 sub-lists savings

**Lines 7 to 8** write each FIELD separated by a comma to the text line. The code on **LINE 10** write the last field with additional “\n” to move on to the next line. Notice, the str() function is used to convert the integer student\_mark[2] to TEXT.

The following text file has now been created:

Text

Description automatically generated

#### Strip, Join and Split

For the purposes of text files, one of the most useful STRING methods is **Strip, Split and Join**. As has been discussed the Strip method removes leading and tailing specified characters from a STRING. The Split method will return a list from the string, separating each element by a specified character:



Here we create a list called nameslist with each element comprising the names from the string and the comma in brackets indicating each split:



Now we have a list, we can process as normal and then later save back to a text file, attaching the “,” and control character “\n”.

The **Strip** method is used to remove unnecessary leading and tailing characters from a string.



Using the default (no parameters), this will remove any leading and tailing spaces:



Removing the “\n” control character after reading from a text file:



We can also use the JOIN command to create a STRING based on a LIST. This is particularly useful when writing data back to a text file.

If we take the program created earlier where we have the following list:



Instead of adding the “,” manually we can use the **JOIN** command. The **JOIN** inbuilt string function is used to join element of the sequence separated by a string separator.

For example:

Text

Description automatically generated

**LINE 7:** this is key line of code. You will notice a **for loop** is included in the join parameter, which iterates through each element in the **LIST.** Each element is converted into a STRING (you will notice the 3rd value is an INTEGER, and the Join command will only work with List elements that are Strings). The **LIST** String elements are then joined together use the join command.

### Tasks

7.3 Create a STRING containing the months of the year:

months = “Jan,Feb,Mar, …”

Use the split command to convert the string into a List called months. Output the list, using a loop, so than months are displayed one line at a time.

7.4 Ask the user to enter a sentence followed by which characters to be removed. The program should output the result. For example, entering “I am Graham” and then “I am”, will output: “Grah”

#### Reading a file

This is much the same as writing, with further processing needed to separate items from a line of text. Each line from the text file is read, splitting where the “,” is found into separate elements ready to insert in a list. Again, a loop is used to repeat this process for each line in the text file:

Text, letter

Description automatically generated

Figure 11 reading text file into list

Evaluating line 5:



Here, the line of text is stripped of the control character “\n” and then using the **split()** method to create a sub-list of elements where each element will be separated using the “,” from the text file:



This list is then appended to Marks lists (list within list).

This process is repeated until the file reaches the end.

The final Marks list will contain the following:



### Tasks

7.5 Change the program in 7.2 so that the file containing username and password is stored on one line, for example:

Text

Description automatically generated

7.6 Create program which asks the user for a year, month, and rainfall e.g., “Jan”, 2020, 5.3. Store these results in a text file, using an appropriate structure. After each entry ask the user if they want to continue, if so repeat process.

7.7 Extend the program in 7.5 so the user is given menu options, with a request for either option 1 or 2:

1) Enter username and password  
 2) Change username and password

Please enter option 1 or 2.

Make sure your program is structured, so that you have a function for option 1 and option2. These functions are called when the user has selected the option. For option 2, the program will need their old username and password first before changing – try to reuse functions.

**Hint:** The common method is to load all username and passwords, change and then save back to the text file, essentially re-writing the file.

### Extension Task

7.8 Extend the program in 7.7 so that the password is encrypted. This should be a encrypt\_password (password, shift) function that given password text and shift will return an encrypted string. Use a simple Caesar cipher method to encrypt the password (e.g., shift each character by 1 so that A becomes B and B becomes C etc. Z should become A.). The encrypted password is used in saving to the file. To compare entered password encrypt the password entry before comparing with password stored on the file.

Question: Why would you want to encrypt passwords on a file?

## Other filing methods

Sometimes when reading and writing to a file you will need to simplify the process and provide exception handling. Exception handling is a way of trapping and preventing your program from crashing when an error occurs e.g., trying to read a text file that doesn’t exist.

Simplifying and providing exception handlings in text file processing can be achieved using the WITH command. Here is a program which reads the names text file we created earlier:

Text, letter

Description automatically generated

Figure 12 Filing using with

Notice, you don’t have to close the file!

#### Error Trapping

In combination with the WITH command you can also error trap, providing an error message in the even there is a file error.

Text, application

Description automatically generated

Figure 13 Text File Error Trapping

The try command will run the code beneath it and at any point if there is an error the coded below except is executed and the program continues to run.

To summarise here are the complete list of methods available when reading and writing to files. So, using the example in **Figure 11 Text File Error Trapping** you can use the following methods with my\_file:

**Method Description**

readline() Returns one line from the file e.g. my\_file.readline()

readlines() Returns a list of lines from the file

seek() Change the file position

seekable() Returns whether the file allows us to change the file position

tell() Returns the current file position

truncate() Resizes the file to a specified size

writable() Returns whether the file can be written to or not

write() Writes the specified string to the file

writelines() Writes a list of strings to the fi

### Tasks

7.9 Enter the program in **Figure 11** Text File Error Trapping, run to check if error trapping works. The file doesn’t exist meaning an error will occur.

## Direct Access Filing

So far in these exercises you have dealt with serial access text files, that is, to read a file you start at the beginning and work your way through. In filing it is often more efficient to go directly to the data you need rather than going through each line of text in turn.

Suppose we had the following text file which stores player’s scores in a game:

Graphical user interface, text, application

Description automatically generated

Figure 14 Scores Text File

If we wanted to find player number 3 score, we would have to go through each line to compare the player number read with the number we wanted. Once we have that position then we can simply go to the next line for the Name, and the next line for Score.

The above method is reasonable when dealing with a few entries, but if we had hundreds possibly millions then this would be very inefficient and potentially very slow to process.

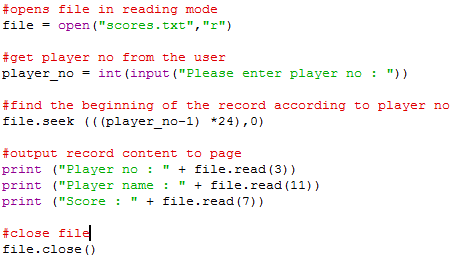
Python allows direct-access filing using the seek command. The seek command allows us to specify a position in the file to read from. However, there are drawbacks, as you must specify the exact character position. To calculate the text position of nth player in the file we would need to fix the character length for each player entry. In the above example the structure is as follows:

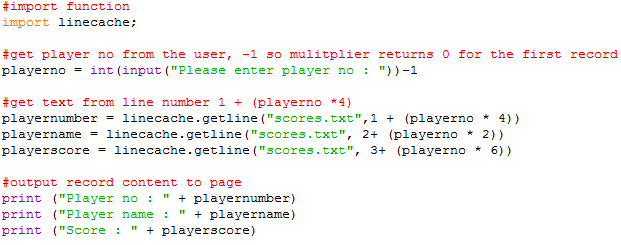
|  |  |
| --- | --- |
| Player No | 3 characters + 1 (carriage return ‘\n’) |
| Name | 11 Characters +1 |
| Score | 7 characters + 1 |

figure 15 Field Structure for text file

**Note:** Carriage return is a control character which tells the text editor (notepad) to take the text to the next line.

Using the table structure in **Figure 15** we calculate that each record is 24 characters long. Knowing the length of the record we can calculate the position of any record in the file using the basic formula player no \*24.

For example, if we want to read player number 4, we could us a basic formula to find the correct start position in the file e.g. (4-1) \* 24 for the character no. In Python:

Although this works, it is a little untidy and complicated. There are other methods which we can use to make the process easier. One method is to use the linecache import. Using this import module, we can specify the line number we want to read. We can adapt the other program to refer to lines as opposed to character positions:

Scores.txt text file

You will notice, using the linecache import, we must load the text file with the line we want each time, thus removing the whole open and close text file approach.

### Tasks

7.10 Write a program which takes a series of marks for a set of students and stores them in a file. You should store the file in the following format:

|  |  |
| --- | --- |
| **Record No** | 2 characters |
| **Firstname** | 10 characters |
| **Surname** | 10 characters |
| **Mark (%)** | 3 characters |

Use the .format command to force how many characters per field before writing to the text file.   
Test the program with 4 records.

7.11 Add to the program so the user can specify a record number to view on the screen. When the record number entered the program should, for example, display the formatted text:

|  |  |
| --- | --- |
| Record No | 00 |
| Firstname | Sarah |
| Surname | Potter |
| Mark (%) | 75 |

7.12 **Extension**

Change the program from exercise 7.6 so the user can specify a record to delete.

**HINT**: load all the data in, delete the record, and save it back.



The above reads the text into the variable lines. Lines is written back starting from line 3 to -1 (from the end of the file)

**Question**

For very large files why might this method be inefficient? Could you suggest another way? Research **Hash tables and files.**

# ­

# Lesson 8 – Object Oriented Programing

|  |  |
| --- | --- |
| **Lesson Outcomes**  In this lesson, you will learn:   * The basics concepts of object-oriented programming * how to create classes and objects * how to create methods * Inheritance, Association, Aggregation and Composition * the difference between private, protected, and public variables. | **C:\Users\Graham\AppData\Local\Microsoft\Windows\Temporary Internet Files\Content.IE5\X6CHINOH\MC900441498[1].png** |

## Introduction to object oriented

AS you learnt in Lesson 6 about structured programming, object oriented is a way of partitioning your program into logical chunks. Whilst functions and subroutines can be used to split up your program they can’t be duplicated easily or changed without re-writing or importing in another program. This is where object oriented comes in and is in fact the most widely used programming paradigm in today’s software development.

Object oriented programming has many concepts which go beyond this exercise; however, we are going to concentrate on two main parts: **Classes** and **Objects.**

In object-oriented programming, a class is a template definition of piece of code which when executed can be used to create many **OBJECTS**. An object is a specific instance of a class; it contains real values instead of variables. The class is one of the defining ideas of object-oriented programming.

The class itself contains all the attributes (variables) and methods (functions) which are related to the overall class essentially **ENCAPSULATED** within it.

### Classes and Objects

For example, imagine the commonly used class “WINDOW” which you often see used in programs.

Graphical user interface, application

Description automatically generated

Figure 1 Two windows opened from a program

From **Figure 1** you can see an average program which utilises two WINDOWS. Each of these WINDOWS share the same properties e.g., they can be expanded, closed, minimised, moved and they have a border, 3 icons in the corner and are normally in the same colours as your desktop. So, in effect, you can say a WINDOW is common across all the different programs.

In object- oriented programming a “WINDOW” is example of a CLASS. Each WINDOW has the same basic functionality but may vary slightly, so in effect the WINDOW has created many OBJECTS based on itself. Each OBJECT is based on the CLASS “WINDOW” but has its own unique properties and may occur many times (try opening the same window program lots of times on your desktop). The Class Window can be represented using a basic UML (Unified Modelling Language) diagram:

Text

Description automatically generated with medium confidence

These are the ATTRIBUTES (Essentially internal variables)

METHODS (Functions contained within Class)

Figure 2 UML class "Window"

Here we have the basic elements of the WINDOWS class. Standard internal **ATTRIBUTES** (Variables) would be width, length, and colour. In terms of **METHODS** (Functions) there would be focus(), close(), open(), resize(), minimise(), and maximise(). Notice we tend to associate METHODS with VERBS e.g., actions to do something. These would essentially be the standard TEMPLATE functions used within the CLASS.

In a real system the above Class Window will be used many times for different running applications:

Diagram, schematic

Description automatically generated

Each OBJECT using all the properties and functions of the Class Windows. Each Object we say is an **INSTANCE** of the Class. In this example we have four WINDOWS containing different applications. Once create these objects have been **INSTANTIATED** (created) will function independently of each other but using the same TEMPLATE code from the Class.

### EMPLOYEE EXAMPLE

In Python, we can illustrate this idea very simply. Imagine we are creating a program to store details of employees in a company.

Table

Description automatically generated

We can create a class called Employee, ensuring we capitalise the name:

Text

Description automatically generated

These are **METHODS,** which are effectively functions.

**Attributes** (variables) defined in the class.

Figure 3 Employee Program

This is effectively the class template. In this we have a CLASS variable called empCount which is initialised to zero. This variable is bound to the class rather than to the instance of the class (see later). Within the class we have defined three methods \_\_init\_\_, displayCount and displayEmployee.

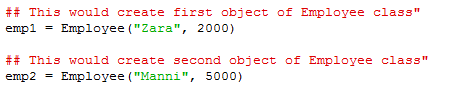
The first method *\_\_init\_\_()* is a special method, which is called the class **constructor** or initialisation method that Python calls when you create a new instance of this class.

You declare other class methods like normal functions with the exception that the first parameter each method is self. Normally the IDE will add the self argument to the list for you; you do not need to include it when you call the methods. Self is the address memory reference to the object being instantiated using the class (object reference). By convention we always have this as the first parameter.

### 

### Objects - Instance of a Class

To create instances of a class, you call the class using the class name and the optional parameters in the brackets. So, from the class defined in the Figure 3 Employee Program, we can create the following instances:

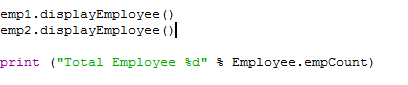


The above program effectively creates two objects emp1 and emp2. Each of these objects has all the methods and variables defined in the class Employee. When instantiated the \_\_init\_\_ method is the first function to be called with the parameters self, name and salary which will initialise the variables:

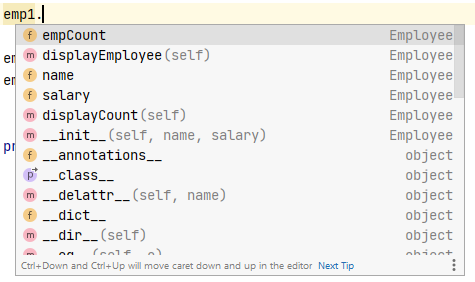
Text

Description automatically generated

Note self refers to the object be created rather than the class itself. You will also notice the “\_\_” either side of init, this is used to identify the method as unique and private to this class (covered later). As you initialise the object you are creating the object variables and assigning the values passed as parameters. The class variable empCount is incremented by one as each instance created. Having created these objects, you can then go on to access the individual attributes and methods.



When referring to attributes and methods of an object we use the “.” Identifier method. Depending on the program IDE you are using, when defining an object, a list of methods and attributes should appear when you type the full-stop:



Calling the method displayEmployee() would produce the following:



Notice that each object has its own variable values, the first emp1 as Zara and emp2 as Manni.

### Private and Public methods/attributes

You may notice from the above program that all methods and attributes within the Class are listed and available on the instantiated OBJECT. For example:

Text

Description automatically generated

Running this will output:

A picture containing text

Description automatically generated

Through the program you are able to call the method displayEmployee() which outputs the employee data in formatted way or you may print out the contents of variable directly without using the displayEmployee() method e.g., print (emp1.salary). Sometimes, when developing large programs and to avoid having two approaches of accessing the same data and to control how the attributes are updated and created, we defined methods/ attributes as **PRIVATE.** This also helps **ENCAPSULATE** all the methods and data within the CLASS, effectively hiding internal functions/data (**INFORMATION HIDING**).

**PRIVATE attributes** in Python are identified by inserting “\_\_” before the attribute name. For example, changing the attributes in the Employee class so they are no longer available outside of the class.

Text

Description automatically generated

If you execute the program again you will get the following error:

A picture containing text, person, screenshot

Description automatically generated

Graphical user interface, text, application

Description automatically generated

It is saying the Employee object has no attribute ‘\_\_name’ – this is now private!

The only way we can now display these values is through the displayEmployee() method.

**Chart

Description automatically generated with medium confidence**

These attributes are available in this function because the function is contained within the CLASS.

It is also possible to make METHODS private, so they can only be used within the class.

Let’s say we want to add another two methods in the Employee class:

Graphical user interface, text, application

Description automatically generated

We have defined \_valid method as **PRIVATE** which means it is accessible inside the CLASS but not outside. It is used to validate the increased salary to ensure it doesn’t go above ceiling, called from the public method increase\_salary. Notice from the increase\_salary method it returns whether valid or not.

Executed from main program:

Text

Description automatically generated

Outputs:

Text

Description automatically generated

### Built-In Class Attributes

Every Python class keeps following built-in attributes and they can be accessed using dot operator like any other attribute −

**\_\_dict\_\_:** Dictionary containing the class's namespace.

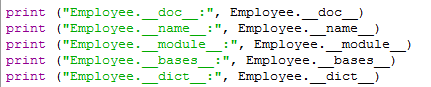
**\_\_doc\_\_:** Class documentation string or none, if undefined.

**\_\_name\_\_:** Class name.

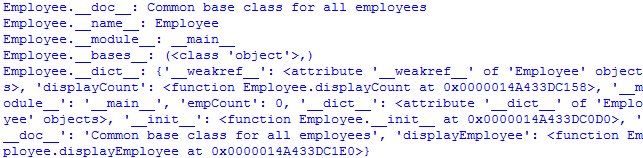
**\_\_module\_\_:** Module name in which the class is defined. This attribute is "\_\_main\_\_" in interactive mode.

**\_\_bases\_\_:** A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

For the above class let us try to access all these attributes −



The output will be:



### Tasks

8.1 Create the following class:

**Car**

|  |  |
| --- | --- |
| **Attributes** |  |
| Make | Car Make. |
| Model | Car Model. |
| Top\_speed | Stores the top speed of a car in mph |
| Type | Type car e.g., hatchback, estate, off-road |
| Colour | Colour of the car |
| Engine\_size | Stores engine size e.g., 1.8, 2.2 |

|  |  |
| --- | --- |
| **Methods** |  |
| \_\_init\_\_ | Define all the attributes above to given parameters |
| Display\_top\_speed | Outputs the top speed of the car |
| Set\_Colour | Sets the colour of the car |
| Display\_Engine\_Size | Prints out Engine size |

Write an program which is able to instantiate an Object from this class e.g.

Car1 = car(“Ford”, “Escort”, 80, “Estate”, ”Red”,1800)

8.2 Using the class defined in 8.1, create 4 more CAR objects, defining the values for each (you can make this up)

8.3 Using the OBJECTS in 8.2, call methods to change colour, display engine size, and display speed.

8.4 Set the engine\_size and top\_speed as **private** attributes.

Create a private method validate\_colour with the parameter colour which is called by public method set\_colour. There are only five colours available [‘Red’,”White”, “Black”, “Blue”, “Green”]. If invalid colour given, then an error should be displayed.

## Instance vs Class Variables

Looking back at the last program**,** you will notice that we had one CLASS variable and two OBJECT instance variables:

Text

Description automatically generated

Class variable

Instance Variable

The class variable is used to provide a fixed variable which is accessible by all objects (instances of the class). This is very useful if you want to provide value which needs to be accessed and used as a common variable across many objects. In the case of the program in above. we have an empCount variable to keep track of the number of employees created i.e., each object instance created. The instance object variables are created and referred to using the self command followed by “.”.

To illustrate how this works, here is a program which creates multiple objects from the same class using instance and class variables. In this example we are going to simulate a card game, starting with dealing ten cards to one player.

Graphical user interface, text, application, email

Description automatically generated

Object Attributes card\_description, card\_no

Instance Variables

Class variable. Creating a pack of cards , loop through suites and then numbers

The object card is created from deck class and appended to mydeck list.

Figure 4 Card dealing

The deck cards list is created as a class variable and each time an instance is created from the dealer deck class one card is removed from the deck class cards list and the next\_cardno is incremented by one and each assigned to the instance variable, card\_description and card\_no. The instance object card is appended each time to my\_deck list.

For output to the user, the program loops through the\_deck list printing out the attributes for each card object.

### 

### Tasks

8.5 Type the program in **Figure 4 Card dealing**, and amend so are two player decks are created from the dealer, each with five cards. Display each player deck.

8.6 Add another method to dealer\_deck called reset(). When called the cards list is reset to its original state. Note to refer to a CLASS variable within a method:

Classname.Variablename = value

Test this method in the main program by re-displaying two player decks.

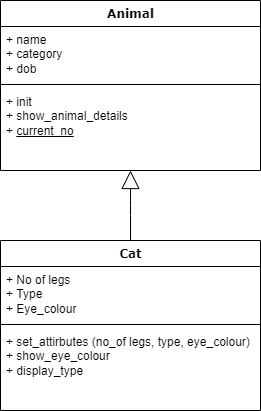
8.6 Change program in 8.5 so that a new class is created called GAME. Move your main code to this GAME class. On program run, the GAME class is instantiated, displaying the two player decks.

8.7 Extension – can you think of another 3 methods you could add to the GAME class? Which attributes could you make private?

Add the methods and incorporate into the game.

## Inheritance

One of the most powerful OO concepts is inheritance. As the name suggest this allows us to create classes which inherits both attributes and methods from another class. When inheriting from a class, two are created, one parent and one child as illustrated below:



Methods (functions)

Attributes (variables)

Figure 5 Inheritance

In this **UML** diagram (Unified Modelling Language) you can see the Animal class with the attributes listed at the top of the box and the methods below. This is the parent class. The CAT class is created which inherits, indicated by an up hollow arrow, **ALL** the methods and attributes from the parent with its own additional attributes and methods. In this case, the attributes are no\_of\_legs, type, eye\_colour and methods init, show\_eye\_colour and display\_type. Note, the init is the constructor method – called when object instantiated. In the Animal class you have a CLASS variable (in OO referred to as a **static variable**), this is indicated using the underline.

In Python, we implement this UML as follows:

Text

Description automatically generated

**Child Class**

Initialising: name, category

Methods: show\_animal\_details and current\_no

**Parent Class**

Initialising: name, category

Methods: show\_animal\_details and current\_no

**Class Name (Inherited class)**

Notice you now have access to both Cat and Animal methods:

A screenshot of a computer

Description automatically generated with low confidence

Output:

Text

Description automatically generated

### Tasks

8.8 Using the UML below create the appropriate structure in Python, ensuring correct attributes and methods added.

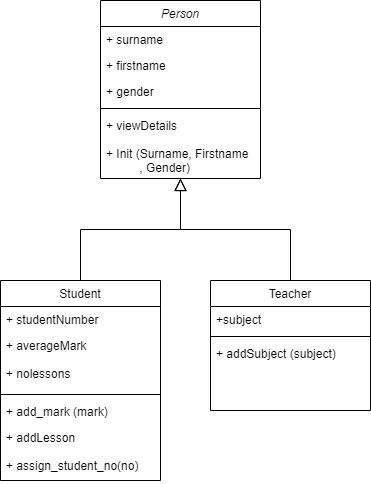


Figure 6 Person, Student, Teacher UML

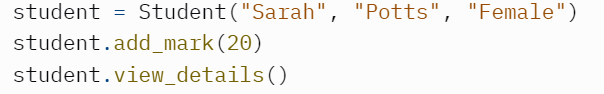
Having created the above classes, the following should allow the instantiation of the Teacher class:

teacher = Teacher("Smith",”Tomasz","Male")

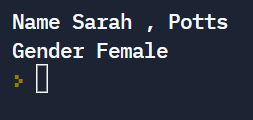
8.9 Extend task 8.8, so for the Teacher class, insert code for the method addSubject, allowing the user to add a subject to the teacher:

teacher.addSubject("Computer Science")

8.10 Extend 8.9, for the Student Class, add code to the method viewDetails and addMark. When the following code added:



The example output would be:



Add further code to view the mark associated with the added student.

## Inheritance – Super Method

When using inheritance one the problems associated with instantiating a child object is which constructor method to use. So far, in the previous examples we created a child object without a constructor method, therefore defaulting to the \_\_init\_\_ method in the parent class. However, suppose we wanted to create a different constructor in the child object? By default, Python will always employ the last child constructor method, but this isn’t always appropriate as you may want to use the parent or indeed both. This is where the super function comes into play.

Take the class structure in **Figure 6 Person, Student, Teacher UML.** Suppose we want to create a constructor when instantiating the student class but still utilising the Person parent:



Here we use the super() to refer to the parent class, effectively creating an object using the parent template, calling the \_\_init\_ method with the parameters surname, forename and gender. We then add an additional attribute mark for the student object, assigning to value of the parameter mark.

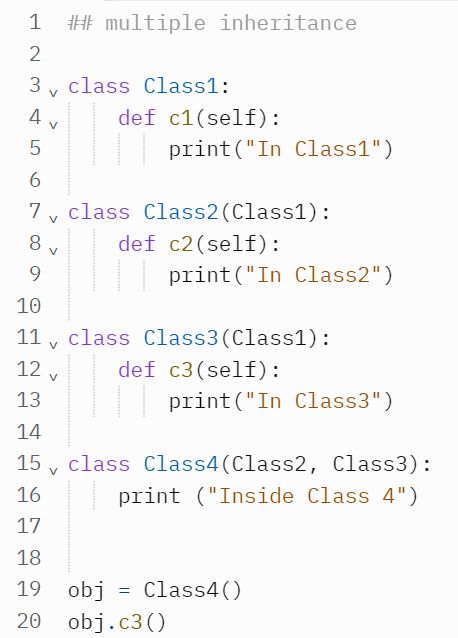
The Student class instantiated as follows:



Notice the extra parameter “20” added at the end, which is used to assign to the student object along with the parent Person attributes.

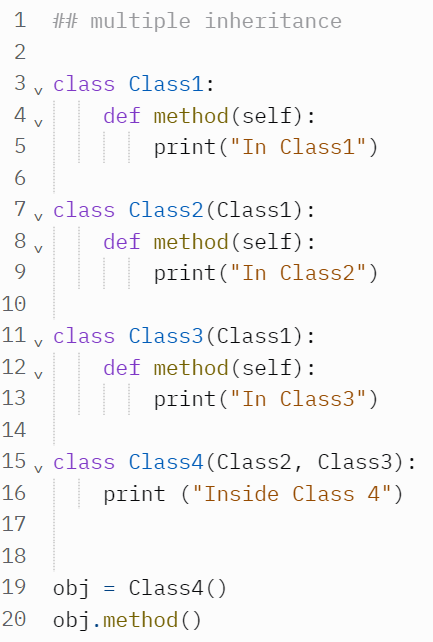
#### Instantiating from multiple parents

As well as inheriting from a single parent, it is possible to inherit the methods and attributes from two classes. This is simple achieved by inserting two class names in the class definition parameters:

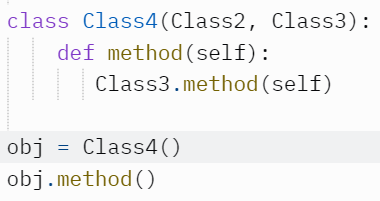


Inheriting from two classes.

This will output “In Class3”. If methods have the same name then the child overrides the parent:



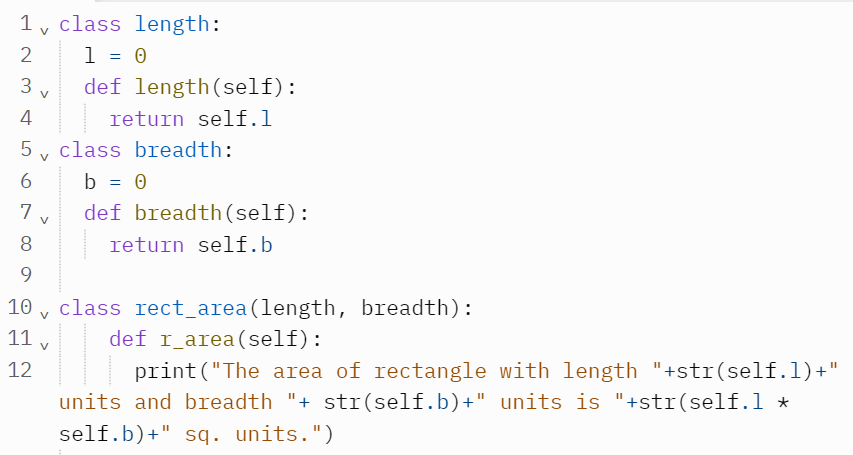
This will output **“In Class2**”, as Class2 is the first parent (overriding class3). You can refer directory to Class3 in the method:



This will output “In Class3”.

### Tasks

8.11 Construct the following methods and classes:



Extend the program so that the user enters the length and width of a shape. Instantiate the appropriate classes and assign values, calling the r\_area method to output the area of the shape.

## Association, Aggregation and Composition

So far, we have covered the basics of object oriented, from creating objects from classes, to methods, attributes, and inheritance. The next core principles of OO design are the relationships between classes and objects. This is where the concepts of Association, Aggregation and Composition come into play. Whilst this may vary across different programming languages, we can help to identify these main ideas in Python.

Association is what is regarded as the “least” relationship between objects, and is normally represented in a UML diagram, using a filled in arrow, as:

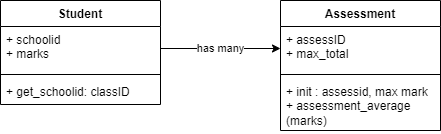


Figure 7 UML Association

Here we have a loose link, with students taking many assessments. Assessment information and methods held in the associated class. This can be represented in Python code as:

Graphical user interface, text, application

Description automatically generated

Only attributes of student object passed to Assessment method.

Objects instantiated outside the classes

Figure 8 Class Association

As demonstrated the attribute marks, containing a list of all assignment marks, from the student object is passed to a generic average calculating method assessment\_average in the assessment class. This will return the average mark. As the assessment\_average method can take any set of list values this is essentially a thin relationship i.e., **Association**.

The next, slightly tighter, relationship is aggregation, where the object itself in passed and used in a class. When the class is destroyed, the original object still exists. The class the object was passed to may use and access the objects properties and methods.

This can be illustrated in a UML, using a hollow diamond to indicate aggregation relationship:



Figure 9 aggregation, student has a class

Implemented in Python code:

Graphical user interface, text, application

Description automatically generated

Objects created outside class.

Student attribute points to student object passed.

Object passed to class.

When program run, the output is:

Graphical user interface

Description automatically generated with low confidence

If we changed an attribute in the student object this is also reflected in the class object:

A screenshot of a computer

Description automatically generated with medium confidence

Here we call the method aclass.student\_schoolid() which returns the school\_id attribute inside class, which points to the student object passed earlier.

Outputs:



If we were to reset or remove the class student object this would not impact or change the student object outside the class. This can be useful if you wish to use other objects properties and methods without removing or changing its structure as it may be used in other classes.

Finally, the last relationship is **Composition.** In composition, a class will instantiate an object outside the class so when the class is destroyed, the instantiated object is also removed. This can be represented in UML as a filled diamond:

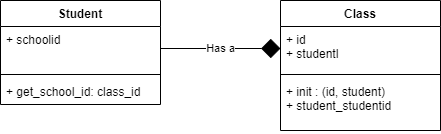


Figure 10 Composition

In Python this is achieved by instantiating the object inside the class:

Graphical user interface, text, application

Description automatically generated

### OO Project – Class Assignments

Using all the samples of code provide, and the completed tasks there is enough to start developing a complete Class Assignments project. The basic UML framework is provided below:

Diagram

Description automatically generated

Create an OO program which resembles the structure above. Try to add data structures, attributes and methods appropriate for the system. Provide a final MENU structure which allows the user to select a variety of actions to provide a USABLE system.

# ­Lesson 9 – SQL Databases

|  |  |
| --- | --- |
| **Lesson Outcomes**  In this lesson, you will learn:   * The basics of a database * What is SQL * Incorporating SQL into your program * Reading records * Writing records * Relational Databases * Object Oriented SQL setup | **C:\Users\Graham\AppData\Local\Microsoft\Windows\Temporary Internet Files\Content.IE5\X6CHINOH\MC900441498[1].png** |

## Introduction

As you have noted in previous exercises, storing program data permanently can be achieved using text files. Text files are a good and convenient way of storing small amounts of structured block data, like passwords and login details, however when processing more complex groups of structured datasets it becomes cumbersome and slow. Normally, in data-processing systems, we use what is known as **relational database** to store data. That is, data stored in **Tables**, where each table is related to one another. To extract data from these tables we use **Structured Query Language (SQL).** This is type of language specifically designed to insert, read, process and update data stored in tables. The program which compiles and executes SQL code is called the **SQL engine.** This program can either be stored on your desktop computer or a server.

If you imagine a typical setup:

Diagram

Description automatically generated

You may have many programs in different languages all use using the same SQL to send to the engine. The engine then updates or extracts from the main database with results sent back to the program. This allows us to separate the **PROGRAM** from the **DATABASE,** meaningyou may have many applications using the same central database. Whilst SQL is a standard language, there are slightly different variations depending on the company supporting and producing the product. That said, there are core commands which are used (*SELECT, UPDATE, DELETE, INSERT, and WHERE*),

## Create Database

Before embarking on a database system, we first need to create the backend, that is the database itself. In Python, you first import SQLite and make a connection to the database file, if doesn’t exist the file is created:

Text

Description automatically generated

**LINE 1** – imports the **sqlite3** module

**LINE 2** – imports error reporting. Useful for printing errors to help with debugging.

**LINE 4** – database filename. Without specifying the folder path, this will be in the same directory as this program.

**LINE 6 & 9** – We use the try, error commands. This is necessary as when you connect to the database there maybe errors such as the file not existing or incorrect file format, which we need to catch rather than terminating the program. Notice we use the Error library import to output the error.

**LINE 7** – create the connection and instantiate as conn object.

**LINE 8** – output the version. This is just for debugging purposes.

## Database Tables

When creating a standard database we use Tables, that is data stored in tabular fashion with columns identified as **FIELDS** and rows as **RECORDS**. Take for example, creating a database to store student details. We create one table:

Primary Key FIELD

FIELDS

**tblStudents:**

|  |  |  |
| --- | --- | --- |
| studentID | firstname | surname |
| 0001 | Harry | Smith |
| 0002 | Nyle | Magana |
| 0003 | Yuvraj | Frazier |
| 0004 | Anniyah | Cunningham |
| 0005 | Cruz | Lindsey |

A RECORD

The fields **StudentID**, **Firstname** and **Surname are identified as the main fields**. For the data set 5 records are to be inserted into the table. You will notice the field StudentID has been added, this is what we call a **PRIMARY KEY** – a field which **UNIQUELY** identifies a record e.g., “0002” uniquely identified “Nyle Magana”. In databases tables it is always necessary to add a primary key, whether containing one or multiple fields.

### Field Data Types

Having identified the basic table structure, that is field names, we now need to create the table in the database before we can insert the data. Each field will need to be assigned a data type and if appropriate, the size. A **DATA TYPE** can be:

|  |  |
| --- | --- |
| NULL | This no data. The field will allow null entry. |
| INTEGER | Signed whole number. |
| REAL | Decimal |
| TEXT | String / Text |
| BLOB | Binary Large Object |

So, given the above table, we assign the following types:

|  |  |
| --- | --- |
| studentID | TEXT (0001, has proceeding ZEROS) **PRIMARY KEY** |
| firstname | TEXT NOT NULL (must have an entry) |
| surname | TEXT NOT NULL |

**NOT NULL**, tells us the **FIELD** must have an ENTRY. **PRIMARY KEY**, the field identified which uniquely identifies a record.

### Create Table

Given the above structure we can use a series of SQL commands to create the table in the database. Here is the basic syntax:

**Text

Description automatically generated**

**LINES 1 to 10** Connect to database and put place error trapping.

**LINES 12-18** This is the SQL statement. Assigned to a STRING. Here we use the **CREATE** command.

**LINES 20** Executes the SQL using the conn object.

Notice the command CREATE is used with the table name Student, (look at naming conventions, <https://www.isbe.net/Documents/SQL_server_standards.pdf>). **IF NOT EXISTS** is used so that the program doesn’t throw an error because the table already exists.

The basic syntax structure for the CREATE command:

(SQLite, n.d.)Diagram

Description automatically generated

Figure 1 Create syntax structure (source SQLite.org)

For full SQL Lite guide: <https://www.sqlite.org/lang_createtable.html>

### Task

**TABLE NAME:** Product

|  |  |
| --- | --- |
| **Field Name** | **Data Type** |
| ProductID | TEXT PRIMARY KEY |
| ProductName | TEXT NOT NULL |
| UnitCost | REAL NOT NULL |
| Qty | INT |

9.1 Create a database with the above table structure.

### Amend Table

You may wish to include functionality in the program to amend an existing table. Let’s say we now wish to add the DOB to the student table.

Text

Description automatically generated

The above example adds the field DOB to the table. Note to add multiple columns you must create a new SQL statement.

### Task

9.2 Amend the table from 9.1 to include:

|  |  |
| --- | --- |
| **Field Name** | **Data Type** |
| Description | TEXT |
| Type | TEXT NOT NULL |

**HINT:** you will have to create two SQL statements executing each.

9.3 The following example code changes a field name:

*SQL* = """ALTER TABLE Student  
 RENAME COLUMN studentID TO studID  
 """  
*conn*.execute(*SQL*)

Change the field name in the table from 9.2 from description to desc

For further commands on deleting and adding, refer to: <https://www.w3schools.com/sql/sql_alter.asp>.

### Adding Data

Now you have created a table we need to add content. This is using the SQL INSERT command:

Graphical user interface, text

Description automatically generated

**LIINES 1 – 12 :** Setup database

**LINES 15-20 :** Create a list which contains multiple TUPLES

**LINES 26 :** Use the executemany command to run the SQL values statement with multiple values. Reference to the list Products. Notice each “?” mark in the values is replaced with values from the Products LIST.

**LINES 29-30 :** **Commit** to save data and close.

### TASK

9.4 Type the above program. Add another 2 records:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ProductID** | **ProducName** | **UnitCost** | **Qty** | **Desc** | **Type** |
| 0005 | Terracotta Pot | 60.00 | 10 | 40cm large Terracotta pot | POTS |
| 0006 | Large Pot | 30.50 | 50 | 100 Ltrs Pot | POTS |

### Querying

The great power of databases is the ability to extract data from large data sets using Queries. In SQL queries are performed using the SELECT statement which has the basic syntax:

**SELECT** *column*

**FROM** *table*

**WHERE** *condition*

Example:

Graphical user interface, text, application

Description automatically generated

**LINE 9:** **SELECT “\*”** means select ALL fields from the Product table. A loop is created to add each record to the row variable, which in turn is output.

You can filter the data displayed by using the WHERE condition in the query. So, for example say we just wanted to display all the products which are of type “POTS”:

Graphical user interface, text, application, email

Description automatically generated

Outputs:



**LINE 13:** Here we have added the placeholder “?” which will be replaced by the contents of the variable FILTER (placed in square brackets as need to be a LIST).

### TASKS

9.5 Using the product table creates. Create two queries to extract all products of type “TOOLS”.

9.6 Alter Task 9.5 to display in the following format:

Graphical user interface, table

Description automatically generated

**HINT:** refer to print out text FORMATTING in booklet 4.

### User Data Entry

We have learnt how to insert data into a table using the INSERT command using data predefined in a list. However, for a system we need to insert data as specified by the user.

Example:

Text

Description automatically generated

**LINES 18-20:** first a record LIST is defined with the record tuple, in the case one record (why the .execute method used as opposed to .executemany). This list is used as part of the SQL to insert records into the database. The conn.commit() command updates the database with the new data.

#### Error Trapping

When entering data, we must ensure to error trap the system. For example, in the above code the user may insert a Product Key that already exists therefore throwing an error, as the data in this field must be unique. To prevent the program exiting we use error trapping as follows, adapting the above code:

Graphical user interface, text, application

Description automatically generated

**LINES 25-33:** Here a while loop is created, so if an error is thrown the user is asked to enter data again. Notice user data entry has been placed in a function.

Displaying updated table data:

Text

Description automatically generated

### Data Updating

SQL uses the update command to update data as follows:

**UPDATE**  *tablename*

**SET fieldname =** *value*

**WHERE** *condition*

Graphical user interface, text

Description automatically generated

The above program will update the product name based on the id you entered, however there is no error checking. You may enter a product id that doesn’t exist, and an error isn’t thrown.

Adding an existence check:

Text

Description automatically generated

**LINES 17 – 19 :** Using a SELECT query to check if productid exists

**LINES 20-24 : U**sing the .fetchone() method to retrieve the first row of the record set as a LIST. If the row list is not empty, then the product exists.

### TASKS

9.7 Enter the code for “User Data Entry”, test it works with existing table “OrderInvoicing.db”

9.8 Enter the code for “Data Updating”, test it works with existing table “Order Invoicing.db”

### TASKS - Books Sample Database

Using the database file “books.db” included with this booklet, complete the following tasks.

Structure of books table:

FIELDS:

Graphical user interface, application

Description automatically generated

9.9 Setup connection code to the “books.db” database. Using a Select Query display all books by the author “J.R.R Tolkien”. Display in the following format (select the appropriate fields to display):

Text, letter

Description automatically generated

HINT: in the SELECT query just select 2 fields “book\_id” and “title”

9.10 Using the same connection display all books published before 1960.

Example:

A picture containing shape

Description automatically generated

# ­Lesson 10 – TKinter

|  |  |
| --- | --- |
| **Lesson Outcomes**  In this lesson, you will learn:   * The core components of TKinter module * Using basic Windows * Widgets and uses. * Creating an example program | **C:\Users\Graham\AppData\Local\Microsoft\Windows\Temporary Internet Files\Content.IE5\X6CHINOH\MC900441498[1].png** |

Contents

[Lesson 10 – TKinter 1](#_Toc122366104)

[Introduction 2](#_Toc122366105)

[Program Structure 2](#_Toc122366106)

[Adding Widgets 3](#_Toc122366107)

[Task 4](#_Toc122366108)

[Tasks 14](#_Toc122366109)

[Menus & Multiple Windows 16](#_Toc122366110)

[Task 16](#_Toc122366111)

[Data Entry and Processing – Student Marks System 18](#_Toc122366112)

[Tasks 21](#_Toc122366113)

## Introduction

TKinter is a module which comes with the installation of Python. In this module you are provided with a whole host of widgets to create a full Graphical User Interface application. To test whether you have TKinter installed correctly simply write the following code:



When run a Window should pop up indicating the TKinter version and a simple Click me! Test button.

Graphical user interface, text, application

Description automatically generated

### Program Structure

Before writing any GUI program using Tkinter it is important you recognise the basic structure.

**Step 1** – Import tkinter, and set as “tk”

Text

Description automatically generated with medium confidence

**Step 2** – Create main window “root”. As you instantiate the class Tk, you call the constructor method initiating the main root window. Set properties of Window.

**Step 3** – add widget(s).

**Step 4** –Run whole program (infinite loop)

### Adding Widgets

As you can see above, the first thing before creating any Windows is to import the TKinter module with a given name, in this case “tk”. Once you have imported tk and created the main window you may then add a number of Widget Classes (components on the window). The basic core Widgets are:

|  |  |
| --- | --- |
| **Widget** | **Description** |
| Label | To display text or image on the screen |
| Button | It is used to add buttons |
| Canvas | Used for graphics, pictures, text etc |
| CheckButton | It display a number of toggle options for the user to select |
| RadioButton | It is used to implement on-of-many selection, allowing only one option |
| Entry | Single text entry box |
| Frame | A container for many widgets |
| Message | It works same as that of label and refers to multi-line and non-editable text |
| Scale | Provides graphical slider which allows to select any value from that scale |
| Scrollbar | Provides up and down slide controller |
| Spinbox | User selets from give set of values |
| Text | Provides use with multiline text entry |
| Menu | Creates all kinds of menu |

Taking the first two, Label and Button we may create the following:

Graphical user interface, text, application

Description automatically generated

**LINE 9** Creates the label using the optional parameters to specify text, background and foreground colour. Every Widget has its own set of optional parameters which you lookup on the Python documentation ([https://docs.python.org/3/library/tkinter.ttk.html#label-options](https://docs.python.org/3/library/tkinter.ttk.html%23label-options)). When setting any option you normal specify the option name and assign a value to it, as you would a normal variable.

Notice the use of the methods “.pack()”, with default options. This is needed to place on the Window. You can specify parameters in the pack methods such as expand, fill, side (where you place the object within the window).

**LINE 10** Create a button, associated to the root window with text. Again, the pack method entered at the end to insert button in window.

### Task

As you have seen it is possible to add widgets at certain positions using the .pack() method. This has three properties – fill, expand, and side. Each property can be assigned a value.

|  |  |  |
| --- | --- | --- |
| **Side** | **Fill** | **Expand** |
| ‘left’ | ‘x’ | True |
| ‘right’ | ‘y’ | False |
| ‘bottom’ |  |  |
| ‘top’ |  |  |

* Place widget to fill entire frame.
* Place widgets on top of each other
* Place widgets side by side.

Example

tk.Label (*root*, text="Hello world ! ").pack(side = "left", fill="x", expand = True)

10.1 Using the above create the following window:

Graphical user interface, application

Description automatically generated

#### Placing Widgets

|  |  |  |
| --- | --- | --- |
| (0,0) | (1,0) | (2,0) |
| (0,1) | (1,1) | (2,1) |
| (0,2) | (1,2) | (2,2) |
| (0,3) | (1,3) | (2,3) |

As you have seen it is possible to place Widgets using the .pack() method however this if fairly limited and not very useful when dealing with complex layouts. The other method is using the **Grid Geometry Manager**. This splits up your window into a grid of Columns and Rows, with each cell give a grid reference (col, row):

Columns

Rows

|  |  |
| --- | --- |
| (0,0) | (1,0) |
| (0,1) | (1,1) |
| (0,2) | (1,2) |
| (0,3) | (1,3) |

Each row and column are given an index, by default first row is 0, 1 etc and column likewise. It is possible to configure the grid so not all the rows and columns are evenly placed. For example, you may use the columnconfigure method to change the weight or a row or column. You might want the second column to be two times wider than the first column:

Code Format is *Container.columnconfigure (index, weight*) e.g.

window.columnconfigure(0,0)

window.columnconfigure(1,2)

|  |  |  |
| --- | --- | --- |
| (0,0) | (1,0) | (2,0) |
| (0,1) | (1,1) | (2,1) |
| (0,2) | (1,2) - columnspan = 2 | |
| (0,3) | (1,3) | (2,3) |

As well as defining the column sizes relative to one another you may also wish to merge columns or rows:

This is particularly useful if you have a widget which needs to spread across more than one column.

Once the grid is defined you can then start to place the Widgets, referring to the row and col reference for position. For example:

Label.grid(row=2, column=0, columnspan = 2)

Example program:

Text

Description automatically generated

**LINES 9-11:** This configures the rows and columns. Notice a loop is created to iterate through 5 rows and 5 columns.

Creating the Widgets:

Text

Description automatically generated

Adding Widgets to the grid:

A screenshot of a computer

Description automatically generated with medium confidence

And finally, running program with loop:

Text, letter

Description automatically generated

Will output (window expanded):

Graphical user interface, application

Description automatically generated

To illustrate the Grid in the above, the window has been expanded. However, rather than expanding the window each time to see all the components we can resize so that this happens automatically when window is created. We can also change some basic properties of the window:

window.geometry (width x height + xpos + ypos)

*Here we can specify the exact size and position of the Window - width and height specifying the dimensions and xpos and ypos specifying the relative position from the corner of the screen.*

window.resizable (False, False)

*Either making the Window not resizable both horizontal and vertical.*

window.minsize (0,0)

window.maxsize(400,400)

*Window min and max resize.*

Container.attributes (‘-alpha’, 0.5)

*Making the Window transparent.*

For example:

Text

Description automatically generated

#### Aligning Widgets – Sticky

When you place a Widget into a grid cell by default it will align to the center. However, it is possible to change the alignment using the sticky option. The value of sticky has the following valid values:

|  |  |
| --- | --- |
| **Sticky** | **Description** |
| N | North or Top Center |
| S | South or Bottom Center |
| E | East or Right Center |
| W | West or Left Center |
| NW | North West or Top Left |
| NE | North East or Top Right |
| SE | South East or Bottom Right |
| SW | South West or Bottom Left |
| NS | NS stretches the widget vertically. However, it leaves the widget centered horizontally. |
| EW | EW stretches the widget horizontally. However, it leaves the widget centered vertically. |

#### Padding

Along with Sticky we can also add padding, that is the margins in and around the cell.

For example, for internal paddings i.e., within the cell.

Grid (column, row, sticky, padx, pady)

External padding (outside the cell):

Grid (column, row, sticky, padx, pady, ipadx, ipady)

Example program, setup Window and Grid properties:

Graphical user interface, text

Description automatically generated

Create Widgets:

Text

Description automatically generated

Place on Grid:

Graphical user interface

Description automatically generated with medium confidence

Run program:



When run will produce:

Graphical user interface, application

Description automatically generated

#### Buttons and Events

To make buttons useful we must add functions which are called in the event the button is pressed. This can simply be achieved by setting the command property of the button:

Button (window, text = “Hello”, command=output\_hello)

In this example we set the command to run the function output\_hello.

Using this method, we can setup multiple buttons to change the properties of labels.

Example program:

Setting up window:

Text

Description automatically generated

Setting up functions for each button:

Text

Description automatically generated with low confidence

Notice we use the method .config to reconfigure the label properties.

Configure grid:

Text

Description automatically generated

Create Widgets.

Text

Description automatically generated

**LINES 26-27:** Using the command property to assign the corresponding functions.

Insert the Widgets onto the grid.

Text

Description automatically generated

Create main loop for program.

Text

Description automatically generated

As well as creating buttons with corresponding functions we can also capture **EVENTS**. An event is something that is activated by a USER such as moving the mouse, clicking, double clicking, selecting, button release. These events can be bound to what we call a handler – the function which runs when the event happens.

TKinter supports the following events – Button, Motion, ButtonRelease, DoubleButton, Enter, Leave, FocusIn, FocusOut, Return, Key, Shift-up, Configure (further information can be found with examples <https://python-course.eu/tkinter/events-and-binds-in-tkinter.php>.

Let’s add a simple BUTTON event to the label, notice the parameter “event”, this must be specified:

Graphical user interface, text, application

Description automatically generated

Attach the event BUTTON clicked to the label, and assign the handler function “clicked\_label”

Text, letter

Description automatically generated

Program output:

Graphical user interface, application

Description automatically generated

### Tasks

10.2 Create a program which displays the following:

Graphical user interface, application

Description automatically generated

When the user clicks the buttons Green, Red, Blue or White the square changes the colour accordingly. The label should display which colour that is currently selected.

**HINTS:**

You can create a shape using the Canvas widget.

*canvas* = tk.Canvas (*root*, width =40, height = 50)

*canvas*.create\_rectangle( 0,0,50,50, fill = "blue")

Note create\_rectangle method using x,y coordinates starting from top left to bottom right.

Clearing Canvas to allow the square to be re-drawn:

*canvas*.delete("all")

For more efficient code rather than writing a function for every button we can utilized the lambda command (<https://www.w3schools.com/python/python_lambda.asp>):

*button1* = tk.Button (*root*, text = "Green", command= lambda: button\_pressed(1))

This allows use to call a function with PARAMETERS.

10.3 Use another import **“ttk”** to gain further Widgets:



Using a **combobox** component of **ttk** create the following interface:

Graphical user interface

Description automatically generated with medium confidence

The dropdown will have 3 options – Square, Rectangle and Circle. Selecting any will result in the corresponding shape appearing on the left of the Window.

To complete this program, you will need to BIND combox CHANGE event and get the current selected option.

First create a combobox using the ttk import, then ASSIGN the bind event so a function is called once selected. In this example the function “option\_ selected” called. Also set the current selection (index 0):

Text

Description automatically generated

From the function it is then possible to get the current selected item in the Combobox:



Don’t forget to DELETE ALL the contents from the canvas to allow a redraw of the shape. For a Circle you will need to use the OVAL command:



## Menus & Multiple Windows

When you start TKinter application you create an instance of the TK class which essentially becomes your main application. When this is closed or destroyed the whole application terminates. Therefore, we refer to the main window as the “ROOT”, and label accordingly. If we want to create multiple windows, we use the Toplevel class from which we can create many instances. For example, to create a new Window we simply:

Text

Description automatically generated

### Task

10.4 Create a main Window with two buttons. Each button when pressed will open a new Window. For example, clicking Window 2 button below will open another appropriately labelled Window.

Graphical user interface, application

Description automatically generated

In addition to multiple windows, you may also wish to create a simple drop-down menu which would also open another window (reference <https://www.tutorialspoint.com/python/tk_menu.htm>) :

Text

Description automatically generated

**LINE 53:** Create a menubar

**LINE 54:** Create main menu and attach to menubar (you may have multiple dropdowns)

**LINE 55-57:** Create listed options with corresponding function calls. Included a separator.

**LINE 58:** Add mainmen

10.5 Add to Task 10.3 so that a Menu option can be selected to open each Window.

Graphical user interface, application, Teams

Description automatically generated

## Data Entry and Processing – Student Marks System

So far, we have used just the basic Windows and Widgets to create simplistic interfaces. The next step is to create a working program. For this we are going to go over a simple Student Marks system - taking information from the user for processing, displaying, and storing student assessment data.

For this system we are going to create a simple record entry form, where you can view a student details, edit records, and look at and enter/ edit marks. Here is a simple window design:

**Grid Lines**

|  |  |  |
| --- | --- | --- |
| **Labels** | **Entry box** | **Listbox** |
|  |  |  |

Column 3

Column 2

Column 1

Column 0

Frame -columnspan = 4

Row 0

**Student Details**

Row 0

Marks

Student ID:

Row 1

Row 1

|  |
| --- |
|  |
|  |
|  |
|  |

First name:

Row 2

Surname:

Row 3

DOB:

Row 2

Notice the grid layout to get idea of positioning. The second row will need a heavier weighting as slightly bigger. The student details will be enclosed in a frame, which in turn has its own rows and columns. Row 2 will be used for navigation / command buttons, which will encompass within a frame.

#### Creating frames adding textboxes

Setup window:

Text

Description automatically generated

Setup main grid:

Text

Description automatically generated

Create a frame, spreading across 4 columns using the columnspan option. Place in row 1.

Graphical user interface, text, application

Description automatically generated

Create labels and edit fields, with padding and sticky set:

Text

Description automatically generated

Here we add a LISTBOX to display marks. Added to frame, spanning four rows:

Text

Description automatically generated

Graphical user interface

Description automatically generated

### Tasks

10.6 Create the above form. Adjusting as necessary.

#### Adding Navigation and Save Buttons

10.7 Using the form created in 10.6, add Navigation buttons first record, last record, next record, last record, and new record. Create a new frame for all buttons.

Graphical user interface

Description automatically generated

Assign functions to each button (use the command pass as we will enter code later)

Graphical user interface, application

Description automatically generated

#### Linking to a database

10. 8 Viewing records

Now a basic form has been created we need to populate with data. For this an SQL lite database will be used to extract and store data. Use the database “StudentData.db” included with these booklets.

Amend existing code to add connection to database:

Graphical user interface, text, application

Description automatically generated

**LINE 12:** fetch all the records for the student table and assigning to the list students.

Near the end of the code, we going to add a global variable called record\_no. This will keep track of that row we are currently looking at from the students list. Using global variables isn’t ideal but for the sake of simplicity we will keep it her. We will also call a function called display\_record with the parameter record\_no – this will display the current selected record on the form.

Text

Description automatically generated

The function display\_record:

Graphical user interface, text, application, email

Description automatically generated

**LINES 27- 35** This simply refreshes each entry box with the contents of the currently selected record, using the record\_no as the index for the students list. Each field in the row (student) is referred to starting from 0 to 3.

**LINE 38** Run a query extracting all the marks for this selected student, use the primary key student\_id. Assign to the student\_marks.

**LINE 42** Iterate through student\_marks, updating line by line the listbox.

Now we can add function code attached to each button. The first two should look like this:

Graphical user interface, text, application

Description automatically generated

**LINES 53- 54** Checking that we can’t go back beyond the first record.

Add code, except for the NEW button, for the remaining. Use the **Len** command to find the number of records in students.

#### Updating the database

10.9 The next step in this system is to update changes made on the form to the database (not the Marks).

For this to work we need to create a function which updates the database table with the contents of the form. Using the date entry fields and the listbox we can update Students\_list and with SQL update command to make changes to the database.

Graphical user interface, text, application

Description automatically generated

**LINES 54-59** Create a List using the form entry boxes. Update the currently selected form into the main students List.

**LINES 61-64** Update the selected record in the database.

Add this function to the Save Button.

10.10 a Add the facility to create a new record. Using a button with corresponding function to insert a new record.

Creating a new record requires a little more complicated processing, as not only do you have to insert a whole new record, but you also have to assign a primary key in order for the record to be inserted.

One common way of doing this to use a **HASH** function which is a way of generating a number based on a STRING input and then we can use a MODULUS calculation to generate a 9-digit number. We can use the SURNAME plus DOB for this, which hopefully should generate a new unique number each time. We could add further processing so that if a number is generated that already existed in the database, we could create a system to generate another one-up approach. For further information look up [HASHING records methods](https://isaaccomputerscience.org/concepts/dsa_datastruct_hash_table?examBoard=all&stage=all).

Another, simpler approach, is to just use the max **RECORD NO in the records and add 1**. This will be a continue incrementing number regardless of deleted records. For simplicity, we will use this approach:

Text

Description automatically generated

**LINE 137** Create new ID, using MAX command for student id across all records.

**LINE 138** Allow change to field and then set back to read-only.

**LINES 138- 144** Clear all entry fields.

**LINE 144-145**  Create new ID. Update students list with new record calling create record (below)

Text

Description automatically generated

10. 10 b Add another button to which when clicked will allow the user to add a test mark – this will call up another window with an entry box to enter mark, with OK, or CANCEL option.

Text

Description automatically generated

**LINES 84-87** Create new window. Use the grabset() method to disable the use of other windows while this is active.

**LINES 90-94** Create labels and Entry boxes adding to the current window.

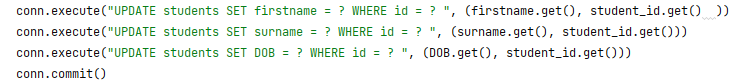
**LINE 95** When button selected call the markOK function, passing event, mark, and the marks\_window as parameters.

**LINE 96** Using the destroy method to close window.

10. 10 c Allow the user to select a Mark from the list box and double click to edit, again opening another window to enter/ change mark.

**HINTS:**

Updating tables:

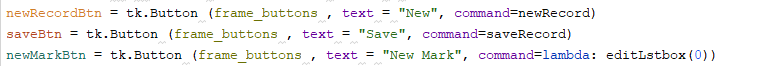


Adding a binding with event (1 meaning edit, 0 meaning new mark).

Chart

Description automatically generated

Creating Buttons:



Another function (markOk) for saving data on the OKButton on the pop-up window, is it inserting or updating the table?

Graphical user interface, text, application

Description automatically generated

Notice the destroy method for closing a window.

10.11 Add a new button and function to delete a record. You will not only need to remove data from the working students list, but also the database.

**HINT:** it is good practice to check first if the user wants to delete the record by use of confirmation. To save time creating another window we can use the built in function askokcancel:

A picture containing text

Description automatically generated

The variable answer will be either True or False.

System complete. You now have a working application to view student details, edit, add, delete, and add and edit marks.

#### Possible Extensions - explore.

Add functions to delete marks, produce statistical data e.g., max mark, average, lowest mark.

You may also use imported packages [(https://packaging.python.org/en/latest/tutorials/installing-packages/](https://packaging.python.org/en/latest/tutorials/installing-packages/)

Such as matplotlib, numpy and pandas. These will give you additional powerful tools for creating charts and statistical data.

Graphical user interface

Description automatically generated