



Professional Development | An tSeirbhís um Fhorbairt
Service for Teachers | Ghairmiúil do Mhúinteoirí

www.pdst.ie



LEAVING CERTIFICATE
COMPUTER SCIENCE

Fundamental Skills Development

Python

PDST

Python Programming

A Manual for Teachers

of

Leaving Certificate Computer Science



This work is made available under the terms of the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Ireland Licence <https://creativecommons.org/licenses/by-nc-sa/3.0/ie/>.

Please cite as: PDST, *Leaving Certificate Computer Science v2.0, Python Workshop*, Dublin, 2022

Table of Contents

Section 1 – Getting Started	1
Hello World	2
Language Syntax	3
Basic Python Syntax	5
Escape Sequences	8
Flow of Control	10
Programming Exercises	12
BREAKOUT ACTIVITIES	13
Section 2 – Data, Variables, Assignments and Expressions	23
Introduction	24
Variable Syntax	26
Datatypes and Operations	29
Program Tracing	31
Input-Process-Output	34
More Built in Functions	40
The Remainder Operator (%)	41
Programming Exercises	43
Running Totals	45
Introducing Random Numbers	47
Additional Notes	49
BREAKOUT ACTIVITIES	51
Section 3 – Strings	63
Introduction	64
String Indexing	66
String Slicing	70
String Addition and Multiplication	71
String Formatting	74
Built-in String Commands	76
Coding Systems	77

Programming Exercises	79
String Methods	80
Additional Notes (Sequences)	83
BREAKOUT ACTIVITIES	85
Section 4 – Lists	95
Introduction	96
Creating Lists	97
Common List Operations	99
List Indexing	101
List Slicing	105
List Methods	109
Two More String Methods (<code>split</code> and <code>splitlines</code>)	111
BREAKOUT ACTIVITIES	113
Section 5 – Programming Logic	120
Introduction	121
Hangman!	122
Boolean Expressions	124
The Guessing Game	128
Selection (<code>if</code> , <code>else</code> , <code>elif</code>)	129
BREAKOUT ACTIVITIES (selection)	138
Iteration (<code>for</code> , <code>while</code>)	141
BREAKOUT ACTIVITIES (iteration)	160
Section 6 – Modular Programming Using Functions	168
Introduction	169
Basic Function Syntax	177
Function Parameters and Arguments	182
Function Return Values	187
Examples and Exercises	190
Boolean Functions	194

Using Functions to Validate Data	199
Programming Exercises 6.1	201
Recursion	205
Programming Exercises 6.2	209
Programming Exercises 6.3	210
BREAKOUT ACTIVITY 6.1 (ATM System)	214
BREAKOUT ACTIVITY 6.2 (Summing Numbers)	224
BREAKOUT ACTIVITY 6.3 (Turtle Graphics)	228
BREAKOUT ACTIVITY 6.4 (Check Digits)	234
Section 7 – Dictionaries	242
Introduction	243
Dictionary Definitions – some examples	244
Creating Dictionaries – syntax	247
Indexing Dictionaries	249
Adding, Changing and Deleting Dictionary Elements	255
Programming Exercises 7.1	263
Iterating over dictionaries	268
Dictionaries and Lists	271
BREAKOUT ACTIVITY 7.1 (Frequency Counters)	273
Appendices	280
Appendix A: Python Keywords	281
Appendix B: Python Built-in Functions	281
Appendix C: Python Assignment Operators	282
Appendix D: Python Arithmetic Operators	282
Appendix E: Python Relational Operators	282
Appendix F: Truth Tables for not, and, and or	283
Appendix G: Sample Solutions to Selected Problems	284

Manual Overview

The purpose of this manual is to provide Leaving Certificate Computer Science (LCCS) teachers with an enhanced knowledge of the Python programming language thereby enabling them to independently improve their own programming skills.

Although the manual will serve as support material for teachers who attend the Python Skills Workshops which are a key component in our two-year CPD programme for LCCS teachers, it is envisaged that its real value will only come into play in the weeks and months after the workshops have been delivered. Beyond these workshops, the manual may be used as a basic language reference for Python, but more importantly, as a teaching resource that might be used to promote in teachers, a constructivist pedagogic orientation towards the planning and teaching of Python in the LCCS classroom.

The manual itself is divided into seven sections as follows:

- sections 1 and 2 cover basic, beginner programming concepts such as program execution, flow of control, simple datatypes, variables, assignments and expressions
- sections 3 and 4 cover the two datatypes - strings and lists – and associated operations
- section 5 covers programming logic – specifically, Boolean expressions, selection and iteration
- sections 6 and 7 cover more advanced Python programming topics such as functions and dictionaries

Of course there is much more to Python than the material covered in this manual. Among the more notable topics that are not explicitly addressed are, object-oriented programming and classes, list comprehensions and exceptions. That said, every effort has been made to ensure that the content contained here is adequate in order to mediate LCCS in the classroom.

Throughout the manual there are lots of examples and related exercises. Readers will find it helpful if they read (and try) the examples before attempting the exercises. The source code from most of the examples are available to download from the PDST GitHub repository (<https://github.com/pdst-lccs/lccs-python>) and links to sample solutions for many of the programming exercises as are available at the end of the manual.

A high-level overview of the content of this manual is presented below.

Section 1 – Getting Started

The aim of this section is to get readers up and running. By the end of this section participants should have a basic understanding of program execution, sequential processing, strings, escape sequences and the importance of language syntax. Participants will have the opportunity to write and modify simple programs. The section concludes with a practical lab/breakout session which introduces turtle graphics and games programming using the `pygame` library.

Section 2 – Data, Variables, Assignments and Expressions

This section will provide participants with a broad overview of the use of variables, assignments and expressions. Participants will have the opportunity to learn how to initialise simple variables as well as use an assignment statement to change its value. The importance of data and datatype is made clear. Arithmetic expressions and the input command are introduced. Participants learn how to apply these concepts through program tracing, testing, the use of the remainder (modulus) operator, running totals and random numbers. The section concludes with a practical lab/breakout session which as well as building on some of the 'projects' started in Section 1, introduces some basic file i/o operations.

Section 3 – Strings

In this section we will cover strings – basic sequence operations such as concatenation, multiplication, indexing and slicing will be explained. Example programs will extend thinking on coding systems and ciphers, and draw on the use of built-in functions - `ord` and `chr`. String specific methods and formatting will also be explained. Participants will be given a hands-on tour of the official online Python reference at <https://docs.python.org/3/>. The section concludes with a practical lab/breakout session where participants will be given an opportunity to write programs to generate web pages and analyse text from live RSS feeds.

Section 4 –Lists

The aim of this section is to extend participants knowledge of sequences through the concept of lists. Motivation is provided through a discussion on the many real-world applications of lists. List construction, indexing and slicing are explored in greater detail. The section describes the most common list specific methods and how to use them. Examples of how to use `split` and `splitlines` to generate lists are provided. The breakout activities

at the end of this section includes the use of lists to construct random sentences, further statistical analysis of data read in from a file of people's heights, and finally, a program to use lists as a basis for giving directions to a graphic turtle object.

Section 5 – Programming Logic (selection/conditions and iteration/loops)

This section explains the syntax and semantics of a number of programming constructs such as `if`, `elif`, `else`, `while` and `for` statements. The emphasis throughout is on application i.e. recognising situations where it is more appropriate to one of these control structures over the other. Many of the examples are layered, based on a guessing game program. The section concludes with a breakout session where participants will be able to consolidate their learning and further develop their project work from the previous breakout sessions. The use of `plotly` to present data in graphical format will also be introduced.

Section 6 – Modular Programming using Functions

The purpose of this section of the manual is to explain how functions can be used to organise programs into logically related units of code. The architecture of a typical Python program is presented and user-defined function are distinguished from built-in and library functions. The syntax and semantics for defining and calling functions is explained, as is the use of arguments/parameters to pass information into functions and return values to pass information out of functions. The examples build on the programming concepts covered in earlier sections and cover topics such as temperature/distance conversions, compound interest/future value calculations, and maximum values. The use of Boolean functions to perform tests such as to determine whether a number is prime or a given year a leap as well as to validate data is explained. The chapter contains many programming exercises designed to elicit the use of functions and structured programming and the topic of recursion is also explored in some detail. The section concludes with a number of practical lab/breakout session which can be adapted for use in the LCCS classroom.

Section 7 – Dictionaries

The purpose of this chapter is to provide a full overview of the dictionary data structure. Particular emphasis is placed on discussing the similarities and differences between lists and dictionaries. The final breakout session – based on frequency counting – is designed to elicit the computational thinking skills such as abstraction, pattern recognition, decomposition and algorithmic thinking.

Conventions

To help with navigation through this manual, the following conventions are used:

- ✓ *Italics* are used to highlight important new words and phrases defined
- ✓ `Courier New` font is used to denote Python code such as keywords, commands and variable names

The icons illustrated below are used to highlight different types of information throughout this manual.



Space to answer questions using pen and paper.



Python syntax rule.



Key technical point. A specific piece of information relating to some aspect of programming.



Experiment. An opportunity to change code and see what happens.



Programming exercises. An opportunity for individuals/pairs to practice their Python programming skills



Breakout Group Work. At the end of every section, readers will work on a number of themed projects relevant to that section.



Reflection log. A space for the reader to reflect on their own learning and record their thoughts.



Meet octocat! This is the GitHub integration symbol. Throughout this manual you will notice this symbol appears along with the example code. When you click on the octocat you will be directed to the source code on GitHub. Readers are recommended to copy the code from GitHub to their preferred Integrated Development Environment (IDE).

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

Blocks like the one shown above contain example Python code

STUDENT TIP
Practice! Practice! Practice!

Boxes like these contain key messages to pass on to novice programming students.

Section 1

Getting Started


Installation and Setup

All of the examples in this manual were tested using Python 3.6.4. A standard installation of Python 3.x and IDLE should be sufficient to run most of the examples and complete the exercises in this manual. However, for some examples/exercises it will be necessary to install third party libraries such as `pygame` and `plotly`.

Hello World

From your IDE create a new file (**File -> New File**).
Key in (or copy+paste from GitHub – see Key Point)
the Python statement exactly as it appears here:

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



KEY POINT: Instead of keying in this code you could click on the octocat and your browser will direct you to the GitHub repository with this source file. From this window you can select and copy the code and then paste it into your IDE.

Save the file (**File -> Save**) and press F5 to run.

If you see the text `Hello World` displayed in the shell window congratulations – you are up and running. The output of the program is displayed in the shell window (output console).

Throughout this manual we will be creating new files, typing in or downloading the example Python code provided. The aim is to get to the point where we can write our own code.

As teachers we should keep in mind that learning to program for the first time can be tricky – there can be lots of stuff going on at the same time, and understanding the syntax of Python can often seem to be more important than the real purpose of programming which is to automate solutions to well defined problems.

The sooner students overcome the initial *syntax barrier*, the sooner they can focus on the skill of problem solving and specifically the skill of using the features of Python to solve problems.

Teachers should continually emphasise to novice programmers that Python is just a tool, and the key skill lies in its application to solve problems.

Another point well worth getting across to novice programmers at an early stage is the difference between *programmers* and *end-users*.

- Programmers usually work as part of a team. They write and test the code that makes up a computer system. Student programmers should be encouraged to bear the needs of the end-user in mind i.e. see the system from the perspective of the end-user.
- An end-user is the person (or organisation) for whom a software system is developed. End-users are the customers and, very often, do not know how to program.

Language Syntax

Most of us are already aware that natural languages such as English, French, German, Polish etc. have their own rules. These rules make up the language *grammar*. The syntax of a language is that part of the grammar which defines how sentences are constructed – syntax is mostly concerned with legitimate words, symbols and the order in which they are used.

In a similar way, all programming languages (e.g. Python, Java, JavaScript, C++, PHP, Perl etc.) have their own syntax – this is called the *language syntax*.

One important aspect of Python's syntax is its vocabulary i.e. the words and symbols that Python understands.

Words can be keywords or commands. The list of all of Python's 33 keywords is given below – only some of these will be needed for LCCS.

False	break	else	if	not	while
None	class	except	import	for	with
True	continue	finally	in	pass	yield
and	def	for	is	raise	
as	del	from	lambda	return	
assert	elif	global	nonlocal	try	

Python 3.6.2 keywords

Programmers can add to Python's vocabulary by defining their own words.

The most common kinds of symbols in Python are operators – these can be arithmetic or relational.

Python also understands white spaces (e.g. spaces, tabs, newlines), numbers and strings (anything enclosed in quotation marks) – more on these later.

All programs must adhere to the syntax of the programming language in which they are written. When a program does not conform to the language’s syntax it is said to contain a *syntax error*. Such programs are said to be *syntactically incorrect*.

When you try to run a program that has a syntax error, Python displays a syntax error message.

Comments are a way to tell Python to ignore syntax. They are used by programmers to improve the readability of their code for the benefit of other programmers. Comments in Python start with the hash character, # , and extend to the end of the physical line. When Python comes across the hash character it ignores the rest of the text on that line

Reflection

Reflect on what you have learned about Python so far.



Use the space below to write **five** things that Python understands.

1.

2.

3.

4.

5.

Basic Python Syntax

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

Syntax Check #1:

Python is case sensitive. This means that Python treats upper and lower case letters differently. For example, Python understands `print` but does not understand `Print`

Try running the following:

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



You will see a message like this:

```
Traceback (most recent call last):
  File "C:/PDST/Python Workshop/src/hello1.py", line 1, in <module>
    Print("Hello World")
NameError: name 'Print' is not defined
```

This is Python's way of telling the programmer that the program contains a syntax error. Python keywords and commands must all be typed in lower case.



Experiment!

Try the following line.

Make some changes - what happens?

```
PRINT("Hello World")
```

STUDENT TIP

Students should be encouraged from an early stage to learn how to deal with syntax errors. One way to build student confidence is to get students to fix syntactically incorrect code (and even deliberately create and fix their own syntax errors).



What did you learn?

Syntax Check #2:

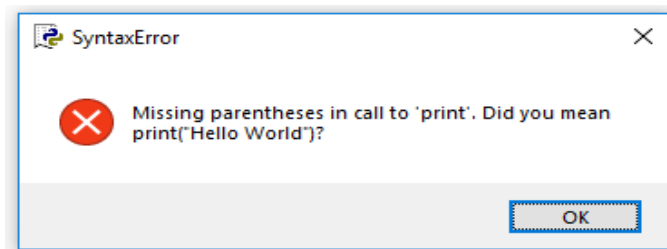
Use opening and closing brackets (parentheses) after the `print` command when you want to display output. For every opening parenthesis there needs to be a matching closing parenthesis.

Try running the following:

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



You will see a syntax error displayed in a message box like the one shown below because the opening parenthesis is missing.



Experiment!

Try each of the following 3 lines separately.

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

```
print "Hello World"
```

```
print"Hello World"
```



What did you learn?

What happens if you put a space either side of the brackets?

Syntax Check #3:

When you want to display text using the `print` command, the text must be enclosed inside matching quotation marks. If either, or both, quotation marks are missing, a syntax error message is displayed.

Experiment: Try each of the following lines separately.

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

Quotation marks can be single (`'`) or double (`"`) – it does not matter as long as they match.

Python is not too fussy about what you type inside quotation marks. Outside quotation marks, Python is very limited in what it understands. One thing Python understands outside quotation marks is number. Numbers do not have to be enclosed inside quotations.

Each of the following lines are syntactically correct. (Try them!)

```
print("What is your age? ")
print(21)
print("My age is 21")
print("My age is", 21)
print("I am", 18, "and my friend is", 21)
print("The print command", "can handle", "more than 1 string.")
```



KEY POINT: The technical word for text is *string*. A string is any text enclosed inside quotation marks.

Notice from the fine three lines in the above example how `print` allows strings and numbers to be separated by commas.



Write a Python program to display the text *Hello, my name is Sam!*

Escape Sequences

Let's say we wanted to display the following text exactly – including the quotation marks.

In the words of Nelson Mandela, "Education is the most powerful weapon which we can use to change the world"

The line below does not work because in the 'eyes' of Python the second quotation closes the first and the remainder of the line is not understood.

```
print("In the words of Nelson Mandela, "Education is the most powerful weapon which we can use to change the world")
```

To fix the syntax error we *escape* the second quotation using the backslash character, \, as follows:

```
print("In the words of Nelson Mandela, \"Education is the most powerful weapon which we can use to change the world\")
```



In the above example the use of \" tells Python include the double quotes as part of the string (as opposed to treating it as the closing quote).

The backslash character introduces an *escape sequence* in a string. Some common escape sequence characters are illustrated in the table below:



Escape Sequence	Meaning
\n	Newline
\t	Tab
\'	Single Quote
\"	Double Quote
\\	Backslash



Experiment!

Try the following and see if you can explain what is going on.

```
print("A\tB\nC")  
print("C:\\Users\\johndoe\\Documents\\myfile.txt")
```



What was the main thing you learned in this section about escape sequences?




What one question about escape sequences do you still have?

Flow of Control

The *flow of control* refers to the order in which the lines of a computer program are run by the computer. Normally lines are executed in the same sequence in which they appear. This type of flow is called sequential. We use the following four-line program to illustrate this concept.

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



When this program is run, execution starts at line 1 which causes the string, *As I was going out one day*, to be displayed on the output console. Execution then moves sequentially through lines 2, 3 and 4 and finally, the program ends as there are no more lines to execute.

The table below illustrates the program output be after each line is executed.

Line Number	Program Output
1	As I was going out one day
2	As I was going out one day My head fell off and rolled away,
3	As I was going out one day My head fell off and rolled away, But when I saw that it was gone,
4	As I was going out one day My head fell off and rolled away, But when I saw that it was gone, I picked it up and put it on.

STUDENT TIP
Students should be encouraged, from an early stage, to trace the execution process in their own heads, before submitting their code to the computer for execution. (This activity is called *program tracing*.)

In reality, we only see the final output after line 4 is executed – this is because the program is executed so fast by the computer. Nonetheless, it is important for students to understand that for the computer to get to the final stage it had to pass through the other stages on the way.

Indentation

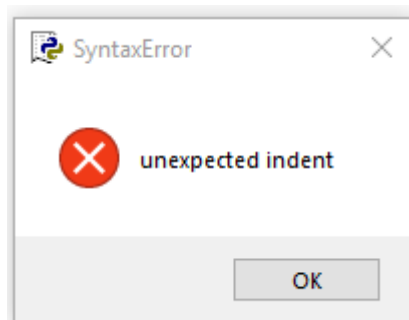
Indentation refers to the empty space(s) at the beginning of a line of code.

Python is very fussy about indentation - try to run the following:

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

Notice that the second line contains a leading space. This is an *indentation error*.

The following syntax error is displayed when a program contains an indentation error.



Syntax Check: Every line of Python code must be properly indented.

Proper indentation means logically related lines (called blocks of code) appear at the same level of indentation.

In the above example indentation is not needed but – as we will see later – it is sometimes necessary to indent code.



Programming Exercises

1. Write a program to display your own name and address.
2. Re-arrange the lines of code below into a program that displays the pattern shown on the right. Note that you can use any line as often as you like, but you won't need to use every line.

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

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

3. Reflect on what you have learned about Python so far. Use the space below to write *three* things that Python likes and *three* things that Python does not like.



Python likes ...

1.

2.

3.



Python does not like ...

1.

2.

3.



BREAKOUT ACTIVITIES

The focus on these activities is on getting used to the Python programming environment and in particular sequential flow of control.

BREAKOUT 1.1: Automated Teller Machine (ATM) Menu System

The Python program below displays the ATM menu shown on the right hand side.

```
# This code displays the main ATM menu
print("\t|-----|")
print("\t\t LCCS BANK LIMITED\t|")
print("\t\t ATM Main Menu\t\t|")
print("\t\t\t\t\t\t\t|")
print("\t\t\t1. Balance Enquiry\t|")
print("\t\t\t2. Cash Lodgement\t|")
print("\t\t\t3. Cash Withdrawal\t|")
print("\t\t\t4. Cash Transfer\t|")
print("\t\t\t5. Change PIN\t\t|")
print("\t\t\t6. Other Services\t|")
print("\t\t\t\t\t\t\t|")
print("\t\t\t7. Exit\t\t\t\t|")
print("\t|-----|")
print("\t\t\t\t\t\t\t|")
print("\t\t CHOOSE AN OPTION >> \t\t|")
print("\t\t\t\t\t\t\t|")
print("\t|-----|")
print("")
```



```
LCCS BANK LIMITED
ATM Main Menu

1. Balance Enquiry
2. Cash Lodgement
3. Cash Withdrawal
4. Cash Transfer
5. Change PIN
6. Other Services

7. Exit

CHOOSE AN OPTION >>
```

Suggested Activities

1. Key in the above program or download it from GitHub and
 - make some changes to the program (e.g. add/remove/edit a menu option)
 - discuss traditional console menus vs. GUI/touch screen interfaces
 - discuss possible logic behind the options
2. Design and implement a menu for some other application of your choice e.g. what are the options on your favourite app? What additional options would you like?
(For this exercise it is useful to think of a system from an end-user's perspective.)

BREAKOUT 1.2: Turtle Graphics

Turtle graphics is a popular way for introducing programming to novice programmers. It was part of the original Logo programming language developed by Wally Feurzig and Cynthia Solomon in consultation with Seymour Paper in 1966.

The movements of the turtle graphic object can be compared to the movements that you would see if you were looking down at a real turtle inside a rectangular shaped box. The program below causes the shape/pattern shown to the right to be drawn out on the screen.

```

1. from turtle import *
2.
3. forward(100) # move forward 100 units
4. left(90)    # turn left by 90 degrees
5. forward(100)
6. right(45)
7. forward(50)
8. left(90)
9. forward(100)

```


Program Listing



Shape

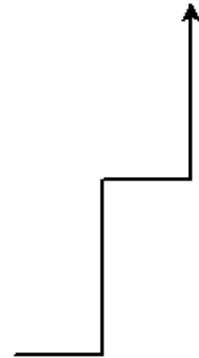
Program Explanation

- Line 1 tells Python to import a *library* called 'turtle'. A library can be thought of as an external Python program that contains useful code. `from` and `import` are two Python keywords. When a library is imported into a program the functionality of that library can then be used in that program.
- The commands on lines 3 to 9 inclusive instruct Python to move the turtle forward and turn it left/right until the shape is drawn.

Students should be reminded to close the turtle window once they have finished running your program. The window can be closed by clicking on  in the top right corner.

Suggested Activities

1. Read lines 3-9 of the program and see if you can figure out how the shape is created
2. Type the program in and run it. (Warning! Do not save the program as `turtle.py`)
3. Insert comment on lines 5 – 9 inclusive. (Lines 3 and 4 are already commented.)
4. Rearrange lines 3 - 9 into different orders and see if you can explain the change in output. You can delete some lines if you wish.
5. Experiment with the numbers used on lines 3 – 9 until you understand what they mean. For example, you could change 100 to 50 on line 3, or change 90 to 45 on line 8.
6. Modify the program so that it displays the shape shown to the right



STUDENT TIP

Students should be encouraged to get into the habit of saving and re-running their program after every little change.

Some of the more common movement commands supported by the `turtle` library are outlined below.

Command	Explanation
<code>forward(n)</code>	This command moves the turtle forward by <code>n</code> units from whatever position the turtle is facing at the time the command is issued
<code>backward(n)</code>	When this command is issued it moves the turtle in the opposite direction to whatever direction the turtle is facing. The turtle is moved by <code>n</code> units from its current position.
<code>right(angle)</code>	This command turns the turtle in a rightwards direction. The amount of turn is specified by the programmer using <code>angle</code> .
<code>left(angle)</code>	This command turns the turtle in a leftwards direction. The amount of turn is specified by the programmer using <code>angle</code> .

Further Activities


The default starting position for the turtle is the centre of the screen with an arrow pointing to the right (i.e. east). It is up to the programmer to keep a track of the position of the turtle on the screen and the direction it is facing.

The best way to learn how to use turtles is to experiment. The following exercise might help.

- Match each code block (numbered below) to the corresponding shape (denoted by letters).

1.


```
forward(100)
right(90)
forward(50)
right(90)
forward(100)
right(90)
forward(50)
```





2.


```
forward(100)
left(60)
forward(100)
left(60)
forward(100)
```

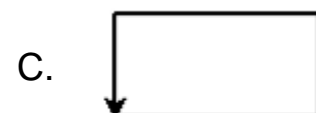




3.


```
forward(100)
left(120)
forward(100)
left(120)
forward(100)
```

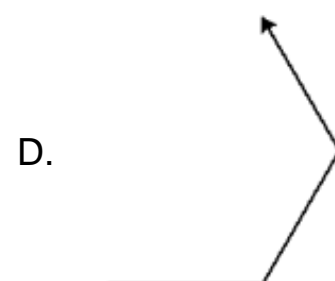




4.

```
forward(100)
left(90)
forward(50)
left(90)
forward(100)
left(90)
forward(50)
```



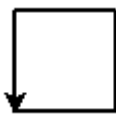


Note: Before running any of the above blocks of code you will need to add the line `from turtle import *` before the turtle commands.

- Now demonstrate that your answers are correct!
Do this by keying in and running each of the separate code blocks.
- The commands listed below can be used to change the appearance of turtle objects

Command	Explanation
<code>shape(s)</code>	This command sets the appearance of the turtle object to be whatever shape is specified by <i>s</i> . Valid values are <code>arrow</code> , <code>turtle</code> , <code>circle</code> , <code>square</code> , <code>triangle</code> and <code>classic</code> . (Use quotation marks.) The arrow shape is the default.
<code>hideturtle()</code>	When this command is used it makes the turtle object disappear from the output screen.
<code>showturtle()</code>	This command makes the turtle visible again.
<code>color(c)</code>	This command sets the colour of the lines drawn by the turtle to be the colour specified by <i>c</i> . Try different values e.g. <code>red</code> , <code>blue</code> , <code>green</code> . (Don't forget to use quotation marks either side of the named colour.)
<code>pensize(n)</code>	This command sets the line thickness of the line drawn by turtle movements. The value of <i>n</i> can be any number from 1 to 10 where 1 is the thinnest and 10 is the thickest. Try it!

Write a Python program to display the shapes shown below.



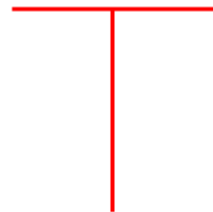
A 50x50 square



A 50x100 red rectangle



A vertical blue line of length 100 units and thickness 5 units



The letter T in red (pen is hidden)

Can you come up with more than one solution for each shape? Compare and discuss your solutions with your classmates.

BREAKOUT 1.2: Games Programming with pygame

*pygame*¹ is a free and open source Python library useful for games programming. As it does not come with the standard Python installation, *pygame* needs to be installed separately.

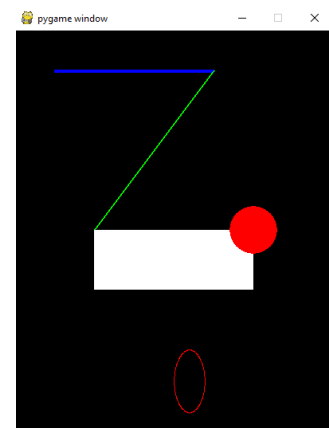
When the program shown below is run it causes the output window illustrated to the right to be displayed. The output window contains 5 different shapes – a blue horizontal line, a green diagonal line, a white rectangle, a red circle and red ellipse. These shapes are drawn in response to the commands on lines 15 to 19.

Read the code carefully – focus your attention on lines 15 to 19 (highlighted in bold) - and see if you can guess which line of code is responsible for drawing which shape.

```

1. import pygame
2.
3. # set up pygame
4. pygame.init()
5. window = pygame.display.set_mode((400, 500)) # (width, height)
6.
7. # set up the colours
8. BLACK = (0, 0, 0)
9. RED = (255, 0, 0)
10. GREEN = (0, 255, 0)
11. BLUE = (0, 0, 255)
12. WHITE = (255, 255, 255)
13.
14. # draw some shapes
15. pygame.draw.line(window, BLUE, (50, 50), (250, 50), 4)
16. pygame.draw.line(window, GREEN, (100, 250), (250, 50), 2)
17. pygame.draw.rect(window, WHITE, (100, 250, 200, 75))
18. pygame.draw.circle(window, RED, (300, 250), 30, 0)
19. pygame.draw.ellipse(window, RED, (200, 400, 40, 80), 1)
20. # update the window display
21. pygame.display.update()

```



Program Listing

Output Window

The individual lines of code are explained on the next page.

Programmers are often presented with code they have not written themselves and need to figure out for themselves what the code does. One tried and trusted method used by programmers to familiarise themselves with 'new' code is to 'play with it' i.e. make small incremental changes to build up an understanding. Try the following suggested activities.

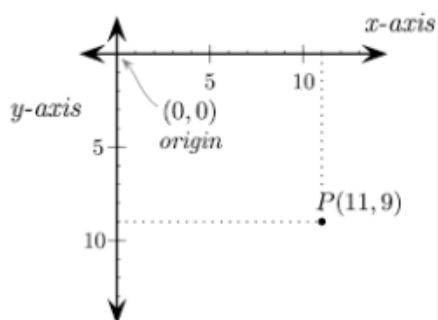
¹ <https://www.pygame.org>

Suggested Activities

1. Key in (or copy+paste from GitHub) the full program and make sure it runs properly.
2. Devise your own theories about the code. For example, you might suspect that line 15 draws the blue horizontal line. In order to test this theory, you could comment out lines 16, 17, 18 and 19 and then run your program to see if you are correct.
This process should be repeated until you have confirmed your understanding of which line of code is responsible for which shape.
3. Experiment by rearranging lines 15-19 into different orders. Each time you jumble them around, run your program to see if they make any difference to the output displayed.
4. Change the code so that the shapes are displayed in different colours
5. Modify the numbers used in the commands used to draw the lines and the rectangle (lines 15, 16 and 17). Can you figure out what the numbers mean?

Co-ordinate System

The diagram below explains the window co-ordinate system used by `pygame`.



window co-ordinate system used by pygame

The co-ordinates of the four corners of a window having width, w and height, h are:

- $(0, 0)$ top left
- $(w, 0)$ top right
- $(0, h)$ bottom left
- (w, h) bottom right

Game Loop

You may have noticed that the output window does not close. To fix this problem you will need to make two changes

- a) Modify line 1 as shown
- b) Add the (game loop) code shown here to the end of the program listing.

```
1. import pygame, sys
```

```
22.
23. # run the game loop
24. while True:
25.     for event in pygame.event.get():
26.         if event.type == 12:
27.             pygame.quit()
28.             sys.exit()
```

Program Explanation

- Line 1 imports the `pygame` and `sys` libraries into the program. `pygame` contains functionality that our program can use to draw shapes.
- Line 4 tells Python to initialise (i.e. start) the `pygame` engine
- Line 5 tells Python to create an output window of width 400 units and height 500 units
- Lines 9 to 12 define the primary colours BLACK, RED, GREEN, BLUE and WHITE. These names are now known to Python and can be used further down in the program.
- Lines 15 instructs Python to draw a horizontal blue line. The co-ordinates of the start and end positions are provided along with codes for the colour and line thickness.
- Lines 16 instructs Python to draw a diagonal green line. The co-ordinates of the start and end positions are provided along with codes for the colour and line thickness.
- Lines 17 instructs Python to draw a white rectangle. The co-ordinates of the upper left corner are provided along with values for the width and height.
- Lines 21 tells Python update the display window with the new shapes.



Log your thoughts.

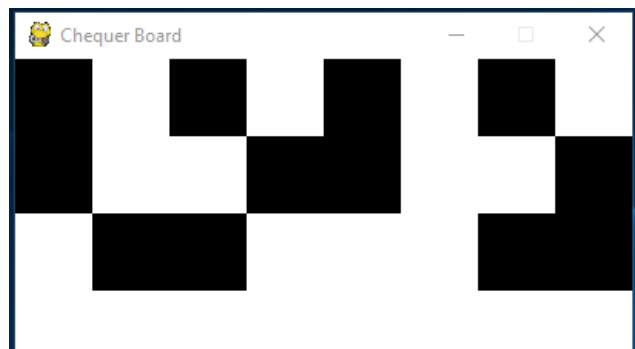
How has your knowledge of programming been extended so far?

Further pygame activities

1. The commands on lines 17, 18 and 19 draw a rectangle, circle and ellipse respectively. Modify the numbers used inside the brackets. Can you figure out what the numbers mean?
2. Study the program listing on the next page carefully.
When the program is run, it displays the first three rows of the pattern as shown.

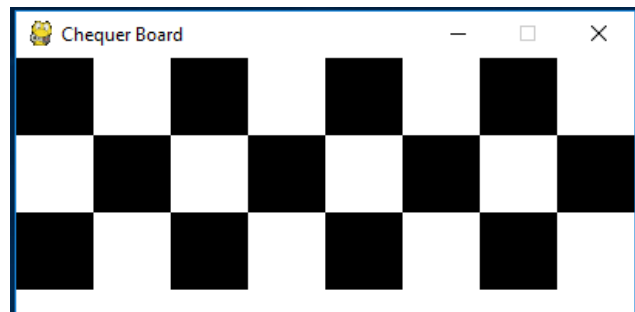
The background is painted white by the `fill` command, so, in actual fact, the program draws four black squares on each of the three rows.

Each individual black square is drawn in response to the command `pygame.draw.rect`. The squares are 50×50 units in size.



There is a problem however.

The programmer had intended the program to display the chequer board pattern shown here to the right.



Can you make the necessary changes?

How might the program differ if the background was painted BLACK instead of white?

3. Calculate the co-ordinates of the centre of a 400x500 window.
Can you generalise this calculation with a formula that would work for a window of any size?
4. Write a program to display a circle centred on the output window. (You choose the size!)
Generalise your solution so that it works for any window size.

Modify the program below so that it displays the first three rows of a proper chequer board pattern.

```
import pygame, sys
from pygame.locals import *

# start the pygame engine
pygame.init()

# create a 400x400 window
window = pygame.display.set_mode((400, 400))
pygame.display.set_caption('Chequer Board')

# define some colors
BLACK = (0, 0, 0)
WHITE = (255, 255, 255)

window.fill(WHITE) # paint the window white

# Draw Row 1
pygame.draw.rect(window, BLACK, (0, 0, 50, 50))
pygame.draw.rect(window, BLACK, (100, 0, 50, 50))
pygame.draw.rect(window, BLACK, (200, 0, 50, 50))
pygame.draw.rect(window, BLACK, (300, 0, 50, 50))
# Draw Row 2
pygame.draw.rect(window, BLACK, (0, 50, 50, 50))
pygame.draw.rect(window, BLACK, (150, 50, 50, 50))
pygame.draw.rect(window, BLACK, (200, 50, 50, 50))
pygame.draw.rect(window, BLACK, (350, 50, 50, 50))
# Draw Row 3
pygame.draw.rect(window, BLACK, (50, 100, 50, 50))
pygame.draw.rect(window, BLACK, (100, 100, 50, 50))
pygame.draw.rect(window, BLACK, (300, 100, 50, 50))
pygame.draw.rect(window, BLACK, (350, 100, 50, 50))

# update the window display
pygame.display.update()

# run the game loop
while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()
```



Section 2

Data, Variables, Assignments and Expressions

Introduction

So let's say we wanted to write a program to display the name of a person and their favourite colour in a greeting string and then display a personalised goodbye message. We could write:

```
print("Hi Alex - your favourite colour is red")  
print("Goodbye Alex")
```



One problem with the above program is that the string *Alex* appears twice and this gives rise to the possibility of a mismatch in spelling. It would be better if had some way of telling our program to remember the person's name. This can be done by using a *variable*.



KEY POINT: A variable is a programming construct used to store (remember) data.

The listing below uses two variables – `personName` and `favouriteColour`.

```
1. personName = "Alex"  
2. favouriteColour = "red"  
3. print("Hi", personName, "- your favourite colour is", favouriteColour)  
4. print("Goodbye", personName)
```



The variable `personName` is used to store a person's name and the variable `favouriteColour` is used to store the person's favourite colour.

The variables are declared on lines 1 and 2 respectively. Each line assigns the initial values *Alex* and *red* to the respective variables.



KEY POINT: A variable must be declared before it can be used. By declaring a variable you are telling Python *here is a new word and this is its initial value*.

Line 3 displays the contents of the variables in a greeting string. Notice that the names of the variables appear outside the double quotations, and also the use of commas to delimit the variables from the greeting string.

When Python comes across the variable names in the `print` command it substitutes the values of the variables into the string to be displayed.

The name of the person or the colour can now be changed, simply by changing the value of the variable. For example, we could write:

```
1. personName = "Charlie"  
2. favouriteColour = "green"  
3. print("Hi", personName, "- your favourite colour is", favouriteColour)  
4. print("Goodbye", personName)
```

The program is considered better because the person's name is stored in a variable and needs to be keyed in by the programmer only once. However, the program still has a problem in that it lacks generality i.e. it only works for one person and one colour. Every time we want to display a different message we need to change the program.

A more general (and realistic) solution would be to ask the user to enter their name and favourite colour. This can be achieved using the `input` command as follows:

```
1. personName = input("Enter your name: ")  
2. favouriteColour = input("Enter your favourite colour: ")  
3. print("Hi", personName, "- your favourite colour is", favouriteColour)  
4. print("Goodbye", personName)
```



Try running the above program for yourself.

The `input` command

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

The string in brackets following the word `input` is displayed as a prompt to the end-user.

Every time the above program is run, whatever values are entered by the end-user are stored in the variables `personName` and `favouriteColour`. These values are then displayed in the output messages.

Without having to make any changes to the program, the output messages can vary on every run. This is an example of a general solution to a problem.

The `input` (and `print`) commands are both examples of Python *built in functions*. The complete list of Python built in functions can be found in the appendix.

Variable Syntax

The general syntax for declaring a variable is made up of a left hand side and a right hand side as follows:

<variable-name> = <expression>

The name of the variable appears on the left hand side and an expression appears on the right hand side. The '=' symbol in the middle is the Python *assignment operator*.



KEY POINT: Although the symbols used to denote the Python assignment operator and a mathematical equation are identical, they should not be confused as they mean two completely different things.

The use of '=' in Python indicates an *assignment statement*. When Python comes across an assignment statement it evaluates the expression on the right hand side first. The result of this evaluation is then stored in the variable named on the left hand side.

The expression on the right hand side can be:

- a literal value such as a string or a number
- an arithmetic expression (which itself can contain variables)
- the name of a built-in command such as `input`, as seen in the previous example.

STUDENT TIP

It is useful to think of a variable as a 'box' in the computer's memory (i.e. a memory location) where a value is stored.

Every time a value is assigned to a variable that value is stored at that variable's memory location.



This is an exercise about vocabulary.
The graphic below shows six Python assignment statements and six (incomplete) English sentences.

Complete the sentences on the right so that each one describes its corresponding assignment statement on the left.

<code>daysLeft = 167</code>	The variable _____ is initialised to 167
<code>rate = 18.27</code>	The value _____ is stored in the variable called <code>rate</code>
<code>name = "Alex"</code>	The value <i>Alex</i> is assigned to the variable, _____
<code>vowels = "AEIOUaeiou"</code>	The English vowels are _____ to the variable _____
<code>pwd = input("Password: ")</code>	The value entered by the user is stored in the variable _____
<code>pay = hoursWorked * rate</code>	The value of _____ is _____ by the value of <code>hoursWorked</code> and the result is stored in the variable _____

Once a variable has been declared the name is added to Python's vocabulary for the remainder of the program.

STUDENT TIP

Variables are used by programs to store data.

A programmer should decide to use a variable whenever they want their program to remember a value that will be needed at a later stage in that program.

When a variable is declared for the first time it must be given some initial value. This is called *initialisation*.

It is up to the programmer to decide what name to give their variables. The rules and guidelines for naming variables are described on the next page.

Guidelines and Rules for Naming Variables

As a general guideline variable names should be simple and meaningful. A meaningful name is one that tells something about what the variable is used for. The use of meaningful variable names makes programs more readable and understandable to fellow programmers.

When choosing a name for a variable it can be helpful to think of a noun that describes the purpose of the variable.

It is considered good practice to capitalise interior words in multi-word variable names. This usage is referred to as *camel case* and `firstName`, `addressLine1`, `stockCount`, `highScore`, and `payRate` are all examples of good variable names.

The syntax rules for naming variables are as follows:

- A variable name cannot be Python keyword (e.g. “import” “def”, etc.)
- Variable names must contain only letters, digits, and the underscore character, `_`.
- Variable names cannot have a digit for the first character.
- Spaces or dots are not allowed in a variable name

If Python comes across a name it does not understand it will display a syntax error.



Which of the following are ‘legal’ variable names?

- a) `student.Number`
- b) `x`
- c) `1x`
- d) `x1`
- e) `input`
- f) `number`
- g) `20`
- h) `h20`
- i) `PPSN`
- j) `ppsn`
- k) `person name`
- l) `address`
- m) `date_of_birth`
- n) `2+4`
- o) `print`

Datatypes and Operations

Programmers need to be aware of the type of data that their programs process. This is referred to as datatype.

Thus far, we have encountered examples of both string and numeric datatypes. If, for example, we wanted a program to store someone's name or favourite colour the variable's datatype would be string. On the other hand, a numeric datatype is the proper datatype for a variable to store a person's age or height.

Python supports several different types of numbers - *integers*, *floating point numbers* as well as a range or more exotic types of numbers (e.g. complex numbers, fixed precision decimals and rational numbers)

Every datatype in Python has a permissible set of operations that are only valid for that type. (For this reason, Python is said to be a *strongly typed* language.) The numeric datatype supports all the usual arithmetic operations such as addition, multiplication etc. These are illustrated in the table below (assume $x=7$ and $y=3$)

Operator	Description	Example	Result
+	Addition	$x + y$	10
-	Subtraction	$x - y$	4
*	Multiplication	$x * y$	21
%	Remainder	$x \% y$	1
/	Division	x / y	2.33333
//	Floor Division	$x // y$	2
**	Power	$x ** y$	343

Python arithmetic operators

The normal precedence rules for arithmetic operators apply.

Reflection

Reflect on what you have learned about variables, datatypes and expressions so far.



Use the space provided to document what extended your thinking about variables, datatypes, and expressions




Indicate in the space below those areas relating to variables, datatypes, and expressions that you don't fully understand yet

Program Tracing

Computers can execute programs at a rate of millions of lines per second. The values of variables can change so fast that programmers can easily lose track and sometimes find it difficult to be sure that their program logic is correct. In order to combat this, programmers often execute a program manually i.e. using pen and paper to keep track of variables line-by-line. This activity, called program *tracing* is used by programmers to verify for themselves that their program will do what it is intended to when it is run by the computer.

We trace through the program shown (as if we were the computer) line-by-line, starting at line 1. Every time a variable is declared for the first time we draw a box and write the value of the variable in the box. When the value of a variable changes we replace the old value with the new one. This activity is called program tracing.

```
1. x = 8
2. y = x
3. print(y)
4. x = x + 1
5. print(x)
6. print(y)
```



The Python code is shown on the left below and the variables are illustrated as boxes on the right. The boxes are used to represent memory locations i.e. they are part of a *notional machine* used by programmers to keep track of the state of their variables at runtime.

```
1. x = 8
```



```
2. y = x
```



```
3. print(y)
```

The program displays the contents of *y* i.e. 8

```
4. x = x + 1
```



```
5. print(x)
```

The program displays 9

```
6. print(y)
```

The program displays 8




Exercise: Program Tracing

Manually trace the programs shown below. (Use the space provided on the next page.) Can you figure out what each program does?


PROGRAM 1

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




PROGRAM 2

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




PROGRAM 3

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




PROGRAM 4

```
1. accountBalance = 1000
2. withdrawalAmount = 600
3. accountBalance = accountBalance - withdrawalAmount
```



PROGRAM 5

```
1. days = 2
2. hrs = 24
3. mins = 60
4. total = days*hrs*mins
5. print(total)
```

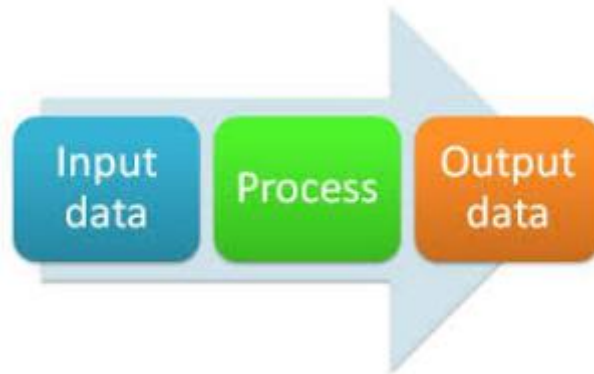


STUDENT TIP

Students should be encouraged to trace existing code as early as possible in the learning process.

Input-Process-Output

Many computer programs follow the input-process-output model as illustrated.



This means that a program accepts data as input, carries out some processing (usually a calculation) and then displays and/or stores the output.

We already know the `input` command is used to prompt an end-user to enter a value into a running program. The value entered can then be stored in a variable.



KEY POINT: programmers need to be acutely aware of the type of data with which their program is working.

By default, the `input` command returns a string. This means that if you want your program to accept numeric data from the end-user, the value entered must be converted from a string to either an integer or a floating point (i.e. a decimal) number.

Fortunately, Python has two built-in commands that can perform these conversions. These are called `int` and `float` respectively.

Built-in Function	Description
<code>int(s)</code>	Converts the string 's' to an integer. The result is a new number object
<code>float(s)</code>	Converts the string 's' to a floating point (decimal) number. The result is a new floating point object

The two commands `int` and `float` are important because they allow Python to use values entered by the end-user in arithmetic expressions.

Example – Year of Birth

We want to write a program to calculate a person's year of birth.


Our program will ask the end-user for two pieces of information - the current year and the age they will be at the end of the current year.

We will store this data in two variables – `year` and `age`.

Since `year` and `age` are both numeric we will need to instruct the program to convert them from strings to integers. This can be done with the `int` command.

The solution is as follows.

```
1. year = int(input("Enter the current year: "))
2. age = int(input("What age will you be at the end of this year : "))
3. print("You were born in", year-age)
```



Lines 1 and 2 both display a prompt asking the user to enter values and then convert these values from strings to integers. The conversion from string to integer is needed here because Python knows how to subtract numbers but cannot subtract strings.

Line 3 subtracts the two integers (to calculate the year of birth) and displays the result in an output message.

Notice how both `int` and `input` are called on the same line. When commands are combined together on the same line like this it is called *function composition*. Python executes the innermost function first and then works back towards the leftmost function which is executed last.

It is important to understand the subtle difference between the string “2018” and the numeric value 2018. One difference is subtraction is supported for numbers but not for strings



KEY POINT: The operations that can be carried out on values are constrained by the value's underlying object type. For example, subtraction is not supported for strings.

Syntax Check: Try running the following program:

```
1. year = input("Enter the current year: ")
2. age = input("What age will you be at the end of this year : ")
3. print("You were born in", year-age)
```

You will see an error like this:

```
Traceback (most recent call last):
  File "C:\PDST\Python Workshop\src\year of birth.py", line 3, in <module>
    print("You were born in", year-age)
TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

Since the values are not converted to integers Python stores them as strings. The expression `year-age` on line 3 is an attempt to subtract two strings which is not allowed in Python. Python does not support the subtraction operation on strings.

It is also worth noting that numbers can be converted to string objects using the `str` function. As an experiment try running the following lines of code separately:

```
print("2000"+18)
```

This causes the following error to be displayed
`TypeError: must be str, not int`

```
print("2000"+str(18))
```

This causes the string “200018” to be displayed.



KEY POINT: The '+' operator for strings means concatenation.

Example – Temperature Conversion

Let's look at another example of a program that accepts an input, performs some processing and displays an output. The program below converts Centigrade to Fahrenheit. The formula used is,

$$f = \frac{9}{5} \times c + 32$$

The input is the Centigrade value entered by the end-user; the processing is the conversion (done by applying the formula); and the output is the Fahrenheit value displayed.

```
1. # A program to convert from Centigrade to Fahrenheit
2. centigrade = float(input("Enter the Centigrade value: "))
3. fahrenheit = 9/5 * centigrade + 32
4. print(centigrade, "degrees C equals", fahrenheit, "degrees F")
```



The use of the `float` command on line 2 above means that the user can enter decimal values for Centigrade. (Integer values are also permitted.)

Notice how both `float` and `input` are combined together on the same line.



Use the space provided to write down what you think would happen if the `float` command was removed from the above function. Why do you think `int` was not used?

A note on Testing

The purpose of testing is to verify that the program does what it is meant to.

It is normal practice for programmers to devise a number of *test cases* as part of the program design process. Each test case is made up of an input and an *expected output*.

When the test case is run the *actual output* should be recorded. If there is a difference between the expected and actual output, then the program contains an error (or bug) which will need to be fixed.

The table below provides a good basis for testing the Centigrade to Fahrenheit program shown on the previous page. Each row in the table is a separate test case.

Sample Input (°C)	Expected Output (°F)	Actual Output
0	32.0	
32	89.6	
1000	1832.0	
-10	14.0	
-40	-40.0	
-1000	-1768.0	

Note that the values in the sample input column are arbitrarily chosen. The expected output values are calculated by using a calculator or come from some other source e.g. world wide web. The values in the actual output column should be recorded by running the program.

When the program passes all test cases it is said to be *unit tested*.

A good unit test will ensure that every line of code is triggered. It will also take 'abnormal' scenarios into consideration

STUDENT TIP

Students should be encouraged to get into the habit of testing their code as early as possible in the learning process.

Reflection

Examine the program below in terms of input(s), processing and output(s).

```
1. # Ask the user for the principal
2. principal = input("Enter principal: ")
3. principal = float(principal)
4.
5. # Ask the user for the interest rate
6. rate = input("Enter rate: ")
7. rate = float(rate)
8.
9. # Ask the user for the length of time
10. time = input("Enter time in years: ")
11. time = float(time)
12.
13. # Simple interest calculation
14. amount = principal * rate * time
15.
16. # Display the answer
17. print("The interest amount is", amount)
```



Use the space below to record your reflection.

Questions to consider might include:

1. *What are the inputs? What processing is done? What is the output?*

2. *What would happen if all or any of lines 3, 7 and 11 were removed from the program?*

3. *How could the code be made more terse?*



Write an additional line of Python code to calculate the new principal.

More built-in functions

Let's take another look at our temperature conversion program discussed earlier.

```
1. # A program to convert from Centigrade to Fahrenheit
2. centigrade = float(input("Enter the Centigrade value: "))
3. fahrenheit = 9/5 * centigrade + 32
4. print(centigrade, "degrees C equals", fahrenheit, "degrees F")
```



This time try an input value of 21°C. The program displays the output shown below – the number of digits displayed after the decimal point is unnecessary and unwieldy.

21.0 degrees C equals 69.80000000000001 degrees F

The level of precision displayed can be controlled by the programmer by using the `round` function. Line 4 tells Python to display the value of `fahrenheit` *rounded* to 1 decimal place.

```
4. print(centigrade, "degrees C equals", round(fahrenheit,1), "degrees F")
```



A full description of the `round` function is given in the table below along with some other useful build-in functions offered by Python.

Function	Description	Example(s)
<code>round(x [,n])</code>	Rounds the number <code>x</code> to <code>n</code> fractional digits from the decimal point. If <code>n</code> is not provided it is taken to be zero.	<pre>round(27.168459, 1) -> 27.2 round(27.168459, 2) -> 27.17 round(27.168459, 3) -> 27.168 round(27.168459, 4) -> 27.1684 round(27.168459, 5) -> 27.16846 round(27.168459, 6) -> 27.168459</pre>
<code>abs(x)</code>	Returns the absolute value of <code>x</code>	<pre>abs(-273) -> 273 abs(-1.27) -> 1.27 abs(0) -> 0 abs(32) -> 32</pre>
<code>pow(x, y)</code>	Calculates <code>x</code> to the power of <code>y</code> . (Same as <code>x**y</code>)	<pre>pow(5, 2) -> 25 pow(2, 16) -> 65536 pow(4, 3) -> 64</pre>



Can you predict what output would be displayed by this line of code?

```
print(pow(10, abs(-2)))
```

The Remainder Operator (%)

The remainder operator, % (aka modulus operator), like all of the other Python arithmetic operators, is a binary operator i.e. it works on two numbers, referred to as operands.

The comments (in red) in the short Python program below describe how examples of the remainder operator works.

```

print(30 % 10) # displays 0 because 30 divided by 10 leaves no remainder
print(30 % 15) # displays 0 because 30 divided by 15 leaves no remainder
print(30 % 20) # displays 10 because 30 divided by 20 leaves a remainder of 10
print(9 % 4)   # displays 1 because 9 divided by 4 leaves a remainder of 1
print(9 % 5)   # displays 4 because 9 divided by 5 leaves a remainder of 4
print(12 % 5)  # displays 2 because 12 divided by 5 leaves a remainder of 2
print(5 % 12)  # displays 5 because 5 divided by 12 leaves a remainder of 5
  
```

It should be evident that the remainder operator works by dividing the second operand into the first. Whatever is left over is the result of applying the remainder operator.

The word *mod* (short for modulus) is often used to phrase questions or answers involving the remainder operator. For example, what is 30 mod 10, or 9 mod 5 is equal to 1. Furthermore,

- *a mod a is zero* because any number divided by itself leaves no remainder
- *a mod 1 is zero* because 1 divides every number evenly
- *a mod 0 is undefined* because division by zero is not defined. In Python any attempt to divide by zero will always result in a runtime error.

Computer Science contains a rich set of problems whose solutions involve using the remainder operator. The application of the remainder operator to solve problems is referred to as modular arithmetic and because modular arithmetic is particularly useful for calculations involving time, it is also referred to as *clock arithmetic*.

For example, let's say we wanted to find out what day of the week it will be in 1000 days' time. We know every week has 7 days, and can calculate 1000 mod 7 to be 6. Therefore, the day of the week in 1000 days will be the same day as it will be 6 days from now.

The same general principal can be applied to working solutions to problems involving other units of time such as seconds, minutes, hours, months, years, leap years, Easter etc.

Let's take another example. You are about to embark on a space journey lasting 850,000 hours! Take off time is exactly 21:00h. What time will it be when you reach your destination?

```
arrivalTime = (21 + 850000) % 24  
print("You will arrive at", arrivalTime)
```

Modular arithmetic is also useful in situations where it is required to group 'things' (e.g. people) into a fixed number of arbitrary sized groups. For example, if we had a group of 20 students in a class and we wanted to create four arbitrary sized groups, we could ask each student to pick a random number – let's call it N . Then $N \bmod 4$ will guarantee a group number for that student because it will always be 0, 1, 2 or 3.

Another useful application is simply to find the properties of numbers. For example, $\bmod 2$ is frequently used in computer programs to test whether a number is even or not (*evenness test*).

Perhaps, some of the most common applications of modular arithmetic can be found in the area of coding systems, ciphers and cryptography. Ubiquitous examples exist in the use of modular arithmetic to perform validity checks on barcodes, ISBN numbers and credit card numbers (e.g. Luhn's algorithm) to name just a few. For example, a 13-digit barcode is valid only if the following expression is evenly divisible by 10.

$$(d1 + d3 + \dots + d13) + 3 \times (d2 + d4 + \dots + d12)$$

where, $d1$ is the leftmost and $d13$ the rightmost barcode digit. If the expression $\bmod 10$ is not equal to zero, the barcode is invalid and the scanned item will be rejected.



Programming Exercises

- Write a program to calculate and display the total from the bill below.

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

Hint: Just re-arrange the lines of code shown here.

```

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

- Re-arrange the jumbled up lines shown below so that the program displays the sum of two integers entered by the end-user.

Warning! There are *three* extra lines that you won't need.

```

    number2 = int(number2)

number1 = int(input("Enter first number: "))

    sum = sum + number1

    number1 = int(number1)

print(number1, "+", number2, "=", sum)

number2 = input("Enter second number: ")

print("The answer is sum")

    sum = number1 + number2
  
```

- Given the following formula to convert Fahrenheit (f) to Centigrade (c) write a program to prompt a user to read in °F and display its °C equivalent correct to 2 decimal places.

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

4. Write a program to find the day of the week for any date using Zeller's Algorithm.

In Zeller's algorithm the year is assumed to begin in March.

Months are numbered as: 3 for March, 4 for April, ... 13 for January and 14 for February.

January 1997 is treated as month 13 of 1996.

The formula is as follows:

$$w = \left(dd + \left\lfloor \frac{13(mm + 1)}{5} \right\rfloor + y + \left\lfloor \frac{y}{4} \right\rfloor + \left\lfloor \frac{c}{4} \right\rfloor - 2c \right) \bmod 7$$

where,

w is the calculated weekday. (*Sat* = 0, *Sun* = 1, *Mon* = 2, ... *Fri* = 6)

dd is the day of the month (e.g. for 25/12/1989, **dd** is 25)

mm is the shifted month, *Mar* = 3, *Apr* = 4, ..., *Jan* = 13, *Feb* = 14.

(e.g. for 25/12/1989 **mm** is 12)

y is the last 2 digits of the year - remember the year is minus 1 for January or February and the actual year for any other month. (e.g. for 25/12/1989 **y** is 89)

c is the first 2 digits of the year (e.g. for 25/12/1989 **y** is 19)

Hint: You need to decide what inputs you need from the end-user and what inputs to the formulae you can derive from these user inputs.

Test your program using the following data:

Date	Expected Output	Actual Output
25/12/1989	2	
01/01/2000	0	
22/11/1963	6	
Your Birthday!	?	

Running Totals

Running totals are needed so often in programs that it is well worth putting some effort into understanding the pattern used to create them.

A running total is a value that usually starts at zero and increases by successive additions until a final total is reached. A very common example is a shopping basket total calculated at a checkout.

Let's say we have three items in our basket and they are valued at €10, €14 and €6 respectively. With very little effort, most people understand that the total bill is €30. However, what most people probably don't realise is that they have (subconsciously) run a running total program similar to that shown below in their own heads.

```
1. # Program to calculate a running total
2.
3. # Initialise the variable
4. runningTotal = 0
5.
6. # Perform the calculations
7. price1 = 10
8. runningTotal = runningTotal + price1
9. price2 = 14
10. runningTotal = runningTotal + price2
11. price3 = 6
12. runningTotal = runningTotal + price3
13.
14. # Display the output
15. print("Total amount is", runningTotal)
```



Novice programmers may find the following tips for dealing with running totals useful:

1. Recognise the need for a running total (this is probably the most difficult step)
2. Initialise your running total variable to zero (line 4 above)
3. Every time you need to add a value to the running total use an assignment (lines 8, 10 and 12 above). Notice the only difference between these lines is the name of the variable being added to `runningTotal`.



KEY POINT: The hallmark of the running total pattern is the `runningTotal` variable appears on both sides of the assignment statement. In this way it is used to update itself.



Can you re-order the lines in the previous listing without breaking the code?



What one question do you still have in relation to running totals?

Introducing Random Numbers

Random numbers provide a rich way of generating numeric data in early stage programming. A more advanced application of random number generation is in games programming.


The following two programs demonstrate random number generation.

The first multiplies two randomly generated numbers and displays the result; the second computes the mean of five randomly generated numbers.

```

1. # Program to multiply two randomly generated numbers
2. import random
3.
4. num1 = random.randint(1,10) # generate a number between 1 and 10
5. num2 = random.randint(1,10) # generate a number between 1 and 10
6.
7. # Multiply the two numbers and display the result
8. print(num1, "times", num2, "=", num1*num2)


```



```

1. # Program to average five randomly generated numbers
2. import random
3.
4. low = random.randint(1,100)
5. high = random.randint(low,100)
6.
7. # Generate the 5 random numbers between low and high
8. n1 = random.randint(low, high)
9. n2 = random.randint(low, high)
10. n3 = random.randint(low, high)
11. n4 = random.randint(low, high)
12. n5 = random.randint(low, high)
13.
14. # Compute their average
15. average = (n1+n2+n3+n4+n5)/5
16.
17. # Add the five numbers and display the result
18. print("The average of", n1, n2, n3, n4, n5, "is", average)

```



Study both programs carefully – paying particular attention to the comments in red - and then answer the questions on the next page *in relation to the second program listing*.



Explain the purpose of the variables `low` and `high`.



Why do you think the variable `low` is used on line 5?



Explain why the brackets are necessary on line 15



Can you recognise how the running total pattern could be used in this program?

Additional Notes

1. Variables and Memory

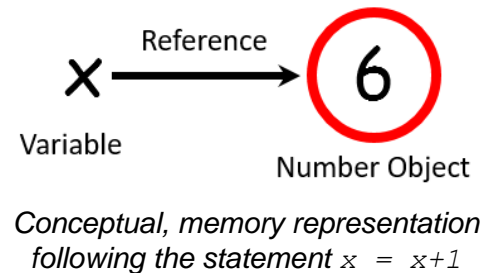
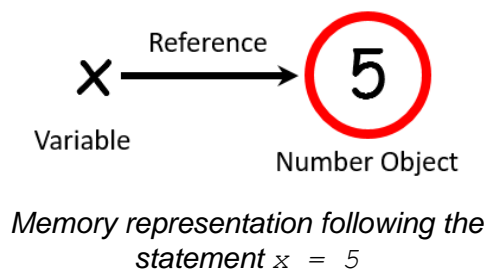
When a variable is assigned a value for the very first time it is said to be *initialised*. We say a variable is initialised to a value. Memory for the variable is *allocated* at runtime when the variable is initialised.

Internally, Python maintains a *system table* with one entry per variable. The values are represented in memory as objects and a reference links the variable to the object.

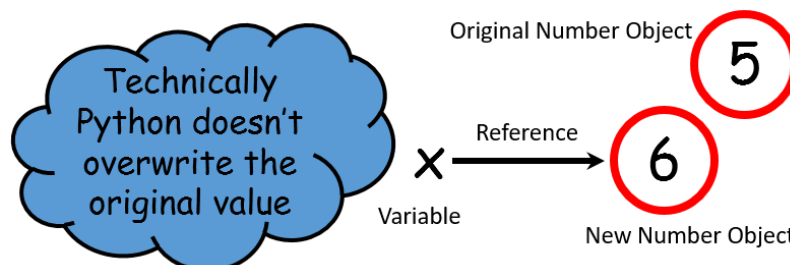


KEY POINT: In Python variables are actually references to objects.

The diagrams below illustrate what happens when we initialise the variable x to 5 and then assign a new value to it.



Conceptually, the value of a variable changes. The old value seems to be replaced with a new value. Technically, what happens is illustrated as follows:



The old value remains on in memory, and the variable that referenced it before the assignment now references the new value.

It is for this reason that objects such as strings and numbers are said to be *immutable*.

Finally, note that the original number object i.e. 5 in this case is left without a reference. Unreferenced values are referred to as *dangling objects*. Such objects are automatically returned by Python, to memory, in a process known as *garbage collection*.)

2. Python Assignment Operators

In addition to the simple assignment operator introduced and used throughout this section, Python supports a variety of other more compact assignment operators.

These are given here for the sake of completeness. (Assume $x=7$ and $y=3$)

Operator	Name	Example	Same as	Result (x)
=	Simple Assignment	$x = y$	N/A	10
+=	Increment Assignment	$x += y$	$x = x + y$	10
-=	Decrement Assignment	$x -= y$	$x = x - y$	4
*=	Multiplication Assignment	$x *= y$	$x = x * y$	21
%=	Remainder Assignment	$x %= y$	$x = x \% y$	1
/=	Division Assignment	$x /= y$	$x = x / y$	2.33333
//=	Floor Division Assignment	$x //= y$	$x = x // y$	2
**=	Power Assignment	$x **= y$	$x = x ** y$	343

Python Assignment Operators

Unlike C++, Java and some other programming languages, Python does not provide built-in support for the unary increment and decrement operators (e.g. $--x$ and $y++$).

STUDENT TIP

It is useful for teachers to be aware that novice programmers commonly confuse the use of the `print` command with an assignment statement.



BREAKOUT ACTIVITIES

BREAKOUT 2.1: Turtle Graphics

Variables, assignments and operations can be used to dynamically vary the dimensions of turtle shapes.

1. Predict what pattern would be generated by the each of the following program listings.

```
from turtle import *

hideturtle() # hide the turtle
color("red") # set the pen colour to red

lineLength = 50
forward(lineLength)
left(90)
lineLength = lineLength + 50
forward(lineLength)
left(90)
lineLength = lineLength + 50
forward(lineLength)
left(90)
lineLength = lineLength + 50
forward(lineLength)
```



Program Listing 1

```
from turtle import *

hideturtle() # hide the turtle
color("red") # set the pen colour to red

lineLength = 50
forward(lineLength)
left(90)
forward(lineLength + 50)
left(90)
forward(lineLength + 50)
left(90)
forward(lineLength + 50)
```



Program Listing 2




Explain the main difference between the two program listings shown above

2. Match each code block (numbered below) with the correct shape (denoted by letters and drawn to scale).

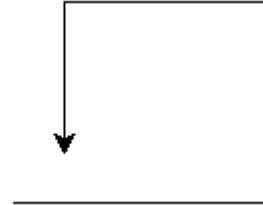
1.

```

x=50
y=25
forward(2*x+y)
left(90)
forward(x+2*y)
left(90)
forward(x+2*y)
left(90)
forward(2*x+y)
  
```




A.



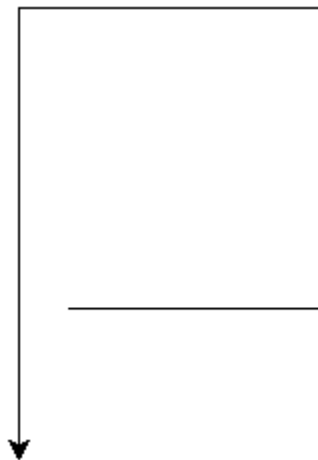
2.

```

x=50
y=25
forward(2*x+y)
left(90)
y=2*y
forward(x+y)
left(90)
forward(x+2*y)
left(90)
forward(2*x+y)
  
```




B.



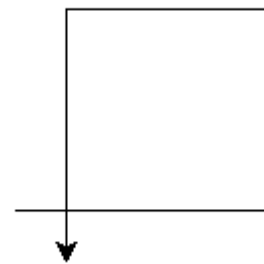
3.

```

x=50
y=25
forward(2*x+y)
left(90)
x=2*x
forward(x+2*y)
left(90)
forward(x+2*y)
left(90)
forward(2*x+y)
  
```




C.



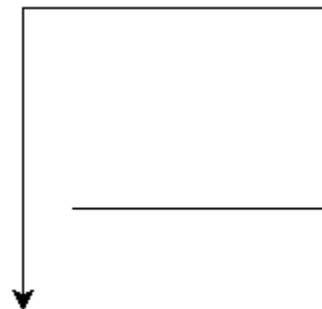
4.

```

x=50
y=25
forward(2*x+y)
left(90)
forward(x+2*y)
left(90)
forward(x+2*y)
left(90)
x=2*y
forward(x+y)
  
```



D.

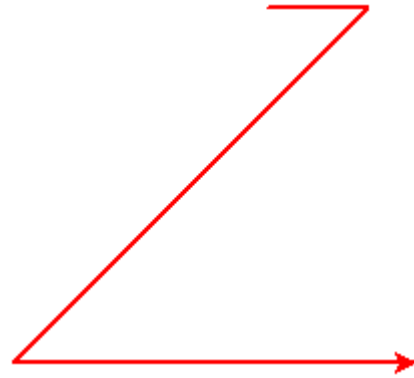


3. Modify the program below so that the angle size is increased by 135° before each turn. The modified program should display the pattern illustrated to the right

```
from turtle import *

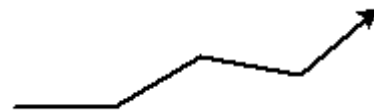
pensize(2) # set the line thickness to 2
color("red") # set the pen colour to red

angle = 90
lineLength = 50
forward(lineLength)
left(angle)
lineLength = lineLength + 50
forward(lineLength)
left(angle)
lineLength = lineLength + 50
forward(lineLength)
left(angle)
lineLength = lineLength + 50
forward(lineLength)
```



4. Implement the following pseudo-code in Python to display the pattern illustrated

Initialise a variable called `angle` to 30
Move forward by 50 units
Turn left by `angle` degrees
Increase the angle by 10°
Move forward by 50 units
Turn right by `angle` degrees
Increase the angle by 10°
Move forward by 50 units
Turn left by `angle` degrees
Increase the angle by 10°
Move forward by 50 units



5. Write a Python program to display five lines of random length. Each line should be joined to the next at a randomly sized angle.

BREAKOUT 2.2: Automated Teller Machine (ATM) Menu System

Recall from Section 1 the menu system for our fictional LCCS bank.

Use the knowledge you have gained so far to convert the pseudo-code shown to the right of the menu below into Python.

```

LCCS BANK LIMITED
ATM Main Menu

1. Balance Enquiry
2. Cash Lodgement
3. Cash Withdrawal
4. Cash Transfer
5. Change PIN
6. Other Services

7. Exit

CHOOSE AN OPTION >>
    
```

Display a welcome message

Initialise a variable called `balance` to 123.45

Display the value of `balance`

Prompt the user to enter an amount to lodge

Increase the `balance` by the amount entered

Display the value of `balance`

Prompt the user to enter an amount to withdraw

Decrease the `balance` by the amount entered

Display the value of `balance`

Hint: You will need to consider what variables you will need as well as their datatype.



Log your thoughts.

What did you find most/least challenging about this task?


BREAKOUT 2.3: Data Processing (average height)

The short program shown below was designed to calculate the mean height from five values entered by an end-user. Study the program carefully until you are satisfied you understand how it works.

```

1. # A program to calculate the average height of 5 people
2. print("Average height calculator")
3. print("=====")
4.
5. # Read in the 5 values
6. h1 = int(input("Enter first height (cm): "))
7. h2 = int(input("Enter second height (cm): "))
8. h3 = int(input("Enter third height (cm): "))
9. h4 = int(input("Enter fourth height (cm): "))
10. h5 = int(input("Enter fifth height (cm): "))
11.
12. # Calculate the average height
13. avgHeight = h1+h2+h3+h4+h3/5
14.
15. # Display the result
16. print("The average height is ", avgHeight, "cm")

```



The test data shown below threw up some differences between expected and actual outputs. Although the program is syntactically correct it contains at least one semantic error (bug).

Input Values (cm)					Expected Output	Actual Output
h1	h2	h3	h4	h5		
150	160	170	180	190	170.0 cm	694.0 cm
190	172	172	178	187	179.8 cm	746.4 cm
171	175	169	182	178	175.0 cm	730.8 cm



Log your thoughts.
What is your opinion of the above program?
In what way(s) could the program be enhanced?

Suggested Activities

1. Key in (or copy+paste from GitHub) the full program and make sure it runs without any syntax errors.
2. Experiment by rearranging lines 6-10 into different orders. Each time you make a change, run your program to see if they make any difference to the output display.
3. Fix the bug(s)
4. Modify the code so that it can accept decimal values for height as well as whole numbers. (Make sure to round your result.)
5. Implement some of your suggestions from the bottom of the previous page
6. Rearrange the lines below into a program that does the same thing.
Note, only five of the lines (excluding comments) are needed but two of these will be needed more than once.

```

        # Calculate the average
h1 = int(input("Enter first height (cm): "))

        # Display the result
height = input("Enter height (cm): ")
avgHeigth = totalHeight/5
totalHeight = 0
print("-----")
height = float(input("Enter height (cm): "))

# A program to calculate the average height of 5 people
print("The average height is "+str(round(avgHeigth,2))+ "cm")
totalHeight = totalHeight + height
print("Average height calculator")
print("The average height is", round(avgHeigth,2), "cm")

        # Read in the 5 values
avgHeigth = (h1+h2+h3+h4+h5)/5

```

7. Add a line of code to display the result in feet and inches as well as centimetres.
(1cm = 0.393701 inches)

Further Activities

By this stage you are more than likely getting tired of having to enter 5 different values for height every time you run your program.

Wouldn't it be nice if you could enter the values into a spreadsheet and every time you run your program it would read the file and calculate the mean based on the 5 values contained in the file?

This is exactly what the following program does.

	A
1	150
2	160
3	170
4	180
5	190
6	



```

1. # A program to calculate the average height of 5 people
2. # The heights are stored in a file called 'heights.csv'
3.
4. heightFile = open("heights.csv","r") # Open the file
5.
6. totalHeight = 0 # Initialise a running total for all the heights to zero
7.
8. height = float(heightFile.readline()) # read the first value
9. totalHeight = totalHeight + height # keep a running total
10.
11. height = float(heightFile.readline()) #
12. totalHeight = totalHeight + height #
13.
14. height = float(heightFile.readline()) #
15. totalHeight = _____ + height #
16.
17. height = float(heightFile.readline()) #
18. totalHeight = totalHeight + _____ #
19.
20. height = float(_____) #
21. totalHeight = totalHeight + height #
22.
23. # Calculate the average
24. avgHeight = _____
25.
26. # Display the result
27. print("The average height is "+str(round(avgHeight,2))+ "cm")
28. print("The average height is", round(avgHeight*0.393701,2), "inches")
29.
30. heightFile.close()

```



Before reading the detailed explanation of the program on the next page see if you can complete the comments on lines 11, 12, 14, 15, 17, 18, 20 and 21.

The file `heights.csv` was created in a spreadsheet and saved into the same folder as the Python source program. This is called the *runtime folder*.

Each of the five lines in the data file contain a numeric value that represents a person's height in centimetres.

Program Explanation

- Line 4 of the program opens a data file called `heights.csv` in read mode. This tells Python that the file will be used for reading (as opposed to writing) purposes. The variable `heightFile` is the program's reference to this file. This is called the *file pointer* (also referred to as a *file handle*). Any operations on the file such as reading the file must use this file handle.

It is useful to think of a file pointer as an imaginary index finger. When a data file is initially opened in read mode the pointer is positioned at the start of that file.

- Line 6 initialises a variable called `totalHeight` to zero. This variable will be used to store the sum of all the height values.
- Line 8 tells Python to read a line from the data file, convert the result to a floating point number and store the resulting value in a variable called `height`. (There's lots going on here so make sure you understand this line as it is a very common type of pattern in Python programming.)

Every time the program reads a line from the file, the file pointer is moved to the start of the next line in the data file. This is a subtle side effect of the `readline` command.

- Line 9 adds the current value of `height` to the value stored in the variable `totalHeight` (which was initialised to zero on line 6). In this case, the answer will be the same as the value in `height`. This answer is then stored in `totalHeight`.
- This pattern of reading the next line from the data file and adding the height value to the running total is repeated four times i.e. once for each line in the data file.
- Line 24 computes the average and lines 27 and 28 display the result.



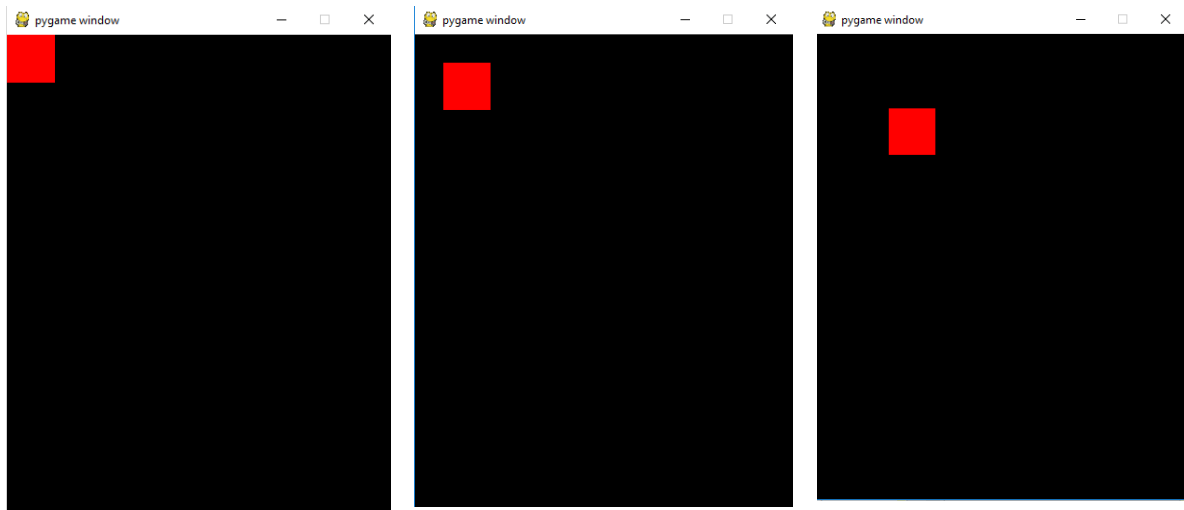
What one question would I still like to ask in relation to this example?

BREAKOUT 2.4: Games Programming with pygame (Animation)

In this activity we will use our knowledge of variables to create the illusion of a 50x50 unit square block moving from a starting position on the top left hand corner of a window towards (and beyond) the bottom right hand corner of the same window.

Before writing any code it is good practice to first design a solution. Start by analysing the problem – make sure you understand what is required. It is always helpful to try to visualise what the final running program will look like.

The three screenshots below depict a red block moving diagonally (downwards and to the right) towards the bottom right corner.



Before writing any code it is often helpful to ask probing questions such as

Have I done something similar before? Can the task be decomposed (broken down)?

What Python commands are there to meet my need? How do these command work?

Do I need variables?

Can I find a solution for a similar problem using the internet? What would I be looking for?

STUDENT TIP

Students should be encouraged to design solutions (with pen and paper) before writing any code.

In this case we will decompose the problem into 2 parts:

- a) Display a 50x50 block in the top left hand corner of a pygame window
- b) Create the required illusion of movement from top left towards bottom right

The solution to part a) along with the output window is presented in the program listing below. (You should key in/download the program and make sure it runs without any syntax errors.)

```

1. import pygame, sys
2.
3. # set up pygame
4. pygame.init()
5.
6. window = pygame.display.set_mode((400, 500))
7.
8. # set up the colours
9. BLACK = (0, 0, 0)
10. RED = (255, 0, 0)
11. GREEN = (0, 255, 0)
12. BLUE = (0, 0, 255)
13. WHITE = (255, 255, 255)
13.
15. # draw a 50x50 square at (0,0)
16. pygame.draw.rect(window, RED, (0, 0, 50, 50))
17.
18. # update the window display
19. pygame.display.update()
20.
21. # run the game loop
22. while True:
23.     for event in pygame.event.get():
24.         if event.type == 12:
25.             pygame.quit()
26.             sys.exit()

```



Program Listing

Output Window

Program Explanation

- It should be possible to understand most of the code from the activity at the end of section 1 (if not you should revisit section 1)
- Line 16 is the key line – this line uses the command `pygame.draw.rect` to draw the 50x50 rectangle (i.e. a square) at position (0,0) in the display window. The ‘clever’ bit is recognising that by using (0, 0) as the co-ordinates the block will be positioned at the top left of the display window as required.
- Line 18 updates the display causing the red square to actually appear in the display window.

The solution to part b) of the problem - presented below - is slightly more intricate and requires a little more in depth knowledge of the 'game loop'.

```

1. import pygame, sys, time
2.
3. # set up pygame
4. pygame.init()
5.
6. window = pygame.display.set_mode((400, 500))
7.
8. # set up the colours
9. BLACK = (0, 0, 0)
10. RED = (255, 0, 0)
11. WHITE = (255, 255, 255)
12.
13. # draw a 50x50 red square
14. x = 0 # x-value of top left co-ordinate
15. y = 0 # y-value of top left co-ordinate
16. pygame.draw.rect(window, RED, (x, y, 50, 50))
17.
18. # update the window display
19. pygame.display.update()
20.
21. # run the game loop
22. while True:
23.     for event in pygame.event.get():
24.         if event.type == 12:
25.             pygame.quit()
26.             sys.exit()
27.
28.     x = x + 5 # add 5 to the top position
29.     y = y + 5 # add 5 to the left position
30.     pygame.draw.rect(window, RED, (x, y, 50, 50))
31.
32.     pygame.display.update() # show the block
33.     window.fill(BLACK) # paint the entire window black
34.     time.sleep(0.02) # pause for 2 milliseconds

```



The 'trick' here is to create the illusion movement by displaying the block in a sequence of different positions. In order to do this the co-ordinates of the block will be needed.

We introduce two variables x and y to store the co-ordinates of the top left corner of the block. The variables are initialised to 0 on lines 14 and 15 respectively.

Our illusion is achieved by changing the value of each variable inside the game loop (lines 28 and 29) before redrawing the block and updating the display.

Line 28 increases the x position by 5 – this causes the horizontal movement.

Line 29 increases the y position by 5 – this causes the vertical movement.



Experiment! Try making the following changes.

1. Comment out line 33. What happens? What is the purpose of this line of code?

2. Change the fill colour on line 33 from BLACK to RED. Explain what happens.

3. Comment out line 28. What happens? Can you explain why?

4. Change the offset amount from 5 to 10 on line 29. Explain any changes in the way the program behaves.

Further Activities

1. Adapt your program so that the block moves as follows:
 - from the top left corner to the bottom left corner
 - from the top left corner to the top right corner
 - from a different starting position in one dimension e.g. top right to bottom right
 - from a different starting position in two dimensions e.g. bottom left to top right
 - from a random starting position to a random ending position
2. Write a program to make a ball (circle) appear to move through the screen.

Section 3

Strings

Introduction

In Section 1 we defined a string as any text enclosed inside quotation marks. Strings are important simply because, along with numbers, they are by far the most common type of data processed by computer programs and systems.

The value of a string can either be a *string literal* or any Python expression that results in a string. Some examples of simple string literals are listed below.

"Please enter your name: "

"John Doe"

"+353-85-1234567"

"182 C 999"

"@PDSTcs Python CPD for #LCCS teachers"

"http://www.youtube.com/watch?v=hUkjib"



KEY POINT

The quotation marks are not part of the string

The individual symbols that make up a string are called *characters*.

Notice from the above examples that characters can be letters, numbers, spaces, punctuation marks and basically any symbol your keyboard will allow you to enter.



KEY POINT: A *string* is a sequence of characters enclosed by quotation marks. Sometimes a string is referred to as an array (or list) of characters.

It is useful to think of the individual characters of a string being stored in consecutive locations of the computer's memory. For example, string *Hello World* could be thought of as follows:

H	e	l	l	o		W	o	r	l	d
---	---	---	---	---	--	---	---	---	---	---

Internally, computers store unique numeric codes for each character and not the actual characters themselves.

When to use a string

It is relatively straightforward to understand what a string is. However, a more necessary skill lies in the ability to recognise the need to use strings (in computer programs).

The art of computer programming usually involves the representation of some real world phenomena by using a programming language such as Python. In other words, programmers use the features of Python to model real world phenomena.

The number of real world situations that can be modelled using computer systems is endless. Examples include buying, selling, order delivery, flight scheduling, medical systems, processing of examination results, news, entertainment and even socialising.

At an even finer level of detail, the string datatype is a feature of Python that is suitable for representing many real world things.



KEY POINT: A string should be used in a computer program to model any real world data that could be composed of any combination of letters, numbers and punctuation symbols.

Examples of string data include names, addresses, phone numbers, passwords, email texts, Facebook posts, SMS text messages, tweets, product codes, descriptions – the list is endless. In fact, most of the data you see on your mobile phone and on the world wide web are represented by strings.

Novice programmers should be made aware of the fact that data which intuitively looks numeric are frequently represented in programs by strings. The reason for this is that these apparent numbers can often contain non-numeric data such as brackets, dashes or letters. Common examples include phone ‘numbers’, vehicle registration ‘numbers’ and Personal Public Service Numbers (PPSN).

String Operations

We already know we can display a string using the `print` command. But what are the other operations that can be carried out on strings?

By far the most important strings operations are *indexing* and *slicing*. Indexing is a programming technique used to access individual characters of a string. Slicing is a variation on this used to access multiple continuous characters from a string (known as a sub-string or a slice).

Other operations that can be carried out on strings are addition, multiplication, formatting and comparisons. Strings also support set operations such as in and not in. These set operations along with comparisons will be discussed in the section on programming logic.

Python also comes with a number of built in commands that can be used on strings as well as a comprehensive set of commands (known as *string methods*) specifically designed for working with strings.

String Indexing

In order to understand string indexing (and slicing) it is first necessary to understand the concept of an index. Consider the string `s` initialised in the line of code shown

```
s = "Hello World"
```

Internally, the computer represents the string `s` as a sequence of characters stored at contiguous memory locations. Each individual character is stored in its own separate memory location.

The individual characters in every Python string have a position. This position is relative to the first character in the string and is known as an *index*. Because the index is an offset from the first character, the index of the first character itself, in every Python, string is zero. Indices are said to be *zero-based*.

The diagram below depicts the index position of every character in the string `s`.



As can be seen the first character i.e. 'H' has an index of zero. This is important. The first character in a string always occurs at index position zero (this is often called the zeroth position), the second character at index position 1, the third character at index position 2 and so on. The string *Hello World* contains 11 characters and each character has a unique index ranging from 0 to 10.

In general, when there are n characters in a string the last character always occurs at index position $n - 1$. Therefore, if there are 5 characters in a string the last character occurs at index position 4.

Each individual character of a string can be accessed by enclosing the character's index inside square brackets. The square brackets must appear directly after the name of the string. For example, the first element of the above string can be accessed using `s[0]`. So,

```
s[0] → "H"
s[1] → "e"
s[2] → "l"
s[3] → "l"
s[4] → "o"
s[5] → " "
s[6] → "W"
s[7] → "o"
s[8] → "r"
s[9] → "l"
s[10] → "d"
```



KEY POINTS

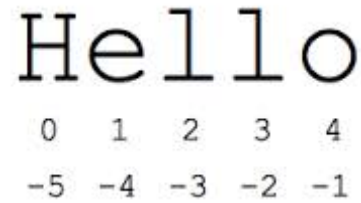
- ✓ Every character in a string has a unique index.
- ✓ The index is the character's position in the string.
- ✓ The first character is at index position zero.
- ✓ The last character in a string of length n is at index position $n-1$
- ✓ Indices are enclosed in square brackets

It is worth pointing out that Python, unlike many other programming languages does not support the concept of a character datatype. All of the single characters returned in the above example are actually strings i.e. single character strings of length 1.

Negative Indices

Python permits the use of negative numbers as indices.

The diagram to the right depicts the positive and negative indices of the string *Hello*.



As can be seen the last (rightmost) character of the string can be accessed using index -1 . Working from right to left the index of each character is one less than its predecessor.

Experiment!



A **pangram** is a sentence that uses every letter of the alphabet at least once. Study the program below and see if you can predict what it does? Record your prediction on the right hand side.

Key in (or copy+paste from GitHub) the program and make some changes? What happens?

```

1. # Program to demonstrate basic string operations
2.
3. # Initialise the string
4. pangram = "The quick brown fox jumps over the lazy dog!"
5. # 012345678901234567890123456789012345678901234567890123
6. # INDEXING
7. print(pangram[0])
8. print(pangram[1])
9. print(pangram[2+4])
10. print(pangram[14])
11. print(pangram[8])
12. print(pangram[43])
13.
14. # The index can be any valid Python expression
15. pos = 17
16. print(pangram[pos])
17. print(pangram[pos+1])
18.
19. # A general pattern used to find the last character
20. print( pangram[len(pangram)-1] )
    
```



Prediction



What one question (if any) would I still like to ask in relation to this example?

Common Pitfalls

It is worth pointing out a couple of very common pitfalls relating to strings and index numbers

The first thing to note is that strings are *immutable*. This means that once a string has been initialised it cannot be changed. For this reason, it is not permitted to use the index operator ([]) on the left hand side of an assignment statement.

For example, the string *Cavan* could not be changed to *Navan* as follows.

```
town = "Cavan"  
town[0] = "N"
```

This code results in a syntax error shown below:

```
Traceback (most recent call last):  
  File "C:/PDST/Work in Progress/Python Workshop/src/Session 3  
ts/Str Ops - syntax.py", line 3, in <module>  
    town[0] = "N"  
TypeError: 'str' object does not support item assignment
```

To 'change' the value of *town* to *Navan* the program needs to make a new assignment `town="Navan"` but be warned that the reference to the original string *Cavan* is now lost.

Secondly, if a program attempted to access a character in a string using an index that is too big (i.e. beyond the last character) for that string, Python returns a runtime error telling you that the index is *out of range*. For example, running the following code snippet would result in a runtime error being displayed because the index 5 is beyond the range of the string.

```
s = "Hello"  
print(s[5])
```

```
Traceback (most recent call last):  
  File "C:/PDST/Work in Progress/Python Workshop/src/Session 3  
ts/Str Ops - syntax.py", line 2, in <module>  
    print(s[5])  
IndexError: string index out of range
```

Out of range errors are also known as *out of bounds* errors. These type of errors can be a source of great frustration even for more experienced programmers.

String Slicing

Thus far we have used indexing to access individual characters from a string. The technique of indexing can also be used to extract a sub-string from a string. This is known as slicing. The part of the string that is extracted is known as the slice (i.e. sub-string).

Slicing is useful when we want to extract a specific piece of information out of a longer piece of information. For example, we may want to extract a share price from a web page displaying stocks.

In order to extract a slice from a string we need to know the indices of the desired slice's start and end positions. The slice is taken by specifying these values inside the square brackets. The values are separated by a full colon.

The technique of slicing is demonstrated by the following short program.

```

1. # Program to demonstrate basic string slicing
2.
3. # Initialise the string
4. pangram = "The quick brown fox jumps over the lazy dog!"
5. #      012345678901234567890123456789012345678901234567890123
6.
7. # Extract from index 1 up to but NOT including index 5
8. print(pangram[1:5]) # "he q"
9.
10. # Extract from index 17 up to but NOT including index 19
11. print(pangram[17:19]) # ox
12.
13. print(pangram[:19]) # "The quick brown fox"
14. print(pangram[20:26]) # "jumps"
15. print(pangram[26:]) # "over the lazy dog!"

```



The colon delimits the start and end positions of the slice we are interested in extracting. The resulting slice runs from the starting index *up to, but not including* the end. If the start is missing it is taken to be zero i.e. the first character of the string. If end is missing it is taken as the index of the last character in the string.



What do you think the statement `print(pangram[:])` would display?

String Addition and Multiplication


Strings, just like numbers can be added and multiplied.

The operation of adding one string to another is commonly referred to as *string concatenation*. We say one string is concatenated to another string to form a new (and longer) string. The plus operator, '+' is used to concatenate two strings.

The programs below illustrate how the plus operator is used to concatenate strings.


PROGRAM 1

```
word1 = "Leaving"
word2 = "Certificate"
word3 = "Computer"
word4 = "Science"
subjectName = word1 + word2 + word3 + word4
print(subjectName)
```




PROGRAM 2

```
pangram = "The quick brown fox jumps over the lazy dog!"
# 012345678901234567890123456789012345678901234567890123
print(pangram[:3] + pangram[16:])
```



PROGRAM 3

```
noun = input("Enter a singular noun: ")
print("The plural of "+noun+" is "+noun+"s")
```




Log your thoughts.

Each of the above programs contain a deliberate subtle error in their outputs. Can you suggest how the output could be enhanced?

Building up a string (string construction)

String concatenation is a useful technique for building up a string made up of separate strings that come from different sources. Let's say we wanted to construct an output string based on the pangram string we used earlier.

```
"The quick brown fox jumps over the lazy dog!"
```

This time however we want to ask the user to enter the colour of the fox and the name of the animal to jump over. The output string will be constructed based on the values entered by the user. Example outputs could be:

```
"The quick red fox jumps over the lazy horse!"
```

```
"The quick orange fox jumps over the lazy hen!"
```

One solution is as shown here.

Observe how the `outStr` is built up bit by bit as the program progresses.

Can you see any similarities between this and the technique for keeping running totals described earlier?


```
# Initialise the output string
outStr = "The quick "

# Ask the user for a colour
colour = input("Please enter a colour: ")

# Concatenate the colour to the output
outStr = outStr + colour
outStr = outStr + " fox jumps over the lazy "

# Ask the user for an animal
animal = input("Please enter an animal: ")
outStr = outStr + animal
outStr = outStr + "!"

# Display the output
print( outStr )
```




String multiplication

Multiplication of strings in Python is not very common. When a string is multiplied by some integer n , a new string is produced. This new string is the original string repeated n times.

The technique of string multiplication is exemplified by the program shown below. The output generated by the code is displayed to the right.

```
word1 = "Hello"
print(word1*3)
print(word1[1]*3)
print(word1[1:3]*3)
```



```
HelloHelloHello
eee
elelel
```



Programming Exercise 3.1

The string variable `a1Num` is initialised as shown here.

```
a1Num = "abcdefghijklmnopqrstuvwxyza1Num0123456789"
#           1           2           3           4           5           6
#           01234567890123456789012345678901234567890123456789012345678901
```

1. Match each index operation on `a1Num` (numbered 1-9 below) to the correct value on the right hand side (denoted by letters A-I)

1. <code>a1Num[3]</code>	A. ABCDEFGHIJKLMNOPQRSTUVWXYZ0
2. <code>a1Num[52:62]</code>	B. d
3. <code>a1Num[51:62]</code>	C. ABCDEFGHIJKLMNOPQRSTUVWXYZ
4. <code>a1Num[26:53]</code>	D. A
5. <code>a1Num[0:25]</code>	E. 0123456789
6. <code>a1Num[1]</code>	F. f
7. <code>a1Num[5]</code>	G. b
8. <code>a1Num[26:52]</code>	H. abcdefghijklmnopqrstuvwxyza
9. <code>a1Num[26]</code>	I. Z0123456789

2. Explain what is wrong with each of the following code snippets



```
name = "Mary"
print(name[4])
```

```
name = "Mary"
name[3] = "k"
```

Strings Formatting

Sometimes it is useful to be able to display a string that is made up of several pieces of data without having to use the string construction techniques just described.

Formatting string expressions enable us to display more than one piece of data in a single `print` statement. Consider the following statement:

```
print("Hello %s" % "World")
```

The first string after the opening brackets on each line is called the *format string*. This is the string that Python displays once it has it formatted. In this example the format string is *Hello %s*

The `%s` is a placeholder for Python to insert a string value into the format string. `%s` is an example of a Python *formatting type code* (aka format specifier).

The `%` immediately after the format string tells Python that what follows is a *string formatting expression*. This expression contains the actual value(s) for Python to substitute into the format string.

In the above statement, Python replaces `%s` the string literal, *World*, and the resulting formatted string *Hello World* is displayed.

Some common string formatting codes are listed below

- `%s` is used to format text (strings)
- `%d` is used to format integers
- `%f` is used to format floating point (decimal) numbers
- `%.2f` is used to format floating point numbers to 2 decimal places (rounded)



KEY POINT: The `print` command replaces each format code, *in strict sequence*, with a value of the appropriate datatype from the formatting expression.

If the format string contains more than one code, then the corresponding values in the expression must be separated by commas and enclosed by brackets.



**The four statements below each generate the same output.
What output is displayed?**

```
print("2 + %d = 4" %2)
print("2 + %d = %d" %(2, 4))
print("%d + %d = %d" %(2, 2, 4))
print("%d + %d = %d" %(2, 2, 2+2))
```



The output generated by the following code is displayed to the right.

```
print("%s" %"String 1")
print("%s %s" %("String 1", "String 2"))
print("%s + %s = %d" %("2", "3", 2+3))
print("%d + %d" %(2, 3))
print("%d + %d = %d" %(2, 3, 2+3))
print("%f" %3.14)
```



```
String 1
String 1 String 2
2 + 3 = 5
2 + 3
2 + 3 = 5
3.140000
```



Experiment!

See if you can figure out what the following code does.

```
print("%s" %3)
print("%d" %3.14)
print("%f" %3)
print("%f" %"Hi!")
```



Finally, it is worth noting that string formatting expressions can, and frequently do, contain variables and/or other Python expressions. This is exemplified in these two snippets.

```
msg = "Hi %s. How are you?"
name = "Hal"
print(msg%name)
```



```
import math
r = 5
print("Radius: %d, Area: %.2f" %(r, 2*math.pi*r))
```



Built in string commands

So, let's say we have a string `s` declared and initialised as follows:

```
s = "The quick brown fox jumps over the lazy dog!"
```

The table below outlines the use of three different built-in string commands – `len`, `min` and `max` - and the output they produce when applied to `s`.

Command	Description	Output
<code>len(s)</code>	Returns the length of the string, <code>s</code> . This is the number of characters in the string.	44
<code>min(s)</code>	Returns the minimum item in <code>s</code> . (See <code>ord</code> on the next page.)	"!"
<code>max(s)</code>	Returns the maximum item in <code>s</code> . (See <code>ord</code> on the next page.)	"z"

Another built in command that relates to strings is `str`. The `str` command is used to convert any object into string. This means a program can dynamically change the type of any object to a string type.

One practical use of `str` is to convert numbers to strings when we want to display them as part of an output message. Observe how `runningTotal` is converted to a string in the last line of the following program.

```
1. # Program to calculate a running total
2.
3. # Initialise the variable
4. runningTotal = 0
5.
6. # Keep a running total of the amounts entered
7. price1 = float(input("Enter the 1st price: "))
8. runningTotal = runningTotal + price1
9. price2 = float(input("Enter the 2nd price: "))
10. runningTotal = runningTotal + price2
11. price3 = float(input("Enter the 3rd price: "))
12. runningTotal = runningTotal + price3
13.
14. # Display the output
15. print("Total amount is €%s" %str(runningTotal))
```



KEY POINT: Python is *dynamically typed*. This means that when a program is running, numeric data can be converted to strings and vice versa.

Coding Systems (ord and chr)

Two other built in Python commands that relate to strings are `ord` and `chr`

Programmers should be aware that all data is represented internally by computers using a coding system. A *coding system* is one which uses combinations of binary digits to represent data values uniquely. **ASCII** (American Standard Code for Information Interchange) and **Unicode** are the names of two very widely used coding systems. An illustration of the full ASCII character set is shown below.

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	␣	Space	64	40	100	␣	@	96	60	140	␣	␣
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	␣	A	97	61	141	␣	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	␣	B	98	62	142	␣	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	␣	C	99	63	143	␣	c
4	4	004	EOT (end of transmission)	36	24	044	␣	␣	68	44	104	␣	D	100	64	144	␣	d
5	5	005	ENQ (enquiry)	37	25	045	␣	␣	69	45	105	␣	E	101	65	145	␣	e
6	6	006	ACK (acknowledge)	38	26	046	␣	␣	70	46	106	␣	F	102	66	146	␣	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	␣	G	103	67	147	␣	g
8	8	010	BS (backspace)	40	28	050	{	{	72	48	110	␣	H	104	68	150	␣	h
9	9	011	TAB (horizontal tab)	41	29	051	}	}	73	49	111	␣	I	105	69	151	␣	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	␣	J	106	6A	152	␣	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	␣	K	107	6B	153	␣	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	␣	L	108	6C	154	␣	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	␣	M	109	6D	155	␣	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	␣	N	110	6E	156	␣	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	␣	O	111	6F	157	␣	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	␣	P	112	70	160	␣	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	␣	Q	113	71	161	␣	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	␣	R	114	72	162	␣	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	␣	S	115	73	163	␣	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	␣	T	116	74	164	␣	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	␣	U	117	75	165	␣	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	␣	V	118	76	166	␣	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	␣	W	119	77	167	␣	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	␣	X	120	78	170	␣	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	␣	Y	121	79	171	␣	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	␣	Z	122	7A	172	␣	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	␣	[123	7B	173	␣	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	␣	\	124	7C	174	␣	
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	␣]	125	7D	175	␣	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	␣	^	126	7E	176	␣	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	␣	_	127	7F	177	␣	DEL

Source: www.LookupTables.com


The two built-in functions – `ord` and `chr` - are used to convert back and forth between characters and ASCII.

`ord(c)` Returns the ASCII representation (or Unicode code point) of the character `c`.

`chr(i)` This is the inverse of `ord`. The function returns a single character as a string whose ASCII representation (or Unicode code point) is the integer `i`.

The programs below illustrate the use of `ord`.

```
1. print(ord('A'))
2. print(ord('A')+25)
3. print(ord('Z'))
4. print(ord('a'))
5. print(ord('1'))
```



Line 1 outputs 65. This is the ASCII representation for the character 'A'


Line 2 outputs 90. The ASCII representation for the character 'A' is 65 and 65+25 is 90.



Use the ASCII table shown on the previous page to predict the outputs of lines 3, 4 and 5 of the above program.

This next piece of code illustrates the use of `chr`.

```
1. print(chr(65))
2. print(chr(90))
3. print(chr(97))
4. print(chr(49))
```



Line 1 outputs character 'A'

Line 2 outputs character 'Z'

Line 3 outputs character 'a'

Line 4 outputs character '1'



Experiment!
Try running the following two pieces of code and see if you can explain what is going on.

```
print(chr(ord('A')))
print(ord(chr(64)))
```

```
inStr = input("Enter any character: ")
outStr = chr(ord(inStr)+1)
print(outStr)
```



Programming Exercise 3.2

A **Caesar cipher** is a way of encoding strings by shifting each letter by a fixed number of positions (called a key) in the alphabet. For example, if the key value is 1, the string 'LCCS' would be encoded as 'MDDT'. Each letter of LCCS is moved on by one.

The short program below prompts a user to enter a single character and then it calculated and displays the character with the next ordinal number e.g. if the user enters the letter *A*, the program will display *B*.

```

inStr = input("Enter any character: ")
outStr = chr(ord(inStr)+1)
print(outStr)

```



Run the program and once you understand what it does, change it so that it "encodes" a 6 letter string using a key value of 1 e.g. "Python" becomes "Qzuiop".

In order to complete this task, you will need to understand:

- ✓ Variables and assignments
- ✓ Strings (indexing and concatenation)
- ✓ Data representation (ASCII/Unicode)
- ✓ How to use the `ord`, `chr` and `print` built-in functions



Log your thoughts.


Suggest ways by which this task could be altered to make it either more challenging or less challenging to complete?

String Methods

String methods are special commands that can be used to manipulate and perform common/useful operations on string values.

The official Python documentation <https://docs.python.org/3/library/stdtypes.html#string-methods> lists and describes over 40 built in string methods. The program below demonstrates six of these – `capitalize`, `upper`, `isupper`, `islower`, `count` and `find`.

```
1. # Program to demonstrate the use of common string methods
2.
3. pangram = "the quick brown fox jumps over the lazy dog!"
4.
5. print(pangram.capitalize())
6. print(pangram.upper())
7. print(pangram.isupper())
8. print(pangram.islower())
9. print(pangram.count("o"))
10. print(pangram.find("fox"))
11. print(pangram.find("Fox"))
12. print(pangram.find("the"))
13. print(pangram.find("z"))
```



When the above program is run it generates the following output.

```
The quick brown fox jumps over the lazy dog!
THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG!
False
True
4
16
-1
0
37
```

Each of the nine lines of output correspond, in sequence, to the result of the string methods used inside each of the nine `print` statements.

The program, along with the output it generates should be studied carefully again after reading the method descriptions on the next page.

As can be seen, the string methods operate using the same dot notation as we used for `turtle` and `pygame` objects earlier. For any string variable `s`, we can use a string method by typing `s`, followed by a dot, followed by the method's name. This is denoted as follows:

<string-variable-name>.<method-name>

Method Name	Description
<code>s.capitalize()</code>	Returns a new string with the first letter of <code>s</code> in capital letters
<code>s.upper()</code>	Returns a new string with all the letters of <code>s</code> in upper case
<code>s.isupper()</code>	Returns the value <code>True</code> if all the letters in <code>s</code> are in upper case. If all the letters are not in upper case the method returns <code>False</code> . <code>True</code> and <code>False</code> are both Python keywords will be explained in the next section on programming logic.
<code>s.islower()</code>	Returns the value <code>True</code> if all the letters in <code>s</code> are in lower case. If all the letters are not in lower case the method returns <code>False</code> . <code>True</code> and <code>False</code> are both Python keywords will be explained in the next section on programming logic.
<code>s.count("o")</code>	This methods counts the number of times any string e.g. <code>o</code> occurs in the string <code>s</code> and returns the answer.
<code>s.find("Fox")</code>	This method looks for a string e.g. <code>Fox</code> in the string <code>s</code> and if it finds it returns the index position of the <code>F</code> . If the search sting is not found the method returns <code>-1</code> .
<code>s.replace(t, u)</code>	Replaces all occurrences of <code>t</code> in <code>s</code> with <code>u</code> . <code>s</code> , <code>t</code> and <code>u</code> are all strings. <code>s</code> remains unchanged if it does not contain <code>t</code> .

STUDENT TIP

Independent student learning can be promoted by encouraging students to use the official Python documentation as much as possible.



Programming Exercise 3.3

The code below initialises four string variables

```
# Initialise four string variables
lowerLetters = "abcdefghijklmnopqrstuvwxy"
upperLetters = "ABCDEFGHIJKLMNPOQRSTUVWXYZ"
allLetters = lowerLetters+upperLetters
digits = "0123456789"
```

Match each *print* statement (numbered 1-14 below) to the corresponding output (denoted by letters A-N)

- | | |
|--|--|
| 1. <code>print(lowerLetters.upper())</code> | A. <code>a</code> |
| 2. <code>print(upperLetters.lower())</code> | B. <code>Z</code> |
| 3. <code>print(upperLetters.find("h"))</code> | C. <code>abcdefghijklmnopqrstuvwxy</code> |
| 4. <code>print(lowerLetters.find("h"))</code> | D. <code>0123456789</code> |
| 5. <code>print(digits.count("123"))</code> | E. <code>ABCDEFGHIJKLMNPOQRSTUVWXYZ</code> |
| 6. <code>print(lowerLetters.capitalize())</code> | F. <code>-1</code> |
| 7. <code>print(digits.count("0"))</code> | G. <code>1</code> |
| 8. <code>print(digits.lower())</code> | H. <code>z</code> |
| 9. <code>print(min(allLetters))</code> | I. <code>A</code> |
| 10. <code>print(max(allLetters))</code> | J. <code>7</code> |
| 11. <code>print(allLetters.count("a"))</code> | K. <code>0</code> |
| 12. <code>print(allLetters.count("aA"))</code> | L. <code>1123456789</code> |
| 13. <code>print(allLetters.count("ABC"))</code> | M. <code>abcdefghijklmnopqrstuvwxy</code> |
| 14. <code>print(digits.replace("0", "1"))</code> | N. <code>6</code> |

Additional Notes (Sequences)

Strings like almost everything else in Python are examples of objects. The string type is a core data type that is built into the language. Some other examples of built in types include numbers, lists, tuples, dictionaries and files. The official documentation on all Python's types can be viewed online at <https://docs.python.org/3/library/stdtypes.html> and is well worth a visit.

More specifically Python classifies strings as part of a more fundamental kind of object known as a sequence. A sequence is defined as an ordered collection of objects

Python supports three basic sequence types (`list`, `tuple` and `range`), a text sequence type (`str`) and a number of binary sequence types use to work with binary data.

All sequence types share a common set of operations, called sequence operations. The table below, which is taken directly from the official Python documentation, lists these common sequence operations in ascending order of precedence.

Operation	Result
<code>x in s</code>	<code>True</code> if an item of <code>s</code> is equal to <code>x</code> , else <code>False</code>
<code>x not in s</code>	<code>False</code> if an item of <code>s</code> is equal to <code>x</code> , else <code>True</code>
<code>s + t</code>	the concatenation of <code>s</code> and <code>t</code>
<code>s * n</code> or <code>n * s</code>	equivalent to adding <code>s</code> to itself <code>n</code> times
<code>s[i]</code>	<code>i</code> th item of <code>s</code> , origin 0
<code>s[i:j]</code>	slice of <code>s</code> from <code>i</code> to <code>j</code>
<code>s[i:j:k]</code>	slice of <code>s</code> from <code>i</code> to <code>j</code> with step <code>k</code>
<code>len(s)</code>	length of <code>s</code>
<code>min(s)</code>	smallest item of <code>s</code>
<code>max(s)</code>	largest item of <code>s</code>
<code>s.index(x[, i[, j]])</code>	index of the first occurrence of <code>x</code> in <code>s</code> (at or after index <code>i</code> and before index <code>j</code>)
<code>s.count(x)</code>	total number of occurrences of <code>x</code> in <code>s</code>

Note that concatenation, multiplication, indexing, and slicing – all described in this section in the context of strings – are all common sequence operations and, therefore, are equally applicable to all the other sequence types.

Sequences can either be *mutable*, meaning that their values can be changed once they have been created, or *immutable*, meaning that their values cannot be changed after creation.

Python considers strings to be as 'elemental' as numbers and accordingly implements them as immutable sequences. Therefore, once a variable has been initialised with a string value, that value can never be overwritten. A program may *appear* to change a string by assigning the same variable to a new or different string. In practice when this happens, Python creates a new string and the original string remains in memory without a reference.

Sequences are also distinguished from one another by their own specific set of operations that are not available to other types. For example, the string methods described earlier in this section are specific to string sequences and cannot be used on other sequence types.

Finally, it is worth noting that unlike many other high level programming languages, Python does not provide any built-in support for the character datatype. In Python, a single character is simply treated as a string of length one.



KEY POINT: Strings are immutable.

They are implemented as sequences of one character strings.



BREAKOUT ACTIVITIES

BREAKOUT 3.1: School Survey Web Page

Among the many applications of Python strings are file processing and HTML files generation. Consider the following scenario.

Your school loves to hear feedback from teachers and students on a wide variety of topics and issues – so much so, that the school website even has a dedicated page for hosting weekly surveys. This week’s survey, shown below, relates to the menu in the school canteen.

School Survey

Do you think the menu in the school canteen should be changed?

First name:

Last name:

Thank you for taking part in the school survey".

The format of the web page is the same for every survey. Surveys differ from each other only in the actual question being asked, and the text of possible answers which always appear in two buttons displayed side-by-side on the page. Participants are always asked to enter their first name and last name.

The questions and answer options for the next three surveys approved by the school board are shown below.

- | | | |
|---|--------|-----------|
| 1. Destination for School Tour next year? | Rome | Barcelona |
| 2. Exam year students should be allowed to take part in <i>all</i> extra-curricular activities? | Agree | Disagree |
| 3. Computational Thinking should be taught as part of every subject | Always | Sometimes |

The problem is that every week the school 'techie', Ms E. Fish, has to edit the HTML file with next week's survey question and answers, and upload it to the web server to make it live on the school web site.

The 'techie', who is also the school's Computer Science teacher understands HTML, but with the new term fast approaching is very busy developing content for the learning outcomes in the new LCCS specification as well as coming up with ideas for teaching the course through the lens of the four Applied Learning Tasks.

Even though the HTML code behind the survey page – shown below - is fairly straightforward, she needs come up with an efficient solution so that her colleague, Mr. Chips (the maths teacher who does not possess any knowledge of HTML), can upload the new survey.

```
1 <!DOCTYPE html>
2 <html>
3 <body>
4
5 <h2>School Survey</h2>
6 <p>Do you think the menu in the school canteen should be changed?</p>
7 <form action="">
8   First name:<br>
9   <input type="text" name="firstname" value="">
10  <br>
11  Last name:<br>
12  <input type="text" name="lastname" value="">
13  <br><br>
14  <input type="submit" value="Yes">
15  <input type="submit" value="No">
16 </form>
17
18 <p>Thank you for taking part in the school survey".</p>
19
20 </body>
21 </html>
```



survey.html

The solution Ms. E. Fish comes up with is as follows:

1. She will create a HTML survey template file with placeholders for the question and answer text to be used in the survey. The name of this file will be `survey_template.html`.
2. She will create a normal text file to store the question and answer text for the next survey. The name of this file will be `survey.txt`.

3. She will write a Python program that generates a HTML for the next survey using two files `survey_template.html` and `survey.txt`. The name of the generated file will be `survey.html`.

All Mr. Chips will have to do every week is edit the text file and run the program. No need to worry about HTML - easy!

Ms. E. Fish was delighted that her proof of concept worked and so commenced the job of implementing her solution. She created the two files - `survey_template.html` and `survey.txt` – as per design. These are shown below. Notice the three placeholders for the text on lines 6, 14 and 15 in the `survey_template.html`


```

1 <!DOCTYPE html>
2 <html>
3 <body>
4
5 <h2>School Survey</h2>
6 <p><place-holder-1></p>
7 <form action="">
8   First name:<br>
9   <input type="text" name="firstname" value="">
10  <br>
11  Last name:<br>
12  <input type="text" name="lastname" value="">
13  <br><br>
14  <input type="submit" value="<place-holder-2">
15  <input type="submit" value="<place-holder-3">
16 </form>
17
18 <p>Thank you for taking part in the school survey".</p>
19
20 </body>
21 </html>
22

```



`survey_template.html`

 survey - Notepad

File Edit Format View Help

Destination for School Tour next year?

Rome

Barcelona

Exam year students should be allowed to take part in all extra-curricular activities?

Agree

Disagree

Computational Thinking should be taught as part of every subject

Always

Sometimes



`survey.txt`

However, soon after she started writing the Python program to generate the weekly survey, Ms. E. Fish realised that she needed to prioritise lesson planning for the fast approaching LCCS.

With the program, shown below, at unit test stage, unless someone steps in, the project is looking like it might have to be shelved and the destination for next year's school tour is still undecided.

```
# Open the survey template
htmlFile = open("survey_template.html","r", encoding="utf-8") # Open the file
htmlStr = htmlFile.read()
htmlFile.close() # close the file

# Open the data file
surveyFile = open("survey.txt","r")
surveyQuestion = surveyFile.readline()


*** YOUR CODE GOES HERE ***

surveyFile.close() # close the file

# Now replace the 'markers' from the template file with the runtime data
htmlStr2 = htmlStr.replace("<place-holder-1>", surveyQuestion)

*** YOUR CODE GOES HERE ***

# Write the survey page
htmlFile = open("survey.html","w", encoding="utf-8") # Open the file for writing
htmlFile.write(htmlStr2)
htmlFile.close() # close the file
```



The desired output is shown below.

School Survey

Destination for School Tour next year?

First name:

Last name:

Rome

Barcelona

Thank you for taking part in the school survey.

Suggested Activities

1. Restate the problem in your own words.



2. Complete the program by adding the necessary code inside the red rectangles to get it displaying the desired output. You will need to make sure the two files - `survey_template.html` and `survey.txt` – are in the runtime directory.



3. Outline how you might test your system.



4. Describe in detail the steps you would have to take in order to introduce a third answer button into the survey



BREAKOUT 3.2: RSS Feed Analysis

Really Simple Syndication (RSS, aka Rich Site Summary) is a technology that allows an end-user to have information delivered automatically from selected website(s) to a device. The information is referred to as a *feed*. Users can subscribe to receive RSS feeds from specific sites, and in this way, keep up-to-date with the information they are interested in.

Typically, these sites contain information relating to business, jobs, blogs, news, entertainment etc. Once a user has subscribed, feeds containing any new information on these sites are automatically 'sent' to that user who can then view them using a RSS reader (typically a web browser).



Use your favourite browser to locate and record the names and URLs of some RSS sites that interest you.

1.

2.

3.

4.

5.

In this session we will modify and write code that analyses data from a live RSS feed of your choice.

A working program that connects to a URL and delivers a live RSS from that URL to a variable called `feed` is provided as a starting point. The program listing is shown on the next page.

Notice that a large section of the code is enclosed in a black box. The black box is used to indicate code that is needed for the program to run, but not necessary to understand. In this example, the code inside the black box tells Python to read a RSS feed from the URL specified as a string on line number 24.

Program to read a RSS feed


The listing below pulls live headline news publically available from Apple's RSS feed. The data is stored in the string variable, `feed`.

Line 24 is an assignment statement. The right hand side tells Python to used code (inside the black box) to read the contents of a RSS feed. The full URL of the site from which to read the feed is *hard-coded* as a string in this line. The feed itself is returned as a string and stored in the variable `feed`.

```

1. from urllib.request import urlopen
2. from xml.dom import minidom
3. import collections
4.
5. # extract the headlines from the feed
6. def extractString(doc):
7.     str = ""
8.     for node in doc.getElementsByTagName('channel'):
9.         for title in node.getElementsByTagName('title'):
10.            str = str + title.firstChild.data + "\n"
11.     return str
12.
13. # extract the feed from the url
14. def getRSSString(url):
15.     results = []
16.     rssString = ""
17.     results.append(minidom.parse(urlopen(url)))
18.     for webDoc in results:
19.         rssString = rssString + extractString(webDoc)
20.     return rssString
21.
22. # PROGRAM START FROM HERE ...
23. # Read the RSS feed from the URL provided
24. feed = getRSSString("https://www.apple.com/main/rss/hotnews/hotnews.rss")
25. # Display the results
26. print(feed)
27. # Display the number of lines
28. print("There were %d lines in this feed" %feed.count("\n"))

```



Line 26 displays the contents of the `feed` on the output console.

Line 28 displays the number of lines in `feed`. It does this by using the `count` command (method) to count the number of newline characters in `feed`. (A standard technique used to count the number of lines in a piece of text is to count the number of occurrences of the newline character, `'\n'`.)

We are now ready to make changes to the code.

Suggested Activities

1. Download (from GitHub) or key in the full program and make sure it runs properly.
2. The feed URL is hard-coded into the program (line 24). This means that a programmer must change the code every time a different feed needs to be read.
Modify the program so that it reads the URL from the first line of a text/data file.

Hint #1: You will need to create a file first (e.g. `feeds.txt`), and save it in your runtime directory. The file might look something like this:



```

feeds - Notepad
File Edit Format View Help
http://www.rte.ie/news/rss/news-headlines.xml
https://www.apple.com/main/rss/hotnews/hotnews.rss
http://www.siliconrepublic.com/feeds/
https://weather-broker-cdn.api.bbci.co.uk/en/forecast/rss/3day/2643123

```

Hint #2: A solution can be obtained by using four of following lines (in a different order).

```

feedFile.close()

feedFile.write("https://www.apple.com/main/rss/hotnews/hotnews.rss")

feedURL = feedFile.readline()

feed = getRSSString(feedFile)

feedFile = open("feeds.txt", "w")
feedFile = open("feeds.txt", "r")

feed = getRSSString(feedURL)

```

3. Extend the code so that it displays the entire feed a) capitalised, b) in upper case and c) in lower case.
4. Write a line of code that replaces every occurrence of a specific word in the feed with another word of your choosing e.g. replace the word 'Apple' with the word 'Microsoft'.
Display the new string.
5. Write a code snippet to count and add the number of vowels in the feed. (Later we will use the `plotly` library to display these data on a bar chart.)

6. Add the following code snippet to the end of your program and run it.

```
words = feed.split()  
print(collections.Counter(words).most_common(5))
```



What happened when you added the above code?

What do these two lines do?

Can you condense these two lines into one line that does the same thing?

-
-
-
7. Write code to replace the first **two** occurrence of a specific word/phrase in the feed with another word/phrase of your choosing. (You will need to identify some word/phrase that occurs at least twice yourself.) The output should display the entire feed with the new word/phrase in place of the original two.

Hint: You will need to use the techniques of indexing/slicing and concatenation to construct the output string.

8. Same as previous activity except, instead of replacing the two words, your program should apply the Caesar cipher algorithm implemented earlier in this section.
9. Browse to the official Python site documentation on string methods (see link below) and identify some method(s) that you have not already used. Now, use the feed from this activity to test drive your new string method.

<https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str>

10. Read the listing below and predict what it will do.
Now key it in and run it. Was your prediction correct?

```
1. # A program to read and display a web page
2. import urllib.request
3.
4. # Open the URL
5. page = urllib.request.urlopen("https://www.compsci.ie/")
6. # Read and decode
7. text = page.read().decode("utf8")
8. # Display
9. print(text)
```



Experiment!

What happens if you change the URL on line 5?



Discuss in groups ways an activity (or sequence of activities) might be developed around the above code. (You may find it useful to reflect on the types of activities carried out in this breakout session on RSS feeds.)

Section 4

Lists

Introduction

A *list* is a collection of variables. The list type is commonly referred to as an *array* in other programming languages but as we shall see there are some subtle differences between lists and traditional arrays.

Lists are useful because they provide us with a means of grouping several variables into a single variable. The value of each variable is known as a *list element* and these can be accessed by using the same indexing/slicing techniques as was described in the previous section on strings.

Just think about it. Thus far we have been using variables to store single values. This has been the case regardless of the variable's datatype i.e. whether it is a string or numeric, only one value at a time can be stored in it.

The main purpose of lists is to provide a mechanism for dealing with multiple values using a single variable.

Strings are a special type of list where the individual elements are the individual characters that make up the string. One key difference between lists and strings, however, is that lists are *mutable*, whereas strings are not.



KEY POINT: A list is a Python programming construct useful for modelling any real world data that can be grouped together under a common name.

Examples of lists include teacher names, schools, subjects, teams, lists of friends, a book list, a list of tweets, play lists (songs), shopping lists, a list of instructions, lists of countries, lists of capital cities, days of the week, months of the year, holiday dates, lists of numbers (e.g. lottery, ages, salaries, sales figures, heights etc.). The list of lists is endless!

Consider for example, scenarios where we needed to keep track of the number of times a particular event occurred. Let's say there are multiple possible outcomes and we need to maintain a count for each one individually. For some reason we might be asked to write a traffic survey program that counts the numbers of pedestrians, bicycles, cars, vans, HGVs etc. passing a particular point at a particular time of the day. Without lists we would need to maintain each count in a separate variable.

Creating Lists

The simplest kind of list is an empty list i.e. one that contains no elements. Empty lists are created by using a pair of square brackets as illustrated in the code below.

The code shown here to the right creates five different empty lists. The square brackets on the right hand side of each assignment tell Python that the datatype of the variable named on the left hand side is a list.

```

boysNames = []
girlsNames = []
favouriteSongs = []
fruits = []
vehicleCount = []
accountDetails = []
  
```

Even though these lists do not contain any data, their construction means that the program can add data to them at a later stage.

Remember lists are mutable. This means that values can be added, deleted or simply changed after the list has been created. The following code snippet illustrates how to initialise lists with data.

```

boysNames = ['John', 'Jim', 'Alex', 'Fred']
girlsNames = ['Sarah', 'Alex', 'Pat', 'Mary']
favouriteSongs = ['Moondance', 'Linger', 'Stairway to Heaven']
fruits = ['Strawberry', 'Lemon', 'Orange', 'Raspberry', 'Cherry']
vehicleCount = [0, 0, 0, 0, 0, 0]
accountDetails = [1234, 'xyz', 'Alex', '1 Main Street', 827.56]
  
```

Note the use of square brackets on the right hand side and the use of commas to separate the individual list elements. The number of elements in the lists are 4, 4, 3, 5, 6 and 5 respectively.

Notice also that `boysNames`, `girlsNames`, `favouriteSongs` and `fruits` are all lists of strings; `vehicleCount` is a list of numbers, and `accountDetails` is a list of values that have a mixture of different underlying datatypes.



KEY POINT: A Python list can be made up of elements having different datatypes.

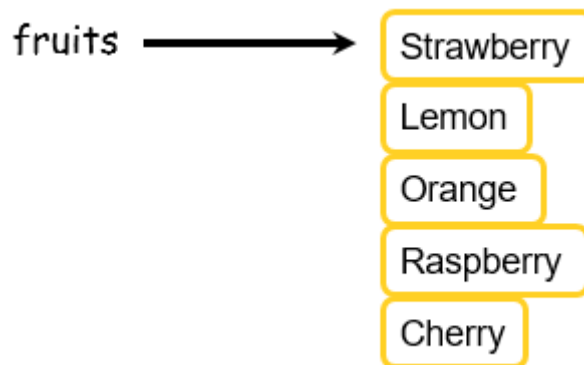
One of the key differences between lists and traditional arrays used in other programming languages is that a Python lists can be made up of data having a mixture of different datatypes, whereas the elements of an array must all be of the same data type.

It is useful to form a mental image of how lists are represented internally by the computer. Lists are frequently depicted in either a horizontal or vertical fashion as shown here.

A horizontal memory representation for the list `boysNames` would look like this.



A vertical memory representation for the list `fruits` would look as follows.



The key point is that the elements of the list should be envisaged in neighbouring memory locations (just like the individual characters of a string as described in the previous section).



List five things you have learned about lists so far in this section.

1.

2.

3.

4.

5.

Common List Operations

Because lists and strings are both sequences, all of the common basic operations work in the same way for lists as they do for string. Specifically, lists can be added to one another (concatenation), multiplied together, indexed and sliced.

The commands `min`, `max`, and `len` also work for lists in the same way as they do for strings returning the minimum value, the maximum value and the number of elements (i.e. the length) in the list respectively.

Concatenation

Lists can also be constructed by concatenating two existing lists together. For example, we could join `boysNames` and `girlsNames` together using the concatenation operator (+) to form a new list called `names` as follows.

```
1. boysNames = ['John', 'Jim', 'Alex', 'Fred']  
2. girlsNames = ['Sarah', 'Alex', 'Pat', 'Mary']  
3. names = boysNames + girlsNames  
4. print(names)
```



Notice that square brackets are not used on line 3.

When a list is used without square brackets like this, Python takes it that every element in the list is to be used.

The `print` command on line 4 displays the entire contents of the list. The output generated by the above code is shown here.

```
['John', 'Jim', 'Alex', 'Fred', 'Sarah', 'Alex', 'Pat', 'Mary']
```

The example below generates the exact same output, demonstrates that concatenation does not always result in a new list being created.

```
1. boysNames = ['John', 'Jim', 'Alex', 'Fred']  
2. girlsNames = ['Sarah', 'Alex', 'Pat', 'Mary']  
3. boysNames = boysNames + girlsNames  
4. print(boysNames)
```



The list, `boysNames` is *extended* to include the list of girls' names.

Some Insights

The previous example provides us with an opportunity to gain some deeper insights into the Python programming language.

Observe, that by the end of the example, the list variable `boysNames` will contain the names of four girls (at least we *think* they are girls) – *Sarah, Alex, Pat* and *Mary*.

Just because we (humans) know that *Mary* and *Sarah* are definitely girls' names, it does not mean that Python knows this too. In fact, unlike humans, Python does not know the difference between a girl's name and a boy's name. (This is because the language was not designed to include any built in features to make such a distinction.). To Python, names are all just strings.

STUDENT TIP

Teachers should be aware that it is a common misconception for students to think that the name of a variable somehow determines the values that can be assigned it.

The fact that the previous program instructs Python to assign the girls' names to a variable that looks like it was named by the programmer to store the boys' names could mean one of two things. Either

- a) `boysNames` was a poor choice of a variable name made by the programmer or,
- b) the assignment was a mistake, again, made by the programmer

In both cases the programmer is the person responsible.



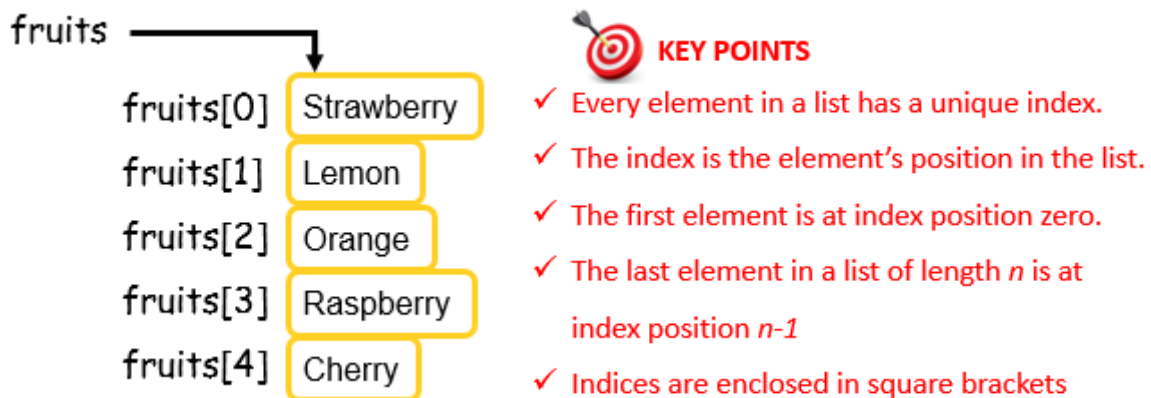
KEY POINT: Python has no understanding of natural language syntax and therefore has no way of inferring the intention of the programmer from the code.

List Indexing

Recall from the section on strings, that indexing is a technique used to access individual elements of a list. List indexing works just like string indexing.

A list element is accessed by using an index which is a zero-based positional value for that element. As was the case with strings, the index must be an integer (or an expression that evaluates to an integer), and, must be enclosed inside square brackets.

The graphic below depicts how index numbers can be included as part of the ‘mental image’ we formed for lists earlier.



Every list element is uniquely referenced by the list name and an index number.


Example program (fruit machine v1)

One of the reasons that computer programming is sometimes referred to as an art is because programmers can express creativity and imagination through the medium of their code.

The next example program demonstrates how to combine the use of random numbers and lists to simulate the operation of a fruit machine.

Read the program carefully and see if you can figure out how it works.

```
1. # Program to simulate a fruit machine!  
2. import random  
3.  
4. # initialise the list of fruits - 5 elements  
5. fruits = ['Strawberry', 'Lemon', 'Orange', 'Raspberry', 'Cherry']  
6.  
7. # generate three random numbers between 0 and 4 incl.  
8. selection1 = random.randint(0, 4)  
9. selection2 = random.randint(0, 4)  
10. selection3 = random.randint(0, 4)  
11.  
12. # show the results - display the fruits  
13. print(fruits[selection1])  
14. print(fruits[selection2])  
15. print(fruits[selection3])
```



- Line 5 initialises a list of fruits.
- Lines 8, 9, and 10 each generate a random number between zero and four inclusive
- Lines 13, 14, and 15 each display an element from the list using the random numbers as the index.



Look up the online documentation for the Python *random* library.
What does the *choice* command do?
How might *choice* be used in the above program?


One final point worth noting is that when a list element is accessed, the datatype of the resulting object is the same as the datatype of the list element.

Changing the value of list elements

Lists are mutable objects. This means that the elements of a list can be changed (as well as accessed). Recall that this is not possible with strings because they are immutable.

It is therefore 'legal' to apply the index operator to a list variable on the left hand side of an assignment statement. The following short program demonstrates this.

```
1. fruits = ['Strawberry', 'Lemon', 'Orange', 'Raspberry', 'Cherry']
2. print(fruits)
3.
4. fruits[0] = 'Apple'
5. fruit = 'Melon'
6. fruits[1] = fruit
7. fruits[2] = 'Raspberry'
8. fruits[3] = fruits[4]
9. fruits[4] = 'Pineapple'
10.
11. print(fruits)
```



- Line 1 initialises a list of fruits and line 2 displays the contents of the list
- Lines 4-9, each make separate changes to the individual elements in the list that are 'housed' at the given index
- Line 11 displays the list again

When it is run the program displays the output shown below. Can you figure it out?

```
['Strawberry', 'Lemon', 'Orange', 'Raspberry', 'Cherry']
['Apple', 'Melon', 'Raspberry', 'Cherry', 'Pineapple']
```



Log any thoughts you have in relation to Python lists.

Index ‘Out of Range’ Errors


At runtime, Python always checks that index numbers lie *within bounds* for the object they are being used to access.

A list index will be *out of bounds* if it lies beyond the range of the list. If Python attempts to access a list element using an index that is out of bounds, it returns an *out of range index error* and the program will crash i.e. stop functioning.

The whole idea of testing is to safeguard against system crashes happening in live (production) code.

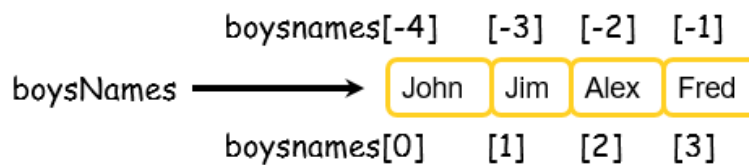
```

1. # Initialise a list of 5 fruits and try and access the 6th one
2. fruits = ['Strawberry', 'Lemon', 'Orange', 'Raspberry', 'Cherry']
3. print(fruits[5]) # <-- Syntax Error - out of bounds
4.
5. # Q. Why will the following lines not be executed?
6. print("Hello World")
7. lifeSupport = True
8. # A. Because Python stops when it encounters a syntax error
  
```



Run the code and you will see that it *crashes* on line 3, and lines 6 and 7 never get executed. Programmers spend much of their time correcting their own syntax errors.

Negative indices can be used on lists in just the same way as they could be used on strings. The last element of a Python list has an index of -1 . Working backwards, the index of each element is one less than its predecessor. Therefore, the valid indices for a list made up of **four** elements would be -4 to 3 inclusive. This is illustrated below.



KEY POINT: In general, the index numbers of any sequence of length of n must lie within the range of $-n$ and $n - 1$. So, $-n \leq \text{index} \leq n - 1$

List Slicing

Lists can be quite long and sometimes we might just be interested in processing a portion of the data they contain. We can extract sub-lists from lists using the exact same technique that we used to extract substrings from strings earlier i.e. *slicing*.

A slice is a list of consecutive elements taken from another (larger) list.

Slices are created using the square brackets index operator. As was the case with strings, the colon delimits the start and end positions of the slice we are interested in extracting. The technique of slicing is demonstrated in the program below.

```

# A program to demonstrate list slicing
fruits = ['apple', 'pear', 'orange', 'banana', 'kiwi']

print(fruits[1:3]) # ['pear', 'orange']
print(fruits[2:4]) # ['orange', 'banana']
print(fruits[2:5]) # ['orange', 'banana', 'kiwi']

print(fruits[1:]) # ['pear', 'orange', 'banana', 'kiwi']
print(fruits[:5]) # ['apple', 'pear', 'orange', 'banana', 'kiwi']
  
```



Can you figure out how the program generates the output shown inside the comments?
How could we ‘slice out’ the fruits that grow in Ireland?

The slice is taken from the start position up to, *but not including*, the last position. For example, the slice created by `fruits[2:5]` starts at position 2 and continues up to, but not including, position 5 i.e. from position 2 to 4 inclusive.

If the first position is not specified it is taken to be zero, and if the last position is not specified it is taken to be the length of the list.

It is worth emphasising the point that a slice is in fact a new list.

For example, the following code results in the new list being stored in the list variable `exoticFruits`. The slice is from element 2 to 4 inclusive.

```
fruits = ['apple', 'pear', 'orange', 'banana', 'kiwi']  
exoticFruits = fruits[2:5]  
print(exoticFruits)
```



When this program is run the contents of `exoticFruits` are displayed i.e.

`['orange', 'banana', 'kiwi']`

Finally, it is also worth noting that Python does not display a syntax error when the end position specified in the slice exceeds the number of elements in the list. In such situations Python just slices up to the last element of the list. Therefore, the result of the following code will be the exact same as the code shown above.

```
fruits = ['apple', 'pear', 'orange', 'banana', 'kiwi']  
exoticFruits = fruits[2:15]  
print(exoticFruits)
```



KEY POINT: In general, the expression, `aList[startPos:endPos]`


creates a new list or *slice* from the list identified by `aList`.

- ✓ The resulting slice starts from index position `startPos` in `aList`
- ✓ The resulting slice runs up to index position `endPos-1` in `aList`
- ✓ If `startPos` is missing it is taken to be zero
- ✓ If `endPos` is missing it is taken to be `len(aList)-1`



Can you find and suggest fixes for the two (not three) syntax errors contained in the code below?

```
# Initialise two lists
1. suits = ['Hearts', 'Diamonds', 'Spades', 'Clubs']
2. cardFaces = ['Ace', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'Jack', 'Queen', 'King']
3.
4. print(cardFaces[1:10])
5.
6. colourCards = cardFaces[10:23]
7. print(colourCards)
8.
9. redSuits = suits[:2]
10. blackSuits = [2:]
11.
12. print(pinStripSuits)
13. print(redSuits)
14. print(blackSuits)
```



Syntax Error 1:

Syntax Error 2:

How does this extend your thinking in relation to range errors and slicing ?

Explain the steps you would need to take in order to extend the program's functionality to generate and display a random card? (e.g. Ace of Spaces)

What additional issues would you have to consider if the program was required to deal a random hand of five cards?

List Methods

As data structures go, lists are *very* flexible. Apart from the basic common operations that can be carried out on all sequences (e.g. concatenation, indexing, slicing), list objects also support a variety of additional type-specific commands (methods).

The table below introduces some of these, but a more complete reference can be found by browsing to the official Python page: <https://docs.python.org/3/library/stdtypes.html#mutable-sequence-types>

Method Name	Description
<code>aList.append(item)</code>	Appends the <code>item</code> to the list named <code>aList</code>
<code>aList.count(item)</code>	Returns an integer count of how many times the element <code>item</code> occurs in <code>aList</code>
<code>aList.extend(anotherList)</code>	Appends the list contained in <code>anotherList</code> to <code>aList</code>
<code>aList.index(item)</code>	Returns the index of the first occurrence of <code>item</code> in <code>aList</code> . A <code>ValueError</code> exception is raised if <code>item</code> is not found.
<code>aList.insert(index, item)</code>	Inserts <code>item</code> into <code>aList</code> at offset <code>index</code>
<code>aList.pop()</code>	Removes and returns the last element from <code>aList</code>
<code>aList.remove(item)</code>	Removes <code>item</code> from the list
<code>aList.reverse()</code>	Reverses the order of all the items of <code>aList</code>
<code>aList.sort()</code>	Sorts objects of <code>aList</code> <i>in place</i> i.e. without creating a new list. A new sorted list can be created using the <code>sorted</code> built in command

The `del` keyword can also be applied to remove individual elements or entire slices from a lists. `del` is used in conjunction with the square bracket index operator as follows.

<code>del aList[i]</code>	Removes element at position <code>i</code> from <code>aList</code>
<code>del aList[i:j]</code>	Removes all elements from position <code>i</code> up to, but not including, position <code>j</code> in <code>aList</code> . (Same as <code>aList[i:j] = []</code>)
<code>del aList[:]</code>	Removes (clears) all the elements from <code>aList</code>

List Methods - Example 1

The following program demonstrates the use of several of these type-specific commands. The generated output is shown as a comment at the end of each print statement. You should read through the code and try to understand the list methods used.

```
fruits = ['pear', 'apple', 'orange', 'banana', 'kiwi']
fruit = 'apple'
vegs = ['peas', 'carrots']

fruits.append(fruit)
print(fruits) # ['pear', 'apple', 'orange', 'banana', 'kiwi', 'apple']

fruits.extend(vegs)
print(fruits) # ['pear', 'apple', 'orange', 'banana', 'kiwi', 'apple', 'peas', 'carrots']

fruits.insert(2, fruit)
print(fruits) # ['pear', 'apple', 'apple', 'orange', 'banana', 'kiwi', 'apple', 'peas', 'carrots']

fruits.pop()
print(fruits) # ['pear', 'apple', 'apple', 'orange', 'banana', 'kiwi', 'apple', 'peas']

fruits.remove(fruit)
print(fruits) # ['pear', 'apple', 'orange', 'banana', 'kiwi', 'apple', 'peas']

fruits.reverse()
print(fruits) # ['peas', 'apple', 'kiwi', 'banana', 'orange', 'apple', 'pear']

fruits.sort()
print(fruits) # ['apple', 'apple', 'banana', 'kiwi', 'orange', 'pear', 'peas']

print(fruits.index(fruit)) # 0
print(fruits.count(fruit)) # 2
```



Notice that the same dot notation as was described for string methods is also used for lists. This means in order to use a list method, the programmer must type the name of the list variable followed by a dot, followed by the method's name, i.e.

<list-variable-name>.<method-name>

List Methods - Example 2

Try the following.

```
# Sample program to build up a list of user details
userList = []
userName = input("Enter your name: ")
userList.append(userName)
userAge = int(input("What age are you "+userName+"?"))
userList.append(userAge)
userCountry = input("What is your country of birth?")
userList.append(userCountry)

print("My database for "+userName+"\n"+str(userList))
```



Two More Strings Methods (`split` and `splitlines`)

Another way to create lists is by using either of the string methods – `split` or `splitlines`.

These two methods (commands) are very similar to one another in the sense that they both break a string into separate ‘tokens’. Both methods create a new list with each token becoming a separate list element.

Take for example the following code.

```

chomskyStr = "Colourless green ideas sleep furiously"
aList = chomskyStr.split()
print(aList)

```



- The first line initialises the variable `chomskyStr`
 [Aside: *Colourless green ideas sleep furiously* is a sentence composed by Noam Chomsky, (American linguist, philosopher and cognitive scientist), as an example of a sentence that is grammatically correct, but semantically nonsensical.]
- Line 2 creates a new list called `aList`. The string is tokenised and each word becomes a separate list element as shown by the output:

```
['Colourless', 'green', 'ideas', 'sleep', 'furiously']
```

The main difference between `split` and `splitlines` is exemplified by the program below. (Notice the use of triple quotes to create the block string which spans multiple lines.)

```

seussStr = """I do not like green eggs and ham.
I do not like them Sam-I-am.
I do not like them here or there.
I do not like them anywhere.
I do not like them in a house
I do not like them with a mouse"""
bList = seussStr.splitlines()
print(bList)

```



The string comes from a poem *Green Eggs and Ham* written by Dr Seuss, American author, political cartoonist and poet

The program creates a new list, `bList` - each line is a separate element as shown.

```
['I do not like green eggs and ham.', 'I do not like them Sam-I-am.', 'I do not like them here or there.', 'I do not like them anywhere.', 'I do not like them in a house', 'I do not like them with a mouse']
```

Example Program (fruit machine v2)

The program below simulates a fruit machine. The sample outputs shown here should give a good idea of what the program does.

Cherry
Strawberry
Melon

Banana
Pineapple
Cherry

Melon
Pineapple
Pineapple

Strawberry
Banana
Strawberry

Fruit machine simulator program

```
# Program to simulate a fruit machine!
import random

# Open the fruits file (already created)
fruitFile = open("fruits.txt", "r")

# Read the entire file
fileContents = fruitFile.read()

# Close the file
fruitFile.close()

# Split the content into a list
fruits = fileContents.split()

# Spin! Display three fruits
print(random.choice(fruits))
print(random.choice(fruits))
print(random.choice(fruits))
```



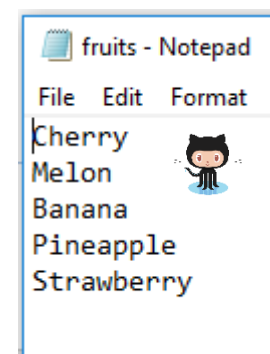
STUDENT TIP

Teachers should encourage students to look for recurring patterns in code.

Once such pattern is the use of the `split` and `splitlines` methods to put some structure on data such as the contents of a file.

The program works by reading the contents of a file called `fruits.txt` (shown here to the right) into a string called `fileContents`. The string is then `split` into individual tokens each of which are stored as separate elements of a list called `fruits`.

Finally, the `choice` command from the `random` library is used three times. Each time it picks a random element from the `fruits` list and displays it.



The purpose of the program is to re-inforce some of the concepts and techniques covered in this section and also to prepare for the following breakout session.



BREAKOUT ACTIVITIES

BREAKOUT 4.1: Random Sentence Generator²

For this exercise you are required to write program that generates a random sentence.

The program will make use of four different lists of strings called `articles`, `nouns`, `verbs` and `prepositions`. The syntax rule for a valid sentence is that it must have the following structure:

```
article noun verb article noun preposition verb
```

A sample list of words for each of the four lists is given:

<code>articles</code>	"the", "a", "one", "some", "any"
<code>nouns</code>	"teacher", "student", "principal", "library", "school"
<code>verbs</code>	"taught", "learned", "read", "walked", "ran"
<code>prepositions</code>	"to", "from", "over", "under", "on"

Valid sentences must be syntactically correct but may not make any sense. For example,

- *the teacher taught a student to read* makes sense while,
- *some school walked under a principal* is semantically nonsensical!

Hints:

1. The solution will need to create a list to store the words of the sentence e.g. `wordlist`. Initially this list will be empty.
2. The words will need to be selected at random from each list. The following line of code may be adapted to generate a random word.

```
print(random.choice(nouns))
```

3. As words are randomly selected they can be appended to `wordlist`
4. Once the seven words have been generated they can be displayed in a sentence.

² Adapted from a problem in Java: how to program, P.J. Deitel, H.M. Deitel, 9th ed., Prentice Hall, 2012

BREAKOUT 4.2: Data Processing (heights)

At the end of Section 2 we completed a breakout activity based on processing a number of height values input by the user. We start this activity by presenting a solution to the activity.

```
# A program to calculate the average height of 5 people
# The heights are stored in a file called 'heights.txt'
heightFile = open("heights.txt","r") # Open the file

totalHeight = 0 # Initialise a running total to zero
height = float(heightFile.readline()) # read the first value
totalHeight = totalHeight + height # keep a running total

height = float(heightFile.readline()) # read the next value
totalHeight = totalHeight + height # keep a running total

height = float(heightFile.readline()) # read the next value
totalHeight = totalHeight + height # keep a running total

height = float(heightFile.readline()) # read the next value
totalHeight = totalHeight + height # keep a running total

height = float(heightFile.readline()) # read the next value
totalHeight = totalHeight + height # keep a running total

# Calculate the average
avgHeight = totalHeight/5

# Display the result
print("The average height is "+str(round(avgHeight,2))+ "cm")
print("The average height is", round(avgHeight*0.393701,2), "inches")

heightFile.close()
```



```
heights -
File Edit |
150
160
180
180
190|
```

The contents of the heights.txt file are shown here to the right. These can be edited and a new set of values supplied to the program at runtime.

We now add the following program into the mix. This three-liner uses the `mean` command from the `statistics` library to calculate the arithmetic mean of the numbers contained in the list `heightList`. If we wanted to calculate the mean of a different set of heights the programmer would need to change the values in this list.

```
from statistics import mean
heightList = [150, 160, 180, 180, 190]
print(mean(heightList))
```



It is worth browsing to <https://docs.python.org/3/library/statistics.html> to take a look at the official Python documentation on the statistics library.

Suggested Activities

1. Compare and contrast the two programs



2. Devise a test plan (similar to that shown in the corresponding activity at the end of Section 2) for the programs. Execute the test plan.
3. Modify the first program so that it uses the `statistics` package to calculate the mean. (The variable `totalHeight` should not appear in the final program.). Repeat the test plan to make sure your solution works.
4. Continue to modify this program with the following changes:
 - instead of there being five separate uses of the `readline` command, the entire file should be read into a string by single `read` command.
 - use the `split` command to create a list of height values from the resulting stringYou may wish to look back at the fruit machine v2 example (page 112) for some technical guidance to complete this activity.

The program should be able to deal with a variable number of heights and as well as decimal values.

5. Display the following information in meaningful messages (preferably using string formatting to display any variable data):
 - The maximum height
 - The minimum height
 - The range and interquartile range
 - The heights in ascending/descending order

BREAKOUT 4.3: Turtle Directions

The purpose of this activity is to build upon the knowledge we already have about turtle graphics and apply the concepts from this section to them.

The turtle graphic program shown below contains two lists – angles and distances. Between them, the lists contain ‘encoded’ data that direct a turtle from point A to point B. The program listing is as follows:

```
from turtle import *

pensize(2) # set the pen size to 2
color("red") # sent the pen colour to red

# setup a list of angles
angles = [0, 90, 60, 90, 90]
# setup a list of distances
distances = [100, 75, 50, 75, 100]

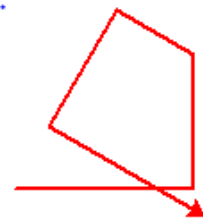
left(angles[0])
forward(distances[0])
left(angles[1])
forward(distances[1])
left(angles[2])
forward(distances[2])
left(angles[3])
forward(distances[3])
left(angles[4])
forward(distances[4])
```



When the program is run it displays the path show.

Experiment by changing the values of the angles and distances to see what journeys you can come up with.

(Remember that the turtle starts in the centre of the screen facing right (east). The arrowhead indicates the end of the path.)



Section 5

Programming Logic

Introduction

Thus far, we have been dealing with *sequential* programs i.e. programs which begin their execution at the first line and execute each line in order until the last line is reached.

In addition to sequence, Python supports two other control structures known as *selection* and *iteration*. The purpose of this section is to explain the syntax and semantics of selection and iteration, and explore some common programming techniques used to apply them in real-world contexts.

Selection

Selection structures are commonly referred to as *decisions*. These structures provide programmers with a branching mechanism whereby, certain blocks of code may be either executed or skipped at runtime. The decision of which block of code to select for execution depends on the result of a *condition* also known as a *Boolean expression*.

The main Python keywords used to support decision structures are `if`, `else` and `elif`.

Iteration

Iteration structures are commonly referred to as *loops*. Loops are used to cause the same block of code to be executed multiple times. At runtime, the code inside a loop (the *loop body*) is executed repeatedly as long as some condition (the *loop guard*) is met. The loop guard is also a *Boolean expression*.

The main Python keywords used to support iteration structures are `for`, and `while`.

Three other (less important) keywords that relate to loops are `break`, `continue` and `pass`.



KEY POINT: Selection and iteration are two programming techniques whose runtime operations are based on the outcome of Boolean expressions.

Boolean Expressions

A Boolean expression is any expression that evaluates to either `True` or `False`.

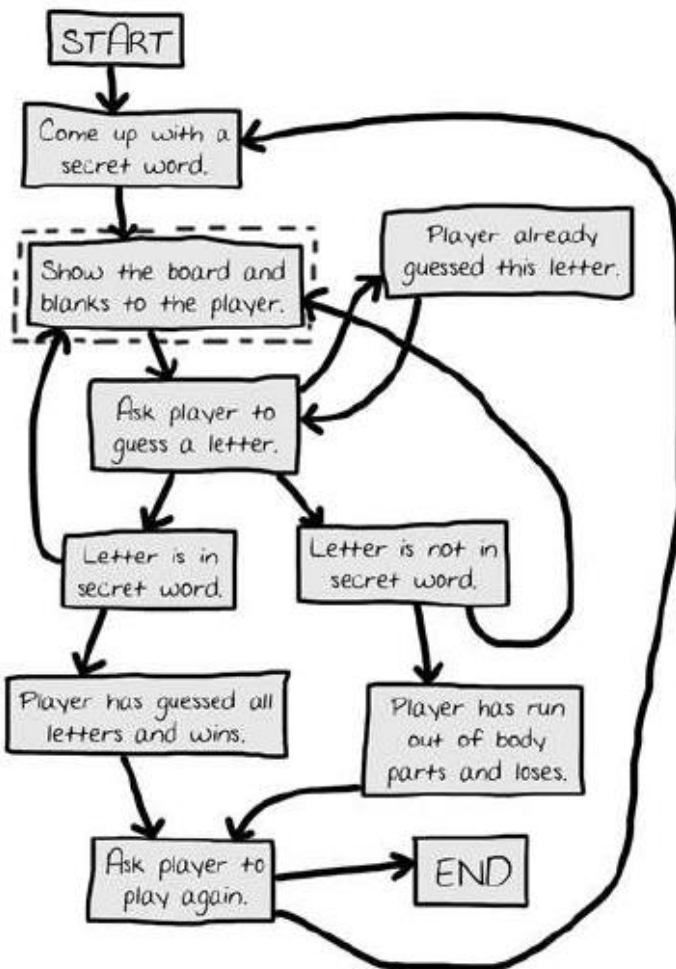
Boolean expressions form the basis of all **programming logic**.



Hangman!

Hangman is a well-known guessing game usually played by two people using pencil and paper. One player thinks of a word and the other tries to guess it by suggesting letters within a certain number of guesses.

The illustration³ below depicts the main steps of the game and the graphic⁴ to the right illustrates a sample run of the game, where the player is trying to guess the word *hangman*.



0		Word: hangman Guess: E Misses:
1		Word: _ _ _ _ _ Guess: T Misses: e
2		Word: _ _ _ _ _ Guess: A Misses: e,t
3		Word: _ A _ _ _ A _ Guess: O Misses: e,t
4		Word: _ A _ _ _ A _ Guess: I Misses: e,o,t
5		Word: _ A _ _ _ A _ Guess: S Misses: e,i,o,t
6		Word: _ A _ _ _ A _ Guess: N Misses: e,i,o,s,t
7		Word: _ A N _ _ A N Guess: R Misses: e,i,o,s,t
8		Word: _ A N _ _ A N Guess: Misses: e,i,o,r,s,t

³ http://calab.hanyang.ac.kr/courses/ISD_taesoo/05_Hangman.pdf

⁴ [https://en.wikipedia.org/wiki/Hangman_\(game\)](https://en.wikipedia.org/wiki/Hangman_(game))

Boolean Expressions

Boolean Logic was invented by the mathematician George Boole, 1815-1864 who was the first professor of Mathematics at University College Cork (UCC). The algebra on which Boolean logic is based is used extensively to build electronic circuits and write computer programs. Boolean logic, therefore, forms the basis of all modern digital devices and software systems.

Boolean expressions are to Boolean algebra, what algebraic expressions are to algebra, and arithmetic expressions are to arithmetic. At any given moment in time, a Boolean expression will evaluate to either `True` or `False`. It can never be anything in between.

Boolean expressions are so important that it could be argued that the secret to good programming lies in the formation of good Boolean expressions. This is the responsibility of the programmer.



KEY POINT: All Boolean expressions evaluate to one of two values - `True` or `False`.

`True` and `False` are two Python keywords which technically behave as if they were the numbers 1 and 0.

Simple Boolean Expressions

A *simple Boolean expression* is one that uses a single relational operator (e.g. greater than, less than or equal to etc.) to compare (i.e. relate) two values.

For example, $7 > 3$ (seven greater than three) is a simple Boolean expression that compares the numbers 7 and 3 under the relation of 'greater than'. It evaluates to `True` because 7 is a bigger number than 3. On the other hand, the expression $7 < 3$ evaluates to `False`, because seven is not less than three.

STUDENT TIP

Teachers should provide students with frequent opportunities to extend their technical vocabulary. For example, the terms *Boolean expression* and *condition* can be used interchangeably.

Simple Boolean expressions (as created by the basic relational operators) are the basic building blocks used to implement decisions and loops in Python.

Python supports the six relational operators given below.

Operator	Description	Example	Result
>	Greater than	7 > 5	True
>=	Greater than or equal to	7 >= 5	True
<	Less than	7 < 5	False
<=	Less than or equal to	7 <= 5	False
==	Equal to (the same as)	7 == 5	False
!=	Not equal to (not the same as)	7 != 5	True

Python relational operators

Relational operators are *binary operators* because they need two operands in order to work. Although in practice operands are usually numeric, operands can be of any datatype that results from a Python expression. String operands, for example, can be compared using lexicographic ordering of their constituent characters.

Some more examples of simple Boolean expressions (aka conditions) are presented below. (Note: $x = 1$, $y = 0$ and $z = -1$.)

Condition	Result	Condition	Result
6 >= 5	True	5 > x	True
0 > 1	False	x > y	True
1 < 0	False	x <= y	False
1 == 0	False	y <= 0	True
4 == 4	True	z > y	False
4 <= 4	True	x == z	False
3 != 4	True	0 == y	True
3 <= 4	True	x != y	True

Compound Boolean Expressions

Compound Boolean expressions are formed by connecting simple Boolean expressions together using any of the three Python Boolean operators - `and`, `or`, and `not`.



KEY POINT:

Boolean operators can only operate on Boolean values i.e. `True/False`.

Just like simple Boolean expressions, compound Boolean expressions always evaluate to either `True` or `False`.

The combinations of values for inputs and their corresponding outputs for `and`, `or`, and `not` can be conveniently represented in a tabular format known as a *truth table*.

`not` is the simplest of the three Python Boolean operators. It is a *unary operator* meaning that it can only work on one operand at a time. The truth table showing the relationship between some proposition `A` and `not A` is shown below.

A	not A
False	True
True	False

Both `and`, and `or` are binary operators meaning that they require two operands to work. The truth tables for `and`, and `or` are shown below.

A	B	A and B
False	False	False
False	True	False
True	False	False
True	True	True

A	B	A or B
False	False	False
False	True	True
True	False	True
True	True	True

The (binary) inputs are given by the columns `A` and `B` and the output for these inputs is shown in the rightmost column.

Examples

For the purpose of the examples shown below we will assume that we have a number of variables assigned as, $x = 1$, $y = 0$, $z = -1$ and $valid = True$, $finished = False$

Condition	Result
$x == 1$ and $y == 0$	True
$x == y$ or $z == -1$	True
$x != y$ and $y != -z$	True
not valid	False
$z \leq y$ and finished	False
$z > y$ or valid	True
finished and not valid	False
not finished or not valid	True



Evaluate the following Boolean expressions

True and False

not True or False

True and not True

not True or not False

not True and not False

The Guessing Game

The remainder of this section will focus on the concepts of selection, iteration and programming logic. A basic ‘guess the number’ game is used as a platform on which to develop ideas and techniques associated with these concepts. As new concepts are introduced they are exemplified by incorporating them so that they add functionality into the game program. The result is seven versions as follows.

Guess Game v1: This is a basic guess game. The base program generates a random number which the user is asked to guess. If the user guess is correct the program displays an appropriate message.

Guess Game v2: This time the program displays a message informing the user that they were either correct or incorrect based on the value entered.

Guess Game v3: In this version of the game the user is provided with more detailed feedback about their guess i.e. correct, too high or too low.

Guess Game v4: This is the same as version 3 except that the user is given at most three chances to guess the correct number. If the user guess correctly within the three allocated chances the program terminates.

Guess Game v5: In this version of the game, the program continues until the user makes the correct guess – a subtle but important and powerful enhancement on the previous version. Each time the user enters a guess the program continues to display one of the three messages, i.e. correct, too high or too low.

Guess Game v6: A refinement on the previous version whereby after guessing correctly, the user is offered the opportunity to play the game again. If the user enters “N” for no the game exits. Otherwise the program generates a new random number and the game starts again.

Guess Game v7: In this final version of the game, the functionality of the game is the exact same as version 6. However, this version validates any data entered by the user i.e. it checks that the guess, is in fact, a number before proceeding.

Selection (`if`, `else`, `elif`)

Selection statements are used by programmers to build alternative execution paths through their code. As already stated they are commonly referred to as decision statements.

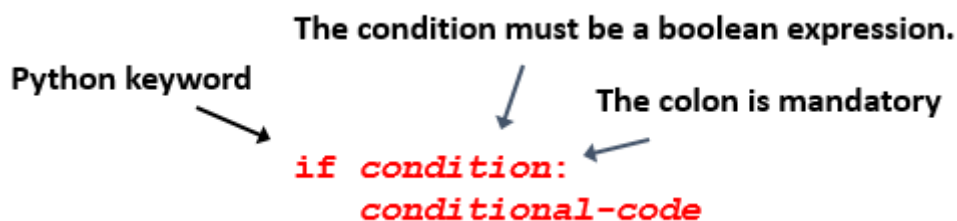
Python provides built in supports for three different kinds of selection statements:

- single option (the basic `if` statement)
- double option (the `if-else` statement)
- multiple option (the `if-elif-else` statement)

When a running program executes a selection statement, it evaluates a condition, and based on the result of this evaluation it will decide which statement(s) to execute next.

The basic `if` statement

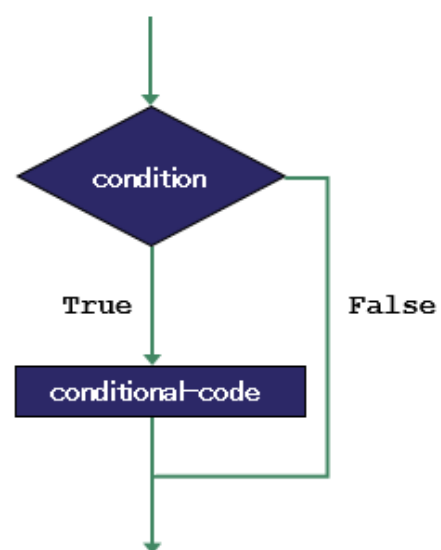
The syntax and semantics of Python's single option if-statement are illustrated and described below.



If Python evaluates the condition to `True`, then the conditional code inside the `if` statement will be executed.

If the condition evaluates to `False`, then the conditional code is skipped and execution continues from the next line of code after the `if`-statement.

Note the use of the colon at the end of the line and also the fact that the *conditional code must be indented*.




Flow chart illustration of `if`-statement

Example (Guess game v1)

The program below generates a random number between 1 and 10 (`number`) and prompts to user to guess the number (`guess`).

```

1. # A program to demonstrate the single if statement
2. import random
3.
4. number = random.randint(1, 10)
5. # print(number)
6.
7. guess = int(input("Enter a number between 1 and 10: "))
8.
9. # Evaluate the condition
10. if guess == number:
11.     print("Your guess was correct")
12.     print("Well done!")
13.
14. print("Goodbye")
  
```



Guessing Game v1

(Uncommenting line 5 will help you test this program faster)

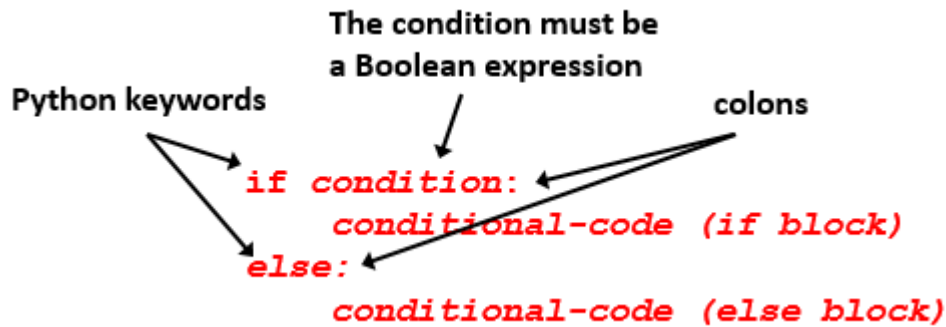
- The `if` statement on line 10 evaluates the condition **`guess == number`**
- The execution of lines 11 and 12 are conditional upon the result of this evaluation. Notice the indentation of these lines. They will be executed only if the condition evaluates to `True`
 - The condition will evaluate to `True` if the guess entered by the user is the same as the computer's generated number
 - The condition will evaluate to `False` if the guess entered by the number is *not* the same as the computer's generated number
- The last line is always executed (unconditionally)

STUDENT TIP

Teachers should encourage students to use 'additional' `print` statements in their programs to 'peek' at values that will not be visible in the final program.

The `if-else` statement

The syntax and semantics of Python's double option if-statement are illustrated and described below.

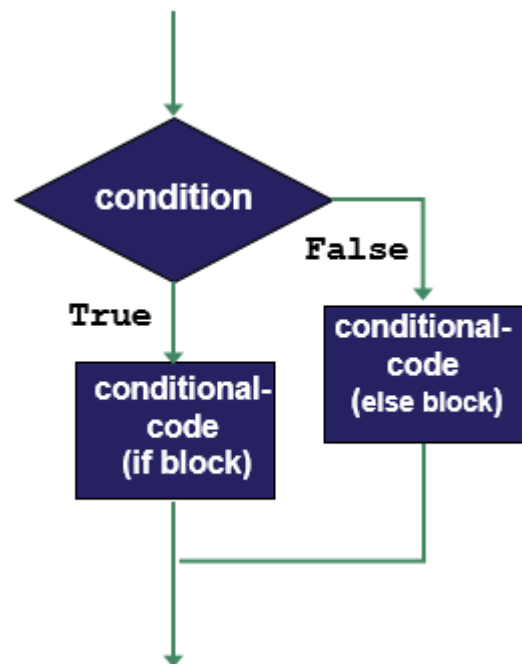


If Python evaluates the condition to `True`, the block of code associated with the `if`-statement (i.e. the `if`-block) is executed.

Otherwise, the block of code associated with the `else` statement (i.e. the `else`-code block) is executed.

In other words, the `else` block is executed only when the condition is evaluated by Python to `False`.

Once either block has been executed the flow of control continues at the next line immediately following the `else`-block



Flow chart illustration of `if-else`-statement



KEY POINT:

All conditional code must be indented to the same level by the programmer.


Finally, it is important to recognise that the two blocks are mutually exclusive. In any given run of the program either one block or the other will be executed, but never both.

Example (Guess game v2)

This example extends guessing game v1 by displaying some messages to the user if they guess the wrong number.

```

1. # A program to demonstrate the double if statement
2. import random
3.
4. number = random.randint(1, 10)
5. print(number) # comment this line out later!
6.
7. guess = int(input("Enter a number between 1 and 10: "))
8.
9. # Evaluate the condition
10. if guess == number:
11.     print("Your guess was correct")
12.     print("Well done!")
13.     print(" ..... play again soon!")
14. else:
15.     print("Hard luck!")
16.     print("Incorrect guess")
17.     print(" ..... play again soon!")
18.
19. print("Goodbye")
  
```



Guessing Game v2

- The condition **guess == number** on line 10 is pivotal here again
 - Lines 11 – 13 are selected for execution if the condition evaluates to `True`
 - Lines 15 – 17 are selected for execution if the condition evaluates to `False`

STUDENT TIP

A common student misconception is that both branches of an `if-else` statement are *always* executed. Teachers should encourage students to use test data that will activate each branch in separate runs of the program.

- Python will always execute the last line of the above program as it is not part of the `if-else` statement



Compare the logic of the two code snippets below. What do you notice?

```
# Evaluate the condition
if guess != number:
    print("Hard luck!")
    print("Incorrect guess")
else:
    print("Your guess was correct")
    print("Well done!")

print(" ..... play again soon!")
print("Goodbye")
```

```
# Evaluate the condition
if guess == number:
    print("Your guess was correct")
    print("Well done!")
    print(" ..... play again soon!")
else:
    print("Hard luck!")
    print("Incorrect guess")
    print(" ..... play again soon!")

print("Goodbye")
```



Programming Exercise 5.1

Re order the individual lines of code shown below into a program that:

- generates two random numbers between 0 and 12**
- calculates their product**
- prompts the user to enter the product of the two numbers**
- displays an appropriate response to the user's attempt**

```
print("Goodbye")
print("The correct answer was %d" %(n1*n2))
n2 = random.randint(0, 12)
else:
print("%d * %d" %(n1, n2))
print("Incorrect!")
ans = int(input("Enter your answer: "))

if ans == n1*n2:
    ans = n1*n2
if ans = n1*n2:
import random
n1 = random.randint(0, 12)
print("Correct!")
n1*n2 = ans
```

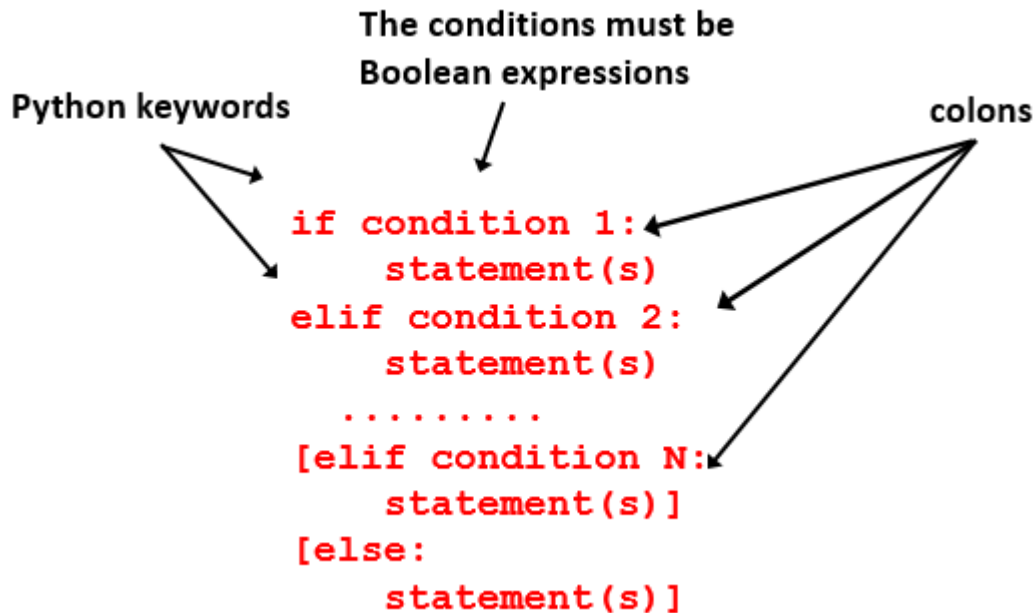
Note - three of the lines are surplus to requirements.

STUDENT TIP

It is a very common pitfall to use the single equals (=) assignment operator unintentionally instead of the double equals (==) relational operator

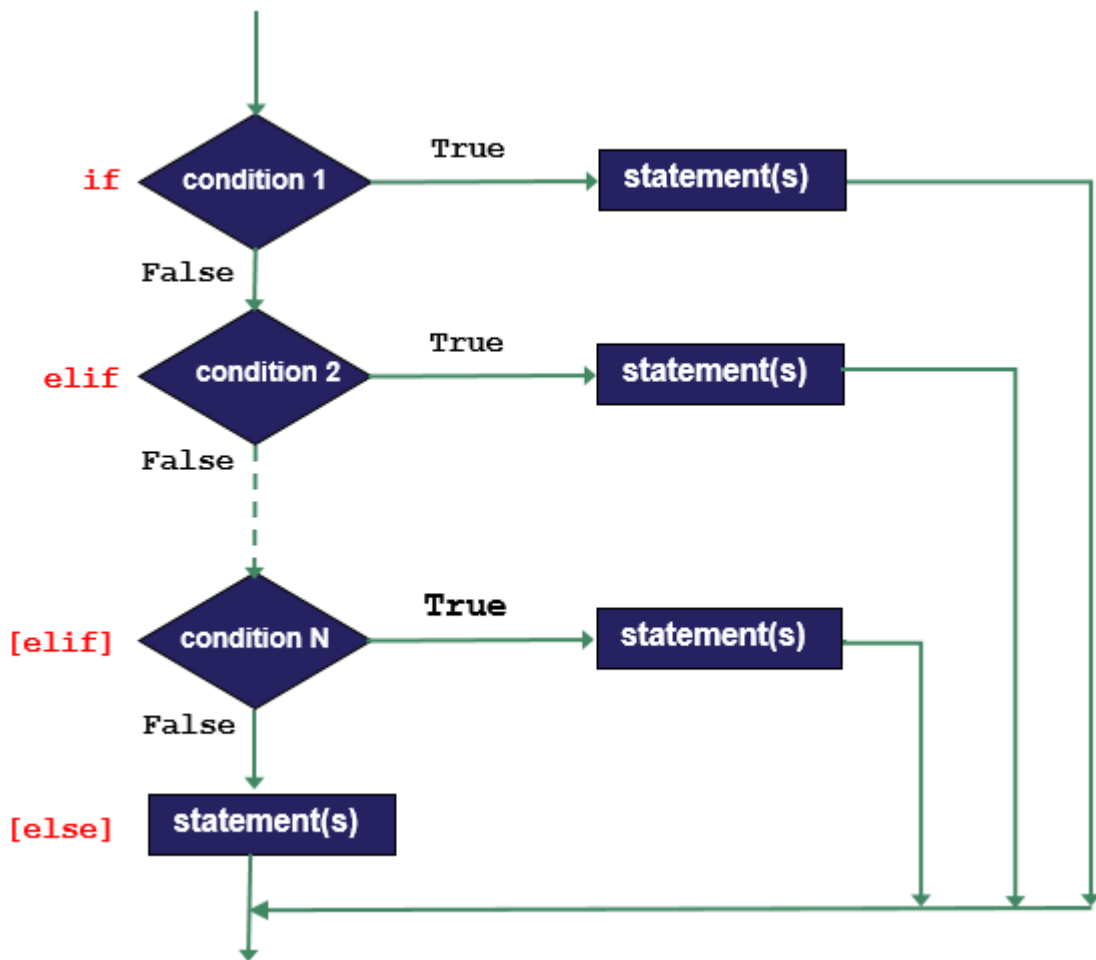
The `if-elif-[else]` statement

The syntax and semantics of Python's multiple option if-statement are illustrated and described below.



- The first condition is always inside an `if` statement
- There can be as many `elif` statements as required
- Each `elif` statement must include a condition
- The use of a final `else` statement is optional (indicated by square brackets)
- The `if`, `elif` and `else` keywords must all be at the same level of indentation.
- A colon must be used at the end of any lines containing `if`, `elif` and `else`
- Each condition is evaluated in sequence. Should Python evaluate a condition to `True` then the associated statement(s) are executed and the flow of control continues from the next line following the end of the entire `if-elif` statement. If none of the conditions are found to be `True`, then Python executes any statement(s) associated with the `else`.

The logic of an `if-elif-[else]` statement is illustrated using the flowchart below.



Flow chart illustration of if-elif-[else]-statement



Use the space below to reflect on what you have learned about Python's three types of selection statements.

Example (Guess game v3)

This example enhances guessing game v2 by displaying more helpful messages to the user when they make an incorrect guess.

```

1. # A program to demonstrate the multiple if statement
2. import random
3.
4. number = random.randint(1, 10)
5. print(number) # comment this line out later!
6.
7. guess = int(input("Enter a number between 1 and 10: "))
8.
9. # Evaluate the condition
10. if guess == number:
11.     print("Correct")
12.     print("Well done!")
13. elif guess < number:
14.     print("Hard luck!")
15.     print("Too low")
16. else:
17.     print("Hard luck!")
18.     print("Too high")
19.
20. print("Goodbye")

```



Guessing Game v3

Think about it – when you ask someone to guess a number between one and ten there are exactly three possible outcomes. The guess can be

- the same as the number you are thinking of
- lower than the number you are thinking of
- higher than the number you are thinking of



KEY POINT: The multiple option if statement should be used to model situations from the real world where there are multiple possible outcomes that require separate specific processing. In this example, that processing is a message tailored to the user's response.

- In any given run of the above example Python will execute either lines 11 and 12, or 14 and 15, or 17 and 18. These lines are mutually exclusive.
- As was the case with the earlier versions of this program Python will always execute the last line of the above program as it is not part of the `if-elif-else` statement



Programming Exercises 5.1:

Fill in the blanks below without altering the logic of the example program on the previous page. Log your thoughts as you proceed.

```

if guess > number:
    [Redacted]
elif guess == number:
    [Redacted]
else:
    [Redacted]
    
```

```

if number [Redacted] guess:
    print("Hard luck!")
    print("Too low")
[Redacted]
else:
    [Redacted]
    
```



Outline any considerations that would have to be made by a programmer to avoid having to duplicate lines 14 and 17 in the example program.



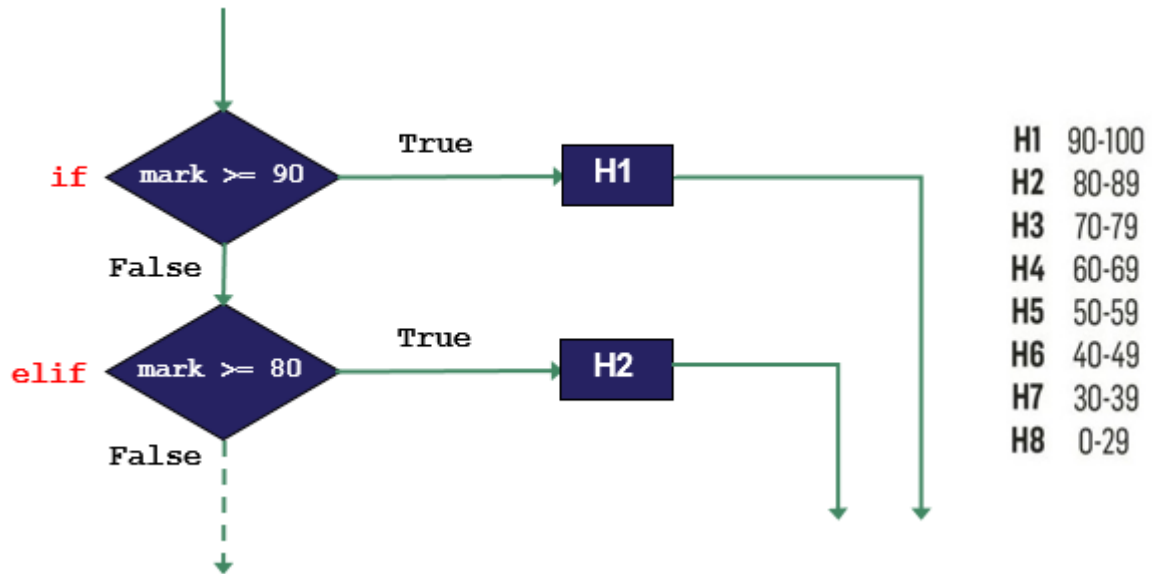
In what way(s) did either/both of the above tasks promote metacognition?



BREAKOUT ACTIVITIES (selection)

BREAKOUT 5.1: Task Development

Use the space provided to complete the flowchart shown below.⁵



⁵ See <https://github.com/pdst-lccs/lccs-python/blob/master/Section%205%20-%20Programming%20Logic/Breakouts/Breakout%205.1%20-%20Task%20Development/README.txt> for some ideas on how to develop this task in the classroom.

Iteration (`for` and `while` loops)

Iteration is a programming technique that allows programs to execute statements multiple times. Python provides built in supports for two different kinds of iteration statements:

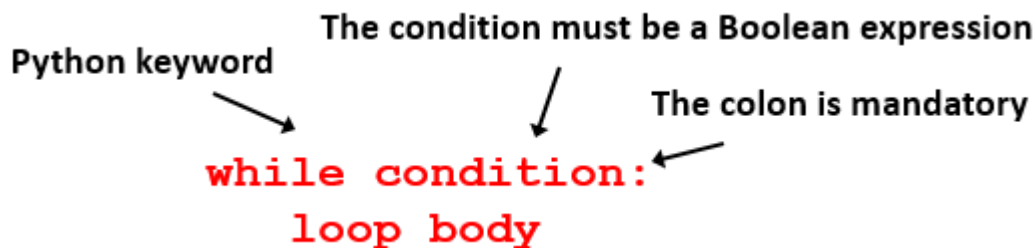
- the `while` loop
- the `for` loop

We now consider these in turn.

The `while` loop

This is Python's most general (and therefore) flexible loop construct.

The syntax and semantics of Python's `while` loop are illustrated and described below.

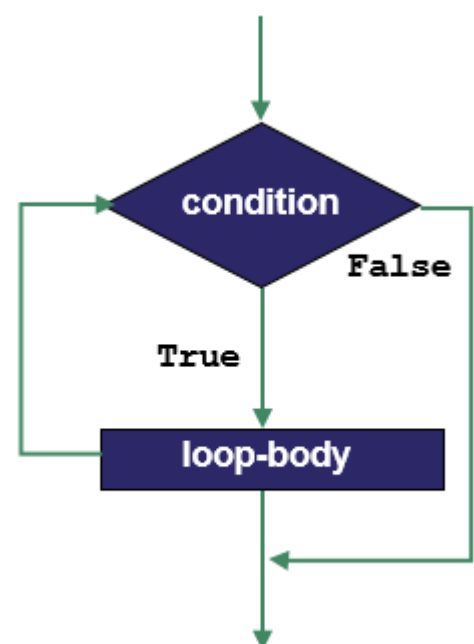


In Python, `while` loops are introduced with the keyword `while`. This is followed by some condition which has to be made up by the programmer (this is the 'hard' part!).

If the result of the condition is `True`, the statement(s) that make up the *loop body* are executed. These statements *must* be indented.

When Python reaches the last line of the loop body the flow of control loops back to the condition which is evaluated again. (Python will know the last line of the loop body from the levels of indentation.)

The above process continues until the result of the condition is found to be `False`.



Flow chart illustration of `while` loop

When (and if) the condition is `False` Python skips the loop body, and the flow of control resumes at the next statement following the loop body.

It should be noted that it is the programmer's responsibility to ensure that the loop body contains a line of code that will cause the loop condition to eventually become `False`. Otherwise, the loop will never terminate. Such loops are called *infinite loops*.

It is also worth noting out that the loop body might not ever be executed. This situation would arise when the condition evaluates to `False` before the first iteration. If this happens the loop body is skipped and the flow of control continues from the first statement after the loop body.

Because the condition 'guards' entry into the loop, it is referred to as the *loop guard*.



KEY POINT: The **loop body** is executed each time the **loop guard** is evaluated to `True`.

STUDENT TIP


It is useful to think of a loop guard as a sentinel who operates a green-red signal system. A green signal means enter the loop and red means skip the loop.

Example

This short program shows the main features of a `while` loop.

```

1. # Simple while loop
2.
3. # Initialise a loop counter
4. counter = 1
5.
6. # Loop 10 times
7. while counter <= 10:
8.     print("Hello World") # Display a message
9.     counter = counter + 1 # Increment the counter
10.
11. # This line is only executed once
12. print("Goodbye")
  
```



Simple while loop demo.

The program displays the message *Hello World* ten times. The string *Goodbye* is displayed once before the program exits.

- The loop is introduced by the `while` keyword on line 7. Note the use of colon (`:`) at the end of this line.
- The condition `counter <= 10` is central how the loop operates. The loop will be executed as long as this condition remains `True`. The condition is initially `True` because the variable `counter` was initialised to 1 on line 4
- Lines 8 and 9 make up the loop body.
 - line 8 tells Python to display the string, *Hello World*
 - line 9 tells Python to increase the value of `counter` by 1 (recall running totals)
- The next line to be executed after line 9 is *always* line 7. (This is the iteration)
- Each time line 7 is executed the value of `counter` will have increased by one since the previous iteration. Eventually, `counter` will have reached a value of 11 and the condition will be found to be `False`. At this point the flow of control jumps beyond the loop body and line 12 is the next, and final, line of the program to be executed.

It is well worth investing some time in this example to make sure you understand exactly how while loops are executed at runtime.

If we define an iteration to be the number of times a loop body has been executed, we can use the technique of tracing to keep track of the loop's progress.

Initially, (before any iterations), `counter` is set to 1, there is no program output displayed, and the condition `counter <= 10` is `True`. After one iteration of the loop, `counter` has a value of 2, the string *Hello World* is displayed and the condition remains `True`.

Continue in this way until you complete the 'trace diagram' shown below.

Iteration #	counter	Program Output	counter <= 10
0	1		True
1	2	Hello World	True
2	3		
3			

Example (Guess game v4)


Let's say we wanted to enhance our guessing game to give the user *three chances*.

We start off in the knowledge that version 3 of the program works properly for one chance. Our enhancement can be achieved simply by 'wrapping' the code from version 3 inside a while loop that runs three times. A solution is presented below.

```

1. # Guess Game - 3 guesses
2. import random
3.
4. number = random.randint(1, 10)
5. print(number) # have a sneak peek!
6.
7. # Initialise a loop counter
8. counter = 0
9.
10. # Loop 3 times
11. while counter < 3:
12.
13.     guess = int(input("Enter a number between 1 and 10: "))
14.     if guess == number:
15.         print("Correct")
16.         break # exit the loop immediately!
17.     elif guess < number:
18.         print("Too low")
19.     else:
20.         print("Too high")
21.
22.     counter = counter + 1
23.
24. print("Goodbye")

```



Guessing Game v4

The technique of wrapping code inside a loop is very important in the development of computer programs and systems.

STUDENT TIP

When starting to learn loops for the first time students should be encouraged to work with code they are already familiar with and wrap it inside a loop structure.

Code wrapping is based on a notion that, if a piece of working code can be written using sequence/selection control structures only, then it should be relatively straightforward to put that code inside a loop.



Complete the ‘trace diagram’ shown below for Guessing Game v4

The diagram has been complete up to, but not including, the point when the user is about to enter a guess for the first time (i.e. the first execution of line 13).

The computer has generated a random number of 5 which has been recorded as `number`. The value has been displayed and `counter` has been initialised to zero. The condition `counter < 3` has been evaluated to `False` and this has also been recorded. You take over from this point.

Proceed by making up a value (i.e. guessing a number) and recording it in the first box underneath `guess`. Now trace the execution of line 14. This requires you (instead of Python) to evaluate the condition `guess == number`. Record your answer in the second black box.

Continue in this manner until the program has run to completion.

Iteration #	number	counter	counter < 3	guess	guess == number	Output
0	5	0	True	<input type="text"/>	<input type="text"/>	5



Experiment! Make some changes to the code. Try achieving the same logic using different conditions e.g. `counter <= 3` or `counter < 4`. How would these conditions affect the initial value of `counter`?


Example (Guess game v5)

In this version of the game we will introduce a Boolean variable to enable the program to continue until the user makes the correct guess (no matter how many guesses this may take!).

```

1. # Guess Game v5 - while not found
2. import random
3.
4. number = random.randint(1, 10)
5. print(number) # have a sneak peek!
6.
7. correct = False # initialise the loop guard variable
8.
9. # Loop until the variable correct becomes True
10. while not correct:
11.
12.     guess = int(input("Enter a number between 1 and 10: "))
13.     if guess == number:
14.         print("Correct")
15.         correct = True # this will cause the loop to exit
16.     elif guess < number:
17.         print("Too low")
18.     else:
19.         print("Too high")
20.
21. print("Goodbye")

```



Guessing Game v5

A Boolean variable is a variable used to store a Boolean value. In Python the only two Boolean values are `True` and `False`.

In this example, the name of the Boolean variable is `correct`. It is initialised to `False` on line 7. The use of the variable `correct` (lines 7, 10 and 15) is *the* central feature of this program.

The loop keeps going as long as the `correct` is `False`. Logically, this is the same as saying that the loop continues as long as `not correct` is `True`. The only place `correct` is set to `True` is on line 15 which gets executed if and only if the value of `guess` is the same as the value of `number`.



Complete the ‘trace diagram’ shown below for Guessing Game v5

The diagram has been completed up to but not including the first execution of line 12 i.e. the user is about to enter a guess for the first time.

The computer has generated a random number of 8 which has been recorded as `number`. The value has been displayed and the Boolean variable `correct` has been initialised to `False`. The condition `not correct` has been evaluated to `True` and this has also been recorded. You take over from this point.

Proceed by making up a value (i.e. guessing a number) and recording it in the first box underneath `guess`. Now trace the execution of line 13. This requires you (instead of Python) to evaluate the condition `guess == number`. Record your answer in the second black box.

Continue in this manner until the program has run to completion.

Iteration #	number	correct	not correct	guess	guess == number	Output
0	8	False	True	<input type="text"/>	<input type="text"/>	8



Take a moment to compare the loop guards used in versions 4 and 5. In what ways are they similar? How do they differ?

Example (Guess game v6)

In this version, we add one final piece of functionality. This time when the user guesses correctly, instead of terminating the loop (and program), this program will ask the user if they want to play another game. If the user responds with anything other than N (for no), the program generates a new random number and continues.

```

1. # Guess Game v6 - while not correct - ask, go again?
2. import random
3.
4. number = random.randint(1, 10)
5. print(number) # have a sneak peek!
6.
7. # Initialise the loop guard variable
8. keepGoing = True
9.
10. # Loop as long as keepGoing is True
11. while keepGoing:
12.
13.     guess = int(input("Enter a number between 1 and 10: "))
14.
15.     if guess == number:
16.         print("Correct")
17.         goAgain = input("Play again? (Y/N): ")
18.         if goAgain == "N":
19.             keepGoing = False
20.         else:
21.             # Generate a new number
22.             number = random.randint(1, 10)
23.             print(number) # why not?
24.
25.     elif guess < number:
26.         print("Too low")
27.
28.     else:
29.         print("Too high")
30.
31. print("Goodbye")

```



Guessing Game v6

This 'play again' logic is incorporated by using another Boolean variable, `keepGoing` which is initially set to `True` (line 8).

The loop will continue as long as the condition on line 11 evaluates to `True`. But the condition here is simply `keepGoing` so as long as this variable remains `True` the loop will continue.

The only circumstances where `keepGoing` is set to `False` is on line 19. Can you figure out from the code what these circumstances are?

Example (Guess game v7)

In this last version of the program there is no new functionality added. Rather, the program demonstrates a standard technique used to validate data entered by the user.

If you run any version of the guess game before this and enter a non-numeric value as the guess you will notice that the program crashes (runtime error). The reason for this is that all earlier versions make the (incorrect) assumption that a user will always enter the correct type of data, which is not very realistic for any production system.

```

1. # Guess Game v7 - while - go again? - data validation
2. import random
3.
4. number = random.randint(1, 10)
5.
6. # Initialise the loop guard variable
7. keepGoing = True
8.
9. # Loop as long as keepGoing is True
10. while keepGoing:
11.
12.     guess = input("Enter a number between 1 and 10: ")
13.     # Validate. Make sure the value is a number
14.     while not guess.isdigit():
15.         guess = input("Enter a number between 1 and 10: ")
16.
17.     # Conver the string to an integer
18.     guess = int(guess)
19.
20.     if guess == number:
21.         print("Correct")
22.         goAgain = input("Play again? (Y/N): ")
23.         if goAgain.upper() == "N":
24.             keepGoing = False
25.         else:
26.             # Generate a new number
27.             number = random.randint(1, 10)
28.
29.     elif guess < number:
30.         print("Too low")
31.
32.     else:
33.         print("Too high")
34.
35. print("Goodbye")

```



Guessing Game v7

Take some time to study the validation technique used here (lines 12 – 15) and see if you can figure out how it works. The condition on line 14 is key. Also notice that lines 12 and 15 are identical but appear at different levels of indentation. A good starting point would be to isolate the four lines of code into a separate program and experiment.

The following pseudo-code outlines a general pattern used to ensure some value entered by the end-user is valid.

```
read value from end-user
loop for as long as the value is invalid:
    [display error message]
    read value from end-user

process value
```

STUDENT TIP

A common student misconception is to think that a while loop's condition is being constantly evaluated and the loop exits the instant it becomes `False` inside the loop body.



Experiment!

Try making the following changes to see what happens.

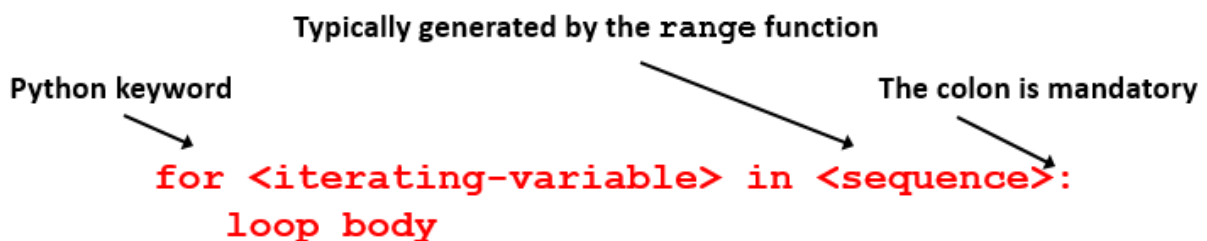
- Version 4 change the initial value of `counter` to 10
- Version 5 change the initial value of `correct` to be `True`
- Version 6 change the initial value of `keepGoing` to be `False`
- Version 7 remove (comment out line 18

The `for` loop

The `for` loop is a more specific iteration construct than its `while` counterpart in the sense that it is designed specifically for stepping through the items in a sequence.

`for` loops are the preferred looping mechanism when the number of required iterations is known in advance (of runtime). Because of their nature they are also commonly used to traverse strings and lists (which are both sequence types).

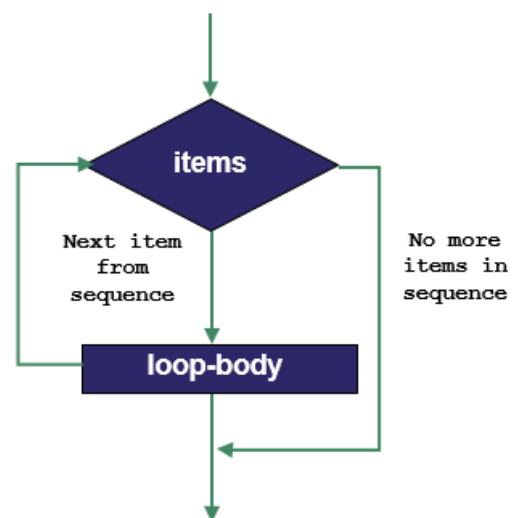
The syntax and semantics of a `for` loop are described and illustrated as follows:



The loop starts by assigning the first item in the *sequence* to the loop variable referred to as *iterating_variable*. Next, the statement(s) that make up the loop body are executed.

The loop continues the cycle of assigning the next item in the sequence to the iterating variable and then processing it in the loop body until the entire sequence is used up.

Observe the use of colon (`:`) and also that the statements which make up the loop body are indented.



for loop flowchart



KEY POINT:

It is essential to be able to recognise situations where a loop is required in a program. The choice of loop construct does not matter greatly. Most loops that can be programmed with a `while` construct can also be constructed using a `for` construct and vice versa.

Example

The short program serves to demonstrate main features of a `for` loop.


```

# Simple for loop

for counter in range(10):
    print("Hello World", counter)

print("Goodbye")

```



Simple `for` loop demo.

```

Hello World 0
Hello World 1
Hello World 2
Hello World 3
Hello World 4
Hello World 5
Hello World 6
Hello World 7
Hello World 8
Hello World 9
Goodbye

```

Program Output

The program output shown to the right above displays the message *Hello World* ten times. The value of `counter` is also displayed alongside the string. The string *Goodbye* is displayed once, before the program exits.

The first thing to observe is how much shorter this program is compared to the simple while loop program we say earlier.

In order to understand the example, it is helpful to understand how `range` works. `Range` can be thought of as a built-in function which returns a list of values. In the above example the call `range(10)` returns a list of all the integers from 0 to 9.

(see <https://docs.python.org/3/library/stdtypes.html#typesseq-range> for a complete description.)

The `for` loop works by iterating over each value in the sequence (i.e. 0 through to 9). At the start of each iteration the value of the next item in the sequence is assigned to the loop variable `counter`.

Notice how at the start of each iteration the iterating variable is assigned the next value in the sequence. The loop ends when the last value in the sequence has been used.

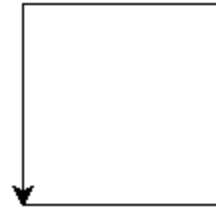
Once the loop terminates, execution continues at the next line after the loop body.

Example 1 - using a for loop to draw the square

Recall from section 1 of this manual the following code which uses the turtle graphics library to draw a simple square as shown.

```
# Draw a square
from turtle import *

forward(100)
left(90)
forward(100)
left(90)
forward(100)
left(90)
forward(100)
```



As can be seen the two lines shown (here to the right) are repeated 4 times – this is an indication that it should be possible to use a loop.

```
forward(100)
left(90)
```

The same functionality can be achieved by wrapping these repeating lines in a `for` loop as shown here to the right.

```
# Draw a square
for count in range(4):
    forward(100)
    left(90)
```

Example 2 – Guessing Game v4

The code below shows an implementation of Guessing Game v4 shown earlier. This version uses a `for` loop (line 8) instead of the `while` loop which was used in the original version. You should take some time to compare the two implementations.

```
1. # Guessing Game v4 using a for loop
2. import random
3.
4. number = random.randint(1, 10)
5. print(number) # For debugging purposes!
6.
7. # Loop 3 times
8. for counter in range(3):
9.
10.     guess = int(input("Enter a number between 1 and 10: "))
11.     if guess == number:
12.         print("Correct")
13.         break
14.     elif guess < number:
15.         print("Too low")
16.     else:
17.         print("Too high")
18.
19. print("Goodbye")
```

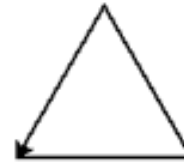
We are now ready to attempt our first `for` loop programming exercises.



Programming Exercises 5.2 (for loops)

1. Wrap the following code blocks in `for` loops to create the shapes shown.

```
forward(100)
left(120)
forward(100)
left(120)
forward(100)
```



```
forward(100)
left(90)
forward(50)
left(90)
forward(100)
left(90)
forward(50)
```



```
forward(100)
left(60)
forward(100)
left(60)
forward(100)
```



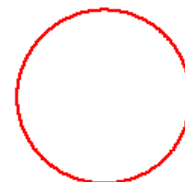
2. Write three separate programs to display the shapes shown.



A pentagon. There are 5 sides and the angle at each vertex is 72°



A hexagon. There are 6 sides drawn in a rotation of 360°



A circle.

3. The pygame program below displays the first three rows of the chequer board as outlined in the breakout session following Section 1 of this manual.

```
import pygame, sys
from pygame.locals import *

# start the pygame engine
pygame.init()

# create a 400x400 window
window = pygame.display.set_mode((400, 400))
pygame.display.set_caption('Chequer Board')

# define some colors
BLACK = (0, 0, 0)
WHITE = (255, 255, 255)

window.fill(WHITE) # paint the window white

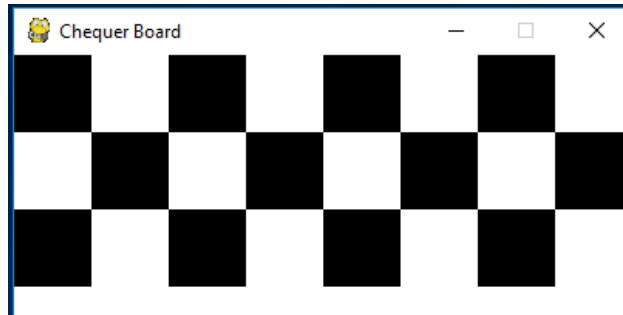
# Row 1
pygame.draw.rect(windowSurface, BLACK, (0, 0, 50, 50))
pygame.draw.rect(windowSurface, BLACK, (100, 0, 50, 50))
pygame.draw.rect(windowSurface, BLACK, (200, 0, 50, 50))
pygame.draw.rect(windowSurface, BLACK, (300, 0, 50, 50))
# Row 2
pygame.draw.rect(windowSurface, BLACK, (50, 50, 50, 50))
pygame.draw.rect(windowSurface, BLACK, (150, 50, 50, 50))
pygame.draw.rect(windowSurface, BLACK, (250, 50, 50, 50))
pygame.draw.rect(windowSurface, BLACK, (350, 50, 50, 50))
# Row 3
pygame.draw.rect(windowSurface, BLACK, (0, 100, 50, 50))
pygame.draw.rect(windowSurface, BLACK, (100, 100, 50, 50))
pygame.draw.rect(windowSurface, BLACK, (200, 100, 50, 50))
pygame.draw.rect(windowSurface, BLACK, (300, 100, 50, 50))

# update the window display
pygame.display.update()

# run the game loop
while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()
```



When the program is run it displays the pattern illustrated below.



Use the space provided to record how you might modify the program to exploit the power of loops

Dice Frequency Program

The program below simulates a one roll of a die by generating a random number between 1 and 6 and repeats this process one thousand times. The program then uses the `plotly` library to display a bar chart depicting the frequency count for 1000 rolls of a 6 sided die.

```
# Program to keep a count of the number of times each face of a dice is rolled
import random
import plotly
from plotly.graph_objs import *

faces=[1, 2, 3, 4, 5, 6,] # A list of die face values
count=[0, 0, 0, 0, 0, 0,] # A list for storing frequencies

for i in range(1000):
    # pick a random number between 1 and 6 (inclusive)
    roll=random.randint(1,6)
    count[roll-1]=count[roll-1]+1

# plot the results
plotly.offline.plot({
    "data": [Bar(x=faces, y=count)],
    "layout": Layout(title="Dice Results")
})
```



Experiment!

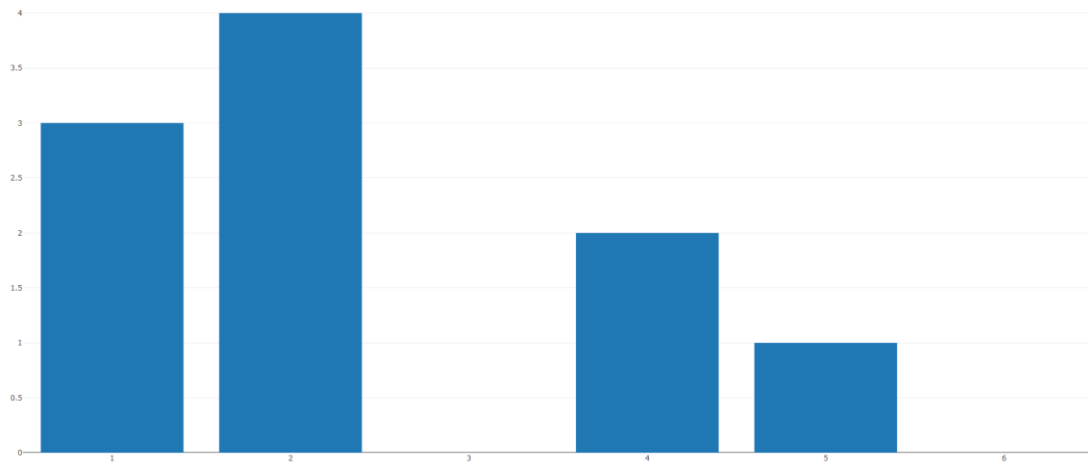
Key in (or copy+paste from GitHub) the program and make sure it runs without any syntax errors.

1. Modify the code by altering the number of dice rolls. What do you notice? What hypothesis could be developed by using this program?

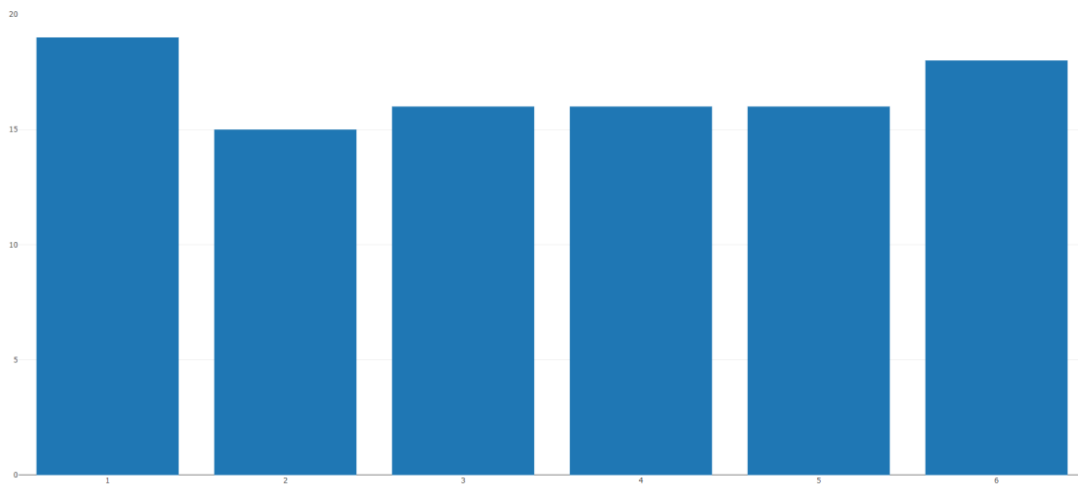
2. Examine the three bar charts on the next page. Can you explain the differences?

3. How might this exercise be adapted to some different context suitable for LCCS?

Dice Results



Dice Results



Dice Results





BREAKOUT ACTIVITIES (Iteration)

BREAKOUT 5.3: Text Analysis

The table below depicts some of the most common words found in two books – *Harry Potter and the Chamber of Secrets*⁶ and *Alice in Wonderland*⁷. Study the data carefully.

Harry Potter and the Chamber of Secrets	Alice in Wonderland
the 3755	the: 1507
to: 1987	and: 714
and: 1968	to: 703
a: 1703	a: 606
of: 1596	of: 490
was: 1243	she: 484
his: 1108	said: 416
Harry: 1065	it: 346
said: 1035	in: 345
he: 975	was: 328
in: 956	I: 261
had: 668	you: 252
at: 644	as: 237
you: 592	Alice: 221
it: 583	that: 213
that: 564	her: 203
as: 533	at: 197
I: 508	had: 175



What are your observations?

⁶ <https://github.com/pdst-lccs/lccs-python/blob/master/Section%205%20-%20Programming%20Logic/Breakouts/Breakout%205.3%20-%20Text%20Analysis/Harry%20Potter%20and%20the%20Chamber%20of%20Secrets.txt>
⁷ <https://github.com/pdst-lccs/lccs-python/blob/master/Section%205%20-%20Programming%20Logic/Breakouts/Breakout%205.3%20-%20Text%20Analysis/Alice%20in%20Wonderland.txt>

The data shown was generated by the program listing shown below.

```
import collections
import plotly
from plotly.graph_objs import Bar, Layout

bookFile = open("book.txt", "r") # Open the file
textList = bookFile.read().split()
bookFile.close()

c = collections.Counter(textList)

words = []
wordCount = []

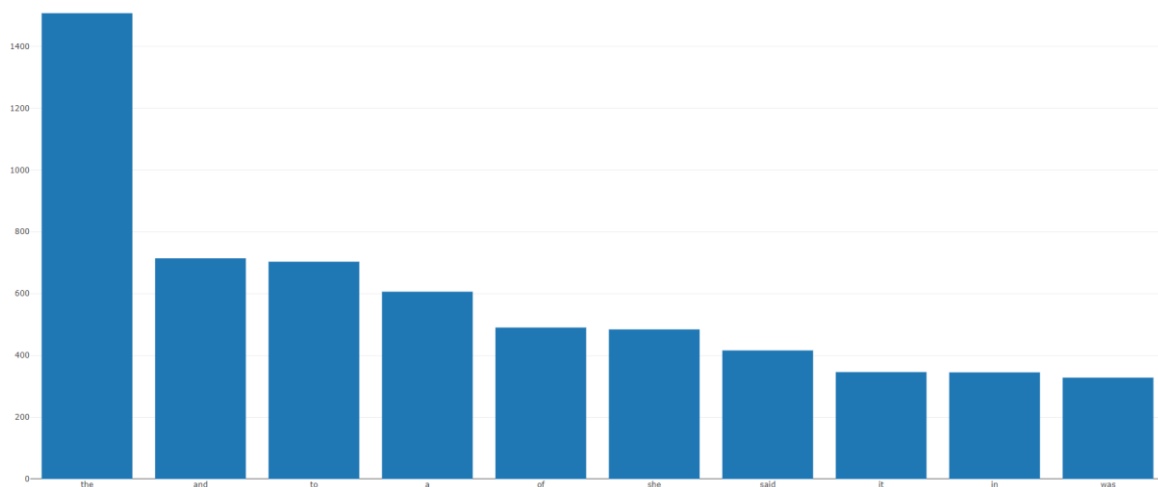
print ('Most common:')
for word, count in c.most_common(20):
    words.append(word)
    wordCount.append(count)
print ('%s: %7d' % (word, count))
```



Explain the purpose of the file `books.txt` in the above program.



Using your knowledge of `plotly` picked up from previous exercises can you add the necessary lines to the above program so that it generates a chart similar to that shown below – notice that number of most common words is reduced from 20 to 10.



BREAKOUT 5.4: Sample Applied Learning Task 2

Study the program below carefully and answer the questions that follow⁸.

```
import re
from collections import Counter
import plotly.plotly
from plotly.graph_objs import Bar, Layout

word = []
word_count = []

file = open("sample_text.txt", "r")

string = file.read()

file.close()

clean_string = re.sub('[^a-zA-Z0-9 \n\.]', ' ', string)
clean_string = re.sub('[0-9]', ' ', clean_string)
clean_string = re.sub('\t', ' ', clean_string)
clean_string = re.sub('\n', ' ', clean_string)
clean_string = re.sub(r'\b\w{1,4}\b', '', clean_string)
clean_string = re.sub(' ', ' ', clean_string)
clean_string = re.sub(' ', ' ', clean_string)
clean_string = re.sub('\.', '', clean_string)
clean_string = re.sub('Ãcã,-ã,,c', '', clean_string)
clean_string = re.sub('Ãcã,-ã€œ', '', clean_string)
clean_string = clean_string.lower()

string_array = clean_string.split(' ')

string_array = [x for x in string_array if x != '']

c = Counter(string_array)

unique_elements = set(string_array)

for letter in unique_elements:
    word.append(letter)
    word_count.append(c[letter])

print ('Most common:')
for letter, count in c.most_common(20):
    print ('%s: %7d' % (letter, count))

plotly.offline.plot({
    "data": [Bar(x=word, y=word_count)],
    "layout": Layout(title="word count")
})
```



⁸ Note: The file `sample_text.txt` may be downloaded from https://github.com/pdst-lccs/lccs-python/blob/master/Section%205%20-%20Programming%20Logic/Breakouts/Breakout%205.4%20-%20Sample%20ALT2/sample_text.txt



Identify the Python constructs and features contained in this program that you have learned about in this workshop.




Does the program contain any features that you are not familiar with? If so, what strategies would you use to overcome this knowledge deficit?

BREAKOUT 5.5: Maths Multiplication Tutor (MMT)

For this final task you are required to implement a Python maths tutor using the guess game we developed earlier to guide you along the way. Version 1 is provided below.

```
1. # Math multiplication tutor v1
2. import random
3.
4. n1 = random.randint(0, 12) # Generate the 1st number
5. n2 = random.randint(0, 12) # Generate the 2nd number
6. ans = n1 * n2 # Calculate the product and store it in 'ans'
7.
8. print("%d * %d" % (n1, n2)) # Display the expression
9. usrAns = int(input("Enter your answer: ")) # Read the response
10.
11. # Evaluate the condition
12. if usrAns == ans:
13.     print("Correct!")
14.
15. print("The correct answer was %d" % (n1*n2))
16. print("Goodbye")
```



Your implementation should be phased along the same lines used during the development of the guess game i.e. complete each version before moving on to the next.

MMT v1

This is a basic version provided above. The computer generates two random numbers and prompts the user to enter their product. If the user's answer is correct the program displays a message.

MMT v2

This time the program should display a message informing the user that they were either correct or incorrect based on the value they enter.

MMT v3

In this version of the application the user should be provided with more detailed feedback about their answer i.e. correct, too high or too low.

MMT v4

This is the same as version 3 except that the program should use a `while` loop to give the user *at most* three chances to get the correct answer.

MMT v5

In this version of the application, the program should continue until the user gets the correct answer. Each time the user enters an answer the program continues to display one of the three messages, i.e. correct, too high or too low.

MMT v6

This version should refine the previous version so that after entering the correct answer, the user is offered the opportunity to continue with a new expression. If the user enters “N” for no the application should exit. Otherwise the program should generate two new random number and the application should start again.

MMT v7

In this final version of the application, the functionality should remain exactly the same as version 6. However, this version should validate any data entered by the user i.e. it checks that the user’s answer is in fact a number before proceeding.



Identify five additional requirements that could be incorporated into the MMT application.

1.

2.

3.

4.

5.

Section 6

Modular Programming using Functions

Introduction

Functions are the building blocks of programs. They allow programmers to organise their code into logically-related sections.

In order to understand where functions fit into the overall scheme of things it is useful to have some understanding of the architecture of a Python program. This is depicted in the following illustration

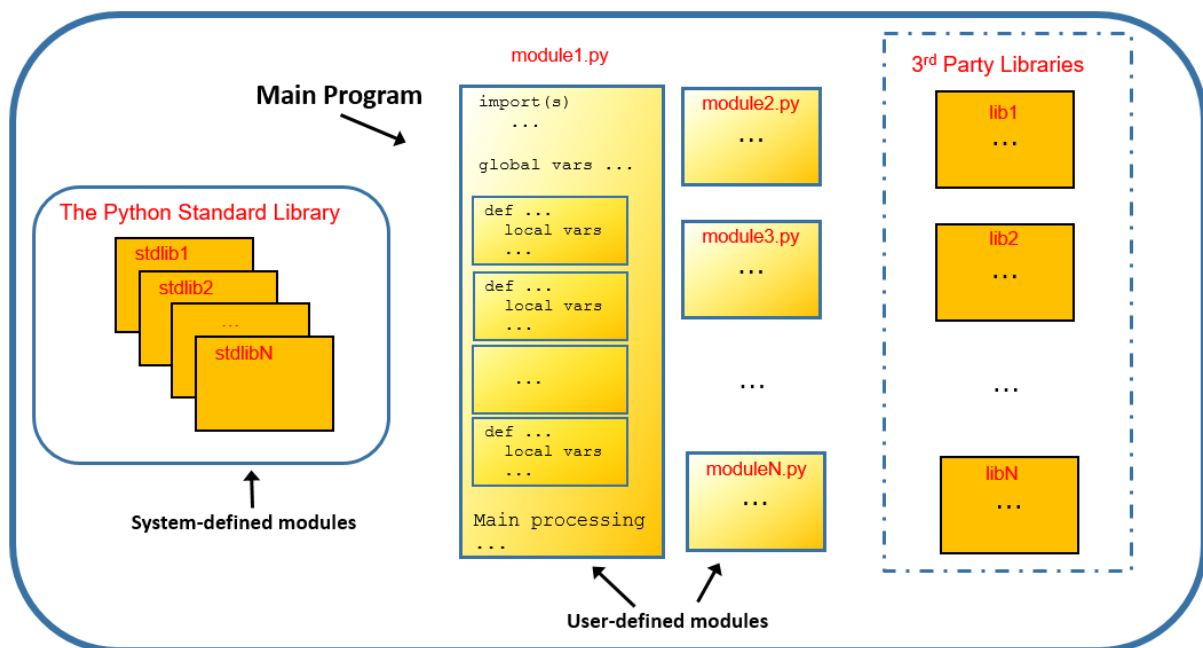


Figure: Python Program Architecture

As can be seen a Python program is typically made up of many components. One of the key components is called a *module*. Modules can be thought of as individual Python script files (i.e. a file with a `.py` extension) made up mostly of functions. It is useful to think of a Python program as a collection of modules.

At runtime, modules and function co-operate with one another to achieve some desired result. Program execution starts in one special module called the *top level module* which is also commonly referred to as the 'main' program.

Modules can be classified into the following three groups:

1. **User-defined modules:** These are modules that are written by the programmer as part of the program which is being developed (aka the current development)
2. **Standard library modules:** These are modules that come pre-installed as part of Python. The Python standard library is designed to save programmers from having to come up with their own solutions to common programming problems. As such, it comprises a full suite of off-the-shelf, ready to go solutions in the form of built-in functions.

Some examples of Python libraries we have already come across are `math`, `random`, `statistics`, and `turtle`. See <https://docs.python.org/3/library/index.html> for a complete reference.

3. **3rd party modules:** These are modules that are developed by an external source either for commercial purposes or as open source. There are literally thousands – some examples include `tkinter`, `numpy`, `plotly`, `scipy`, `Django`, and `flask`. The Python Package Index (PyPI) is a repository of software for the Python programming language. See <https://pypi.org/> for more information.



KEY POINT: A Python program consists of one or more modules and modules are made up of functions. Each individual module is a Python script or `.py` file.

So, modules and functions are constructs used by programmers to organise their code into separate units (or chunks). A package is another such construct – it is used to group a number of related modules into a single entity. Conceptually, it is helpful to think of a package as a set of modules that reside on the file system in the same folder/directory.

Programmers can use the Python `import` statement to make the functionality of external modules accessible to the script they are currently developing. These external modules can be individual modules or multiple modules that have been grouped together into a package.

For the purpose of Leaving Certificate Computer Science (LCCS), a typical Python program might be made up of a single file that draws on and exploits the functionality made available by the standard library and, potentially, some other third party package(s).

Most Python files are organised into three sections – the import statements (typically at the top), the function definitions (by far the longest and most important section) and finally the main code i.e. the section from where the program execution begins.

We will now take a closer look at functions.

Functions

The ‘art of computer programming’ can be seen as a process of designing and creating individual functions and combining them together into larger units of code called modules. Over time, programmers combine these modules to produce the final program.

We have already learned that functions provide a means for programmers to organise their code, but what exactly is a function and why are they important?



KEY POINT: A *function* is a short piece of re-usable code that carries out a specific task.

Each individual Python module is (mostly) made up of functions.

User-defined vs Built-in functions

Functions are very useful because they typically provide solutions to common programming problems e.g. display some text on the screen, read data from the end-user, send a tweet, process a cash withdrawal etc.

In certain cases, the programming problems are so common that Python provides a built-in function to do the job. Such functions are called *built-in functions*.

In other cases, the problem to be solved is so specific to the program being developed that the programmer needs to design and write the code themselves. Such functions are called *user-defined functions*.

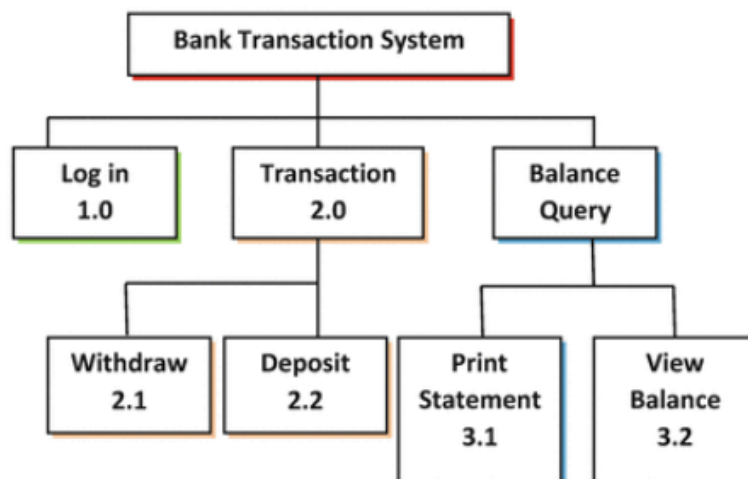
All functions, regardless of whether they are built-in or user-defined are given a name as part of their definition. The function name can then be used in a program to invoke the code contained in the function.

We'll examine the syntax and semantics of defining and invoking functions shortly but first let's take a closer look at the two main reasons why functions are considered important:

1. they lead to modular systems
2. they can be used to maximise code reuse and minimise code redundancy

Large scale software systems are very often developed by breaking big problems down into smaller problems (decomposition). While designers and programmers are working on the detail of one part of the system they can ignore the rest of the system (abstraction).

Consider the functional decomposition diagram of an ATM/Bank Transaction system show below. The diagram shows the system broken down into a number of smaller sub-systems. The functionality of each sub-system can be implemented using functions. This piece-by-piece approach to developing systems is sometimes called *divide and conquer*.



Functions are a programming construct that support this divide and conquer approach to software development. They lend themselves to *modular systems* which are easier and less costly to maintain than their non-modular counterparts.

A second reason why functions are so important is that they can be used to minimise (and even avoid) code duplication. Consider the following example.

Example 1 – Maximising code reuse and minimising redundancy

Study the short program shown below that displays a slightly adapted version of the Dr Seuss poem, *Green Eggs and Ham*

```
# A program to display Green Eggs and Ham
1. print("I do not like green eggs and ham.")
2. print("I do not like them Sam-I-am.")
3. print() # display a blank line
4. print("I do not like them here or there.")
5. print("I do not like them anywhere.")
6. print("I do not like them in a house")
7. print("I do not like them with a mouse")
8. print() # display a blank line
9. print("I do not like green eggs and ham.")
10. print("I do not like them Sam-I-am.")
11. print() # display a blank line
12. print("I do not like them in a box")
13. print("I do not like them with a fox")
14. print("I will not eat them in the rain.")
15. print("I will not eat them on a train")
16. print() # display a blank line
17. print("I do not like green eggs and ham.")
18. print("I do not like them Spam-I-am.")
```



Evaluate the above program by asking - do you recognise any patterns? Is there any code duplication? Can you spot any typing mistakes? Elaborate.

The above code is considered poor design because it contains duplication. Lines 1,2 are duplicated in three different places. This is an example of redundant code. To eliminate the redundancy, we write a *function* to display the chorus.

The program below uses a function to eliminate the duplication referred to earlier.

```
# A program to display Green Eggs and Ham (v2)
1. def displayChorus():
2.     print()
3.     print("I do not like green eggs and ham.")
4.     print("I do not like them Sam-I-am.")
5.     print()
6.
7. displayChorus()
8. print("I do not like them here or there.")
9. print("I do not like them anywhere.")
10. print("I do not like them in a house")
11. print("I do not like them with a mouse")
12. displayChorus()
13. print("I do not like them in a box")
14. print("I do not like them with a fox")
15. print("I will not eat them in the rain.")
16. print("I will not eat them on a train")
17. displayChorus()
```



This program displays the same output as the program on the previous page – the only difference is that this program uses a function called `displayChorus` to eliminate duplication of code.

Every function must have a name (assigned by the programmer). In this case, the name of the function is `displayChorus` and it is defined (or made known to Python) on lines 1–5 inclusive.

The lines from line 7 onwards are executed in sequence. Lines 7, 12 and 17 *call* the function `displayChorus`. Every time the function is called the lines 2-5 are executed. Even though these lines only appear once in the program they are used on three different occasions.



KEY POINT: When a function is *called*, the flow of control jumps to the first line of the function and execution continues from that point to the last line of the function. Once the last line of the function has been executed the flow of control jumps back to the point from which the call to the function was initially made.

STUDENT TIP

Avoid duplicating blocks of code. Code duplication is considered to be symptomatic of 'poor code'.

If you find that you need to re-use a number of lines of code to do something specific use a function instead of duplicating the code.

Example 2 – Modular Code

Let's continue with the same example to demonstrate the use of functions to develop modular code.

In this program, the code to display each part of the poem is '*factored*' into separate functions. The program is considered better than the two previous versions because it is more *modular*. Modular code is the result of good design and is both easier and less costly to maintain than non-modular code.

```
# A program to display Green Eggs and Ham (v3)
1. def displayChorus():
2.     print()
3.     print("I do not like green eggs and ham.")
4.     print("I do not like them Sam-I-am.")
5.     print()
6.
7. def displayVerse1():
8.     print("I do not like them here or there.")
9.     print("I do not like them anywhere.")
10.    print("I do not like them in a house")
11.    print("I do not like them with a mouse")
12.
13. def displayVerse2():
14.    print("I do not like them in a box")
15.    print("I do not like them with a fox")
16.    print("I will not eat them in the rain.")
17.    print("I will not eat them on a train")
18.
19. displayChorus()
20. displayVerse1()
21. displayChorus()
22. displayVerse2()
23. displayChorus()
```

Consider the differences between the program shown here and the two programs shown earlier in this section.

Even though the three programs are different they all do the same thing.

Notice the different level of indentation inside each function body.

The above listing defines three functions as follows:

- The function `displayChorus` is defined on lines 1-5
- The function `displayVerse1` is defined on lines 7-11
- The function `displayVerse2` is defined on lines 13-17

Lines 19-23 are executed in sequence and cause the poem to be displayed.

Summary

- A Python program is made up of one or more modules. Each individual module is a Python script or `.py` file
- Modules are made up mostly of functions.
- Functions are the building blocks of modules (and by extension, programs)
- A function is a short piece of re-usable code that carries out a specific task.
- All functions have a name which must be used to invoke the function's code
- Python comes with a set of pre-installed modules - called the *standard library*
- Logically related modules can be grouped together into packages. A package is made up of multiple modules.
- The functionality of individual modules and entire packages can be made accessible to other Python files by using the `import` statement
- Functions that do not require the `import` statement in order to be accessible are known as *built-in functions* e.g. `print`, `input`. Python 3.6.x comes with 67 built-in functions; these are listed in the appendix – more details can be found by browsing to <https://docs.python.org/3/library/functions.html>



Reflect on what you have learned about functions so far. Use the space below to explain what functions are and why they are important.



Develop your own example to motivate the use of functions. (The example used in the text was the Dr Seuss poem, Green Eggs and Ham)

Basic Function Syntax

Once the 'need' for a function in your program has been recognised (the difficult bit!) the next step is to write the code. For this, we need to understand some syntax (the easy bit!).

Functions are made known to Python by writing a *function definition*. In Python, the function definition is made up of two parts - a *header* and a *body*.

- the function header (aka the function *signature* or *prototype*) is always the first line of the function definition
- the function body contains the Python statements required to carry out the work of the function.

The code below illustrates the definition of a function called `displayPoem`. The function header is on line 1 and the function body runs from line 2 to 5 inclusive.

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



Function definition for displayPoem

The function header (this is the very first line in the function definition)

Every function header is composed of four separate parts:

- (i) The word **def**: This is a Python keyword which tells Python to create a new function object. Every function must start with the `def` statement.
- (ii) The function name (in this case `displayPoem`):
It is up to the programmer to decide what name to give a function. The rules for naming functions are the same as those for naming variables.
- (iii) Brackets: These provide a mechanism for passing information into functions. In this example, no information is being passed into the function and therefore nothing appears between the opening and closing brackets.
- (iv) Colon: The colon is used to terminate the function header. If the colon is omitted, Python will display a syntax error.

The function body

The body of every Python function consists of one or more Python statements. These statements combine to provide the function's task. Although there is no limit to the number of statements that can be in a function body, it is generally considered good practice to keep functions short.

Notice how the four lines of code that make up the function body (lines 2 – 5) are indented. In Python, the statements inside a function body must always be indented. The function body ends when the indentation ends i.e. when the next statement appears at the same level of indentation as the `def` statement.

Calling a function

It is important to realise that the code inside a function body will not be executed unless the programmer explicitly requests it to be. The term for such a request is a *function call*.



KEY POINT: A *function call* causes the code inside the function body to be executed.

Functions can be called (or invoked) at runtime by writing the name of the function followed by brackets. The code to call the function `displayPoem` is shown below.

```
displayPoem()
```

This line calls the function `displayPoem`

Note, the brackets are needed but `def` and colon are not

When the above call is made the four lines in the function body are executed thus causing the following four lines of text to be displayed on the output console.

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


The semantics of a function call are now explained.

Call semantics

Consider the order in which the lines of code in example program below are processed by Python.

```

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



Python starts at line 1 and notices that it is a function definition and skips over all of the lines in the function definition until it finds a line that is no longer included in the function (line 8). On line 8 it notices that it has a function to execute, so it goes back and executes that function – lines 2-6 inclusive. Once all the lines in the function body have been executed it continues at line 9. The result of the above program is:

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



Describe how the above program could be modified so that it would output the text shown *in italics* below.

Can you come up with more than one solution and if so which solution do you think is better?

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



Design and write a function to display the output shown below.
Write a line of code to call your function.

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



Study the program shown carefully and identify *two* syntax errors.

```
# A program to display Green Eggs and Ham (v4)
1. def showChorus():
2.     print()
3.     print("I do not like green eggs and ham.")
4.     print("I do not like them Sam-I-am.")
5.     print()
6.
7. def showVerse1():
8.     print("I do not like them here or there.")
9.     print("I do not like them anywhere.")
10.    print("I do not like them in a house")
11.    print("I do not like them with a mouse")
12.
13. def displayVerse2():
14.    print("I do not like them in a box")
15.    print("I do not like them with a fox")
16.    print("I will not eat them in the rain.")
17.    print("I will not eat them on a train")
18.
19. showChorus()
20. displayVerse1()
21. showChorus()
22. showVerse2()
23. showChorus()
```





SYNTAX CHECK: When Python comes across a word it does not understand it displays a syntax error. Therefore, when a call is made to a function using a name, a function of that name must already have been made known to Python with the `def` keyword.

Guidelines and Rules for Naming Functions

We begin with a few simple guidelines as opposed to actual rules. These guidelines help improve program readability (and therefore maintainability).

Function names should be meaningful i.e. they should in some way describe what the function does. Since functions are usually actions, the name should contain at least one verb.

If a function name is only one single word, the convention is to use lowercase; if the name of the function is made up of more than one word, the use of camel case or underscore as a means of delimiting the words is considered good practice.

The basic syntax rules for naming functions are:

- A function name cannot be a Python keyword (e.g. `def`, `if`, `while`, etc.)
- Function names must contain only letters, digits, and the underscore character, `_`.
- Function names cannot have a digit for the first character.



Comment on the validity and quality of each of the following function names

a) `sendTweet`

b) `calculate_salary`

c) `_login`

d) `bin`

e) `2binary`

f) `MAX_OF_3`

g) `Binary Search`

h) `Search*`

i) `return`

j) `Print`

k) `doSomething`

Function Parameters and Arguments

Let's return to the homework function we were looking at earlier and ask the question – how can we modify the function to display the verse using names other than Jack?

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



Every time this function is called it always displays precisely the same text. We'd like to move away from the concrete name – Jack – to a more abstract name – anybody.

One solution would be to have a different version of the function for each different name we wanted to have it display, but this would contradict the whole purpose of functions which is to eliminate code duplication.

Another solution is to use *parameters*.

What we really want is some way of telling the function what name to display as part of the verse i.e. we need a means to pass information into the function from the code outside the function. This is exactly the purpose of parameters.

A *parameter* is a special kind of variable which appears as part of the function header and can be used inside the function body. Take a look at this!

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



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

The function `homework` modified to use a parameter

Program output

STUDENT TIP

When you want to pass information into a function use parameter(s)



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

Notice `personName` appears between the brackets in the function header? This is a function **parameter**. Parameters are received by functions.

Notice also the text *David* between brackets in the function call (line 8)? This is a function **argument**. Arguments are passed into functions.

Functions can be defined to accept multiple arguments.

```
# Functions can have multiple parameters
1. def displayMessage(msg1, msg2):
2.     print(msg1)
3.     print(msg2)
4.     # End of function
5.
6. displayMessage("Hello world", "How are you today?")
```



Hello World
How are you today?

The function displayGreeting has two parameters

Program output

When more than one parameter is received by a function they are separated by commas. For example, the two parameters `msg1` and `msg2` are separated by commas on line 1. Similarly, when more than one argument is passed into a function they are separated by commas. For example, on line 6 above the two arguments, *Hello world* and *How are you today?* are both separated by commas on line 1.

When the function `displayGreeting` is called Python performs two assignments:

- the value of the first argument (i.e. *Hello world*) is assigned to the parameter `msg1`.
- the value of the second argument (i.e. *How are you today?*) is assigned to the parameter `msg2`.

Parameters are received into a function in the same order as the arguments provided. Check what happens if the arguments were switched around like this in the function call on line 6.

```
6. displayMessage("How are you today?", "Hello world")
```

Thus far in this section the arguments used in the example programs have all been string literals. However, arguments can be literals of any datatype (e.g. numeric, Boolean etc.); arguments can also be expressions made up of variables and/or literal values together. Be careful though - as a general guide the number of arguments provide should match the number of parameters provided for in the function header.



Experiment!

Predict what the following calls to the function `displayMessage` would do? After making each prediction you should use Python to see if you were correct.

1. `displayMessage(1, 2)` 

2. `displayMessage(1, 2, 3)`

3. `displayMessage("Testing", 123)`

4. `displayMessage("Testing", 1 2 3)`

5.

```
str1 = "Hello world"
str2 = "How are you today?"
displayMessage(str1, str2)
```

6.

```
str1 = "Hello world"
str2 = "How are you today?"
displayMessage(str2, str1)
```

7.

```
str1 = "Dear "
str2 = "John"
displayMessage(str1+str2, "Hi!")
```

8.

```
displayMessage(1+1, 2)
```

9.

```
pi = 3.14
r = 5
displayMessage("Area:", pi*r**2)
```

10.

```
pi = 3.14
r = 5
displayMessage("Area:", "pi*r**2")
```



Reflect.

What conclusion(s) about the syntax and semantics of parameters and arguments did you arrive at from the previous experiment?



Devise a number of test cases that would test the assertion that the number of arguments passed in as part of the function call should match the number of parameters provided for in the function header



KEY POINT: The advantage of using parameters and arguments is that they make functions much more flexible and provide for more general solutions to problems.

The runtime behaviour of a function can be altered by passing different arguments into it. This is useful, and a very common way, of achieving *abstraction*.

Function Return Values

Functions can be thought of little machines (black box) that accept input(s) and produce output(s).



In the previous section we learned that data can be passed into functions through the use of arguments (at the function call) and parameters (as part of the function header). In this section we explore the use of the `return` statement as a means to pass data out of a function.

Consider the function shown below to add the first n non-negative integers.

```
# Add all the numbers from 0 to n
1. def sumOfN(n):
2.     total = 0
3.
4.     for i in range(n+1):
5.         total = total + i
6.
7. return total
8.
9. sumOfN(10) # call the function
```



`range(n)` generates a list of integers from 0 to $n-1$ e.g. `range(10)` [0, 1, 2, 3, ... 9]

The `for` loop iterates over each integer with the value of the next item in the sequence being assigned to the loop variable `i`

The function works by maintaining a running total of all the numbers from 0 to $n+1$ in the variable `total`. At the start of each loop iteration, the loop variable (`i`) is assigned the next value in the sequence. On each iteration of the loop, the value of the loop variable is added to `total` and the result is used to update `total` with the new running total. The loop ends after the last value in the sequence has been processed.

Line 7 shows the `return` statement being used to pass the value of `total` out of the function.

Line 8, `sumOfN(10)` calls the function passing in an argument of 10. The intention is to calculate the sum of all the integers from 0 to 10. .



Experiment!

Key in (or copy+paste from GitHub) the above code and make the function call. Explain why nothing appears to happen?

The reason nothing appears to happen is that although the function is called and it does calculate and return the sum of the first 10 non-negative integers, the calling code takes no action to save or process the result.



KEY POINT: The return value of a function can be saved for further processing by making the function call part of an assignment statement.

Line 9 in the code below assigns the result of the function (i.e. `total`) to the variable `answer`.

```
# Add all the numbers from 0 to n
1. def sumOfN(n):
2.     total = 0
3.
4.     for i in range(n+1):
5.         total = total + i
6.
7.     return total
8.
9. answer = sumOfN(10) # call the function and save the result
10. print("The sum of the first 10 integers is %d" %answer)
```

The value of `total` is passed out of the function (line 7) and assigned to the variable `answer` (line 9)


STUDENT TIP

When you want to save the return value of a function, the function call should be made as part of an assignment statement :

`<variable-name> = <function-call>`

Alternatively, the programmer may decide there is no need to save the result of a function in a variable and process the result as part of the call. This type of *inline processing* is shown on line 10 of the code below where the result of the function is passed in as an argument to the `print` built-in function.

```
# Add all the numbers from 0 to n
1. def sumOfN(n):
2.     total = 0
3.
4.     for i in range(n+1):
5.         total = total + i
6.
7.     return total
8.
9. # call the function and display the result
10. print("The sum of the first 10 integers is", sumOfN(10))
```




In this example the result of the function is not caught by the calling code. Rather, it is passed directly as an argument to `print`. This technique is called *function composition*.

In situations when a functions are used directly as arguments to other functions, Python starts at the innermost function and works its way out i.e. the inner function is always evaluated first and evaluation continues from right to left.

For example, let's say we had a function called `sub` to subtract two integers (b from a) as shown below.

```
def sub(a, b):
    answer = a - b
    return answer

print( sub( 2, sub(3, 4)))
print( sub( sub(2, 3), 4))
print( sub( 2, sub(3, sub(4, 5))))
```



When run, the code would display 3, -5, and -2 on separate lines.

As a final note it is worth pointing out that It is not always necessary for a function to return data. A return statement may be omitted entirely or can be used without an expression (i.e. just `return` on its own). In both cases the value returned by Python is `None`.

Examples and Exercises

Study each of the following examples carefully – read the code first, predict what it would do. Then, key the code in and run it to test your prediction(s). In each case you should complete the reflection exercise provided before you finally move on to implement the programming challenges at the end.

Temperature and Distance Conversions

The program below implements two functions:

- (i) `cent2fhar` converts from Centigrade to Fahrenheit and
- (ii) `kms2miles` converts from kilometres to miles

```
# Convert centigrade to fharenheit
def cent2fhar(centigrade):
    return 9/5*centigrade + 32

# Convert kilometers to miles
def kms2miles(kilometers):
    return( kilometers * 0.62)

# Test cent2fhar
cent = int(input("Enter a value in centigrade: "))
fhar = cent2fhar(cent)
print(cent, "degrees C is", fhar, "degrees F")

# Test kilometers to miles conversion
kms = int(input("Enter a value in kilometers: "))
print("%dkms = %.2fmiles" %(kms, kms2miles(kms)))
```



What you have learned about functions from this example?



1. Modify the program so that the constant values (e.g. 9/5, 32 and 0.62) are stored in variables
2. Extend the program with functions to convert from a) miles to kilometres and b) Fahrenheit to Centigrade

Compound Interest

The program below can be used to find out which would yield a greater future value:

Scenario A: €10,000 at a rate of 5% for 10 years or

Scenario B: €10,000 at a rate of 10% for 5 years

```
# Calculates future value
def future_value(principal, rate, time):
    fv = principal * (pow((1 + rate / 100), time))
    return fv

# Driver Code
fv_A = future_value(10000, 5, 10)
fv_B = future_value(10000, 10, 5)
print("Scenario A: %.2f \nScenario B: %.2f" %(fv_A, fv_B))
```



The formula used is $FV = p(1 + i)^t$ where, p is the initial principal, i the interest rate, expresses as a decimal, and t the time in years.

(`pow(x, y)` is a built-in function that returns x raised to the power of y .)



What you have learned about passing information in and out of functions from this example?

1. Modify the program so that it can accept the values for principal, interest and time from the end-user
2. Extend the program so that it can compare compound interest with simple interest



STUDENT TIP

When you want to pass information out of a function use `return`

Maximum of Three Numbers

The function below finds and returns the largest of any three integers.

The solution is based on three, two-way comparisons in which each of the integers is compared to the other two. The individual comparisons are combined into a compound Boolean condition using the `and` operator.

```
# A function to find the largest of 3 numbers
def maxOf3(x, y, z):
    if (x > y) and (x > z):
        return x
    elif (y > x) and (y > z):
        return y
    elif (z > x) and (z > y):
        return z

# Test the function
print(maxOf3(1, 2, 3))
print(maxOf3(1, 3, 2))
print(maxOf3(3, 2, 1))
```



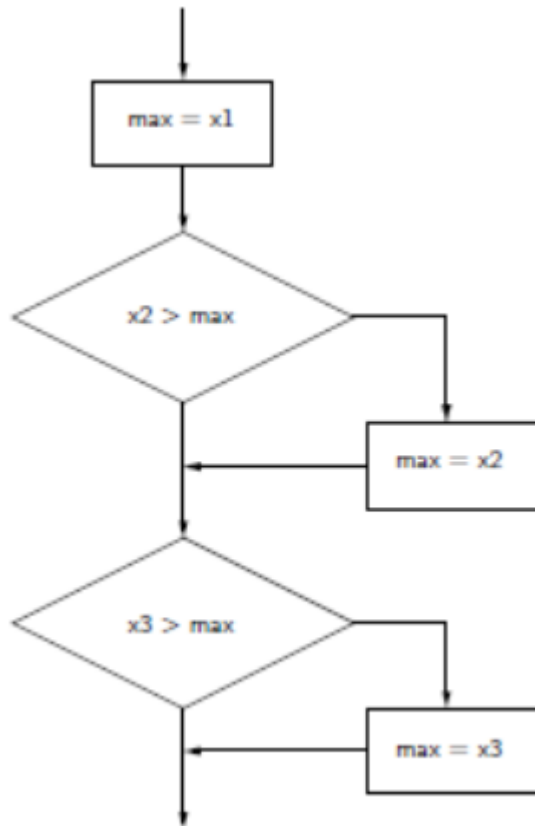
Reflect.

How has this example extended your knowledge in relation to forming Python Boolean expressions?

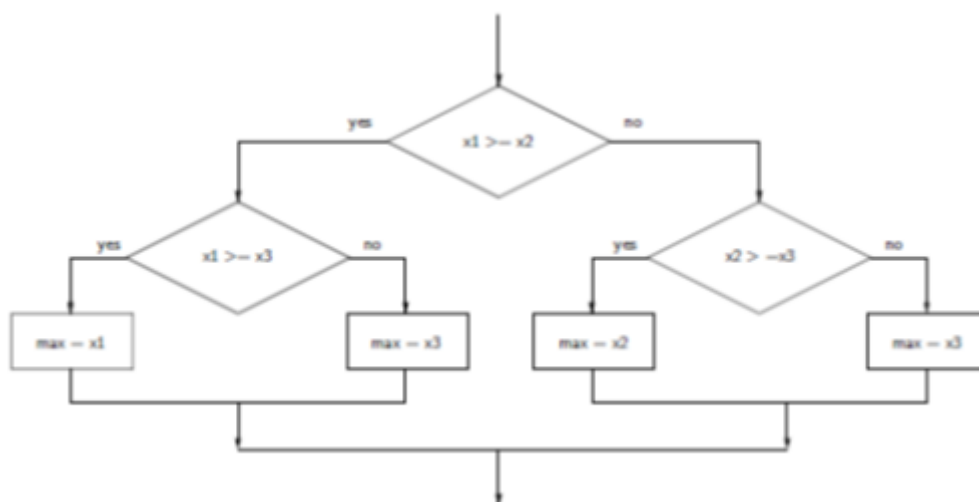


1. Modify the program so that it prompts the end user to enter three values and then displays their maximum in a meaningful output message.
How would this implementation be altered if the three numbers were a) randomly generated or b) read from a file?
2. Use the flowcharts on the next page to implement two alternative solutions for `maxOf3`
3. Design a flowchart to find the minimum of three numbers.
4. Use your flowchart to implement a function to find the minimum of three values. Call it `minOf3`. You could start by copying and pasting the `maxOf3` function definition and renaming it.

Flowchart 1



Flowchart 2



Boolean Functions

Boolean functions are functions that return either `True` or `False` usually to indicate the outcome of some test. By convention the name of a Boolean function starts with the prefix `is` e.g. `isEven` might be a Boolean function that tests the 'evenness' of a number.

```
# A function to determine evenness
def isEven(number):
    if (number % 2 == 0):
        return True
    else:
        return False
```

The function uses the remainder operator (`%`) to test whether the value passed in as `number` is even or not – if it is the function returns `True`. Otherwise, the function will return `False`.

The code shown here to the right demonstrates how the above function could be used to display all the even numbers between 1 and 100.

```
# display even numbers < 100
for i in range(100):
    if isEven(i):
        print(i)
```



The line `if isEven(i):` is the key. Here, the call to the function appears as part of a conditional statement. This is fine, since conditions evaluate to `True` or `False` and the function `isEven` is guaranteed to return one of these values.

We can exploit our knowledge of even and odd numbers to define the function `isOdd` as shown. The function applies the `not` operator to the result of the call to `isEven` and returns the result to the caller.

```
# A function to determine oddness
def isOdd(number):
    return not isEven(number)
```

This is a good example of abstraction because the implementation of `isOdd` hides the detail.



Implement the following two Boolean functions

```
def isEqual(n1, n2):
```

A function to take two values and return `True` if they are both equal; `False` otherwise.

```
def isDifferent(a, b):
```

A function to do the opposite to `isEqual`

A prime example

A prime number is a positive integer that has exactly two factors; itself and 1.

The Boolean function below `isPrime` determines whether the number passed in is prime or not. The function will return `True` if number is a prime number; `False` otherwise.


```
import math

# A function to test whether a number is prime or not
# Returns True if the number is prime; False otherwise
def isPrime(numToCheck):

    # Any number less than 2 is not prime
    if numToCheck < 2:
        return False

    # see if num is evenly divisible by any number up to num/2
    divisor = 2
    while (divisor <= numToCheck/2):
        if (numToCheck % divisor == 0):
            return False
        divisor = divisor+1

    # The number must be prime so return True
    return True
```



The function works by attempting to divide every integer from 2 up to half the number being tested (`numToCheck`) - if the division leaves no remainder, it means the number being checked has factors and is therefore not prime. The following exercises, based on the test harness provided are designed to be used to explore the function `isPrime`.



```
# Test harness
for num in range(100):
    if (isPrime(num)):
        print(num, "is prime")
```

A test harness for `isPrime`

- Predict what the test harness would do?
- Run the test harness? Was your prediction correct?
- Investigate. The program looks for factors up to half the number being checked. What would happen if the program stopped at the square root?
- Modify the test harness so that it displays the first 100 prime numbers.
- Make a program that displays the n^{th} prime where n is a number entered by the end user.

A case study of leap years

Boolean functions provide a useful framework which can be used to determine whether a given year is leap or not. One example is presented as follows:

```
# Determines whether n is leap or not
def isLeap(n):
    if n % 400 == 0:
        return True
    if n % 100 == 0:
        return False
    if n % 4 == 0:
        return True
    else:
        return False

yr = int(input("Enter a year: "))
if isLeap(yr):
    print(yr, "is a leap year")
else:
    print(yr, "is NOT a leap year")
```

Key in/download the code and use it to determine whether the years listed are leap or not.

2000 _____

1900 _____

2017 _____

2012 _____

2100 _____

2400 _____

2600 _____



Use the code to derive and record five facts about leap years

1. _____
2. _____
3. _____
4. _____
5. _____



Describe when is a year a leap year



Now examine the two implementations below – one is correct and one contains a subtle error (i.e. one version reports incorrectly that 2000 is not a leap year.)

```
def isLeapV1(year):
    if year % 4 == 0 and year % 100 != 0:
        return True
    elif year % 100 == 0:
        return False
    elif year % 400 == 0:
        return True
    else:
        return False
```



```
def isLeapV2(year):
    if year % 4 == 0 and year % 100 != 0:
        return True
    elif year % 400 == 0:
        return True
    elif year % 100 == 0:
        return False
    else:
        return False
```



Can you explain the subtle error and suggest/implement a solution.




In isLeapV2 above can you justify the need for the second elif block? Explain.

STUDENT TIP
Use a Boolean function as a means to organise a block of code that performs any test that leads to one of two possible outcomes - True or False




Encapsulate the code shown below into two functions – call them *isLeapV3* and *isLeapV4*

```
year = int(input("Enter a year: "))
if year % 4 == 0 and (year % 100 != 0 or year % 400 == 0):
    print(year, "is a leap year")
else:
    print(year, "is NOT a leap year")
```



```
year = int(input("Enter a year: "))
if (year % 4) == 0:
    if (year % 100) == 0:
        if (year % 400) == 0:
            print(year, "is a leap year")
        else:
            print(year, "is NOT a leap year")
    else:
        print(year, "is a leap year")
else:
    print(year, "is NOT a leap year")
```




Compare and contrast the two functions *isLeapV3* and *isLeapV4*.



Write a one line Boolean function to determine whether a given year is leap or not

Using Functions to Validate Data

Functions provide programmers a convenient way to organise code that perform specific tasks such as data validation. Consider the following scenarios and examples.

1. One common scenario faced by novice programmers is the need to read non-negative integers from the end user. Since the `input` command returns a string we need to write code that ensures the data entered is numeric before converting it to an integer using `int`

The following four solutions (and there are many more!) are offered for consideration.

```
def readIntegerV1():
    strN = input("Enter a number: ")
    while not strN.isdigit():
        strN = input("Enter a number: ")
    return int(strN)
```

```
def readIntegerV3():
    while True:
        strN = input("Enter a number: ")
        if strN.isdigit():
            break
    return int(strN)
```

```
def readIntegerV2():
    valid = False
    while not valid:
        strN = input("Enter a number: ")
        if strN.isdigit():
            valid = True
    return int(strN)
```

```
def readIntegerV4():
    while True:
        strN = input("Enter a number: ")
        for i in range(0, len(strN)):
            if strN[i] not in '0123456789':
                break
        else:
            break
    return int(strN)
```



Which of the above solutions do you prefer and why?

Would the functions work to read negatives or floating-point values?


2. Building on the previous scenario, it may be that we need to make sure that the number entered is within a specific range.

This time we offer two versions of a solution.

```
def readIntegerInRangeV1(lwr, upr):

    valid = False
    while not valid:
        strN = input("Enter a number: ")
        if strN.isdigit():
            n = int(strN)
            if (n >= lwr) and (n <= upr):
                valid = True


    return n
```



```
def readIntegerInRangeV2(lwr, upr):

    valid = False
    while not valid:
        strN = input("Enter a number: ")
        if strN.isdigit():
            n = int(strN)
            if lwr <= n <= upr:
                valid = True

    return n
```





Compare and contrast the two solutions. What is the main difference? Comment on the use of the flag `valid` in the above code. Under what circumstance is the value of `valid` set to `True`?

3. The function below can be used to ensure a user enters either Y or N

```
def readYesNoResponse():

    response = input("Enter response [Y/N]: ")
    while response != "Y" and response != "N":
        response = input("Enter response [Y/N]: ")

    return response
```




Can you suggest any alternative solution(s)?



Programming Exercises 6.1

1. A factor is any integer which divides exactly into another integer.

For example, 5 is a factor of 20 because 5 divides exactly into 20 ($20 \div 5 = 4$) leaving no remainder.

The code below depicts two Boolean functions `isFactorV1` and `isFactorV2`. Both accept two parameters, `n1` and `n2` and return `True` if `n2` is a factor of `n1`; `False` otherwise.

```
def isFactorV1(n1, n2):
    if n1 % n2 == 0:
        return True
    else:
        return False
```

```
def isFactorV2(n1, n2):
    return n1 % n2 == 0
```



Which of these two implementations do you prefer and why?

The complete list of factors of 20 is: 1, 2, 4, 5, 10, and 20.

Write a program that lists all the positive factors of any positive integer entered by the end-user

2. The greatest common divisor (GCD) of two integers, `a` and `b` is the largest integer that divides both of them with no remainder. Study the steps shown below to find the GCD of 63 and 72:

Step 1: List the factors of the two numbers:

The factors of 63 are: 1, 3, 7, 9, 21, 63

The factors of 72 are: 1, 2, 3, 4, 6, 8, 9, 12, 18, 24, 36, 72

Step 2: Find the largest integer that these two lists share in common i.e. 9

Design and implement a function to find the GCD of any two positive integers.

- Write a program that reads a date as three integers (day, month and year) from the keyboard. The program should output the message *Valid* if the date is valid and *Invalid* otherwise. A valid date is any date spanning from 01/01/2000 up to the current date.
- Ordinal numbers are the words representing the rank of a number with respect to some position (i.e. first, second, third, etc.). Ordinal numbers are alternatively written in English with numerals and letter suffixes: 1st, 2nd, 3rd, 4th, etc.

Write a function called `ordinal` that accepts a (validated) number as input and returns a string representation of its ordinal value by concatenating an appropriate suffix to the number that was inputted.

Input	Output
1	1st
2	2nd
3	3rd
12	12th
21	21st

The table shown to the right lists some example inputs and outputs.

Hint #1: If the number ends in 11, 12 or 13 the suffix is 'th'. In all other cases the suffix can be determined from the last digit of the number from the table below.

Number	0	1	2	3	4	5	6	7	8	9
Suffix	th	st	nd	rd	th	th	th	th	th	th

Hint #2: The last two digits of a number can be extracted by using modulo 100 and the last digit of a number can be extracted by using modulo 10.

Hint #3: The solution can be arrived at by re-arranging the code below into the correct order

```

for n in range(100):
    print(ordinal(n))

def ordinal(num):
    else:
        s = suffix[num % 10]
    return str(num) + s
suffix = ['th', 'st', 'nd', 'rd', 'th', 'th', 'th', 'th', 'th', 'th']
if num % 100 in (11,12,13):
    s = 'th'

```

5. Modify the program from exercise 3 to display the date in the format *d MMM yyyy*, where,
 - *d* is the ordinal day number
 - *MMM* is the abbreviated month name (i.e. Jan – Dec)
 - *yyyy* is the four-digit year.

6. The table to the right displays a 10 step algorithm for calculating the date of Easter Sunday for any given year (given by the *pth* day of the *nth* month)

Step	Number	Divide by	Answer	Remainder (if needed)
1	$x = \text{year}$	100	$b =$	$c =$
2	$5b + c$	19	-	$a =$
3	$3(b + 25)$	4	$r =$	$s =$
4	$8(b + 11)$	25	$t =$	-
5	$19a + r - t$	30	-	$h =$
6	$a + 11h$	319	$g =$	-
7	$60(5 - s) + c$	4	$j =$	$k =$
8	$2j - k - h + g$	7	-	$m =$
9	$h - g + m + 110$	30	$n =$	$q =$
10	$q + 5 - n$	32	-	$p =$

The code on the left hand side below provides a partial implementation of the algorithm. The code on the right shows implementations of `ordinal` and `toEasterMonth`

```
def getEaster(year):
    # Step 1
    b = year // 100
    c = year % 100

    # Step 2
    a = (5*b+c) % 19

    # Step 3
    r = (3*(b+25)) // 4
    s = (3*(b+25)) % 4

    ...

    # Step 10
    p = (q+5-n) % 32

    dStr = ordinal(p)
    mStr = toEasterMonth(n)
    yStr = str(year)
    return dStr+" "+mStr+" "+yStr
```

```
def ordinal( num ):
    suffix = ['th', 'st', 'nd',
              'rd', 'th', 'th',
              'th', 'th', 'th', 'th']

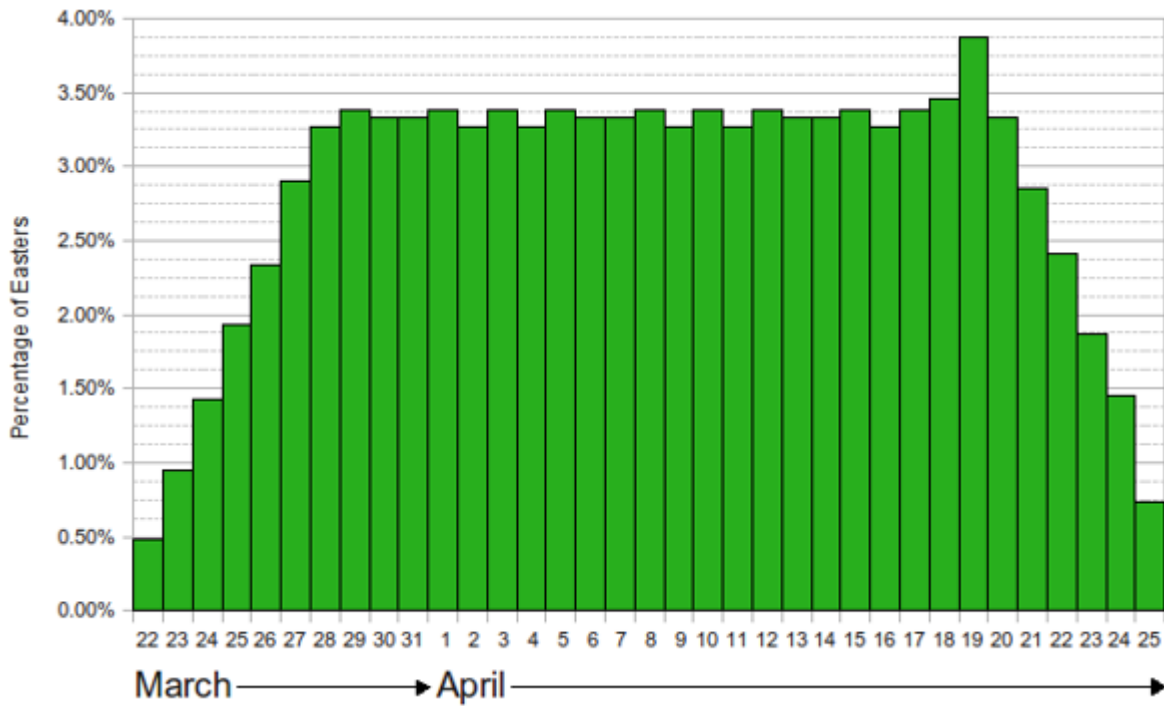
    if num % 100 in (11,12,13):
        s = 'th'
    else:
        s = suffix[num % 10]

    return str(num) + s
```

```
def toEasterMonth(mm):
    if mm == 3:
        return 'March'
    elif mm == 4:
        return 'April'
```

- a) Complete the Python implementation of the algorithm started above
- b) Use your algorithm to display the dates of Easter Sunday between 2010 and 2030
- c) Investigate the earliest and latest dates of Easter Sundays in the 21st century

d) Investigate different strategies (e.g. `plotly`, Microsoft Excel) you could use to illustrate the data in a histogram such as the one illustrated below⁹.



Distribution of the Easter Sundays over a 5,700,000 year cycle

⁹ Taken from https://en.wikipedia.org/wiki/Computus#/media/File:Easter_Distribution.svg

Recursion

Recursion is an example of a divide-and-conquer problem solving technique whereby a solution is expressed in terms of a simpler version of the same problem.

Once classic example is the factorial function which is defined as follows:

The factorial of a non-negative integer, n denoted by $n!$ (pronounced n factorial) is the product of all non-negative integers from n down to 1 and where $0!$ is accepted to have a value of 1.

$$n! = n \times (n - 1) \times (n - 2) \times \dots \times 3 \times 2 \times 1 \text{ for all } n > 0 \text{ and } 0! = 1$$

The evaluation of $5!$ for example is the product of all the integers from 5 down to 1 inclusive as shown here to the right: $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

The factorial of a non-negative integer, n can also be defined as n multiplied by the factorial of $n - 1$ (for all $n > 0$). This can be expressed mathematically as follows:

$$n! = n \times (n - 1)! \text{ for all } n > 0 \text{ and } 0! = 1$$

Notice how this definition defines factorial in terms of itself. This is the essence of recursion.



KEY POINT: A recursive function is a function that is defined in terms of itself.

The code below shows both non-recursive and recursive implementations of factorial.

```
def factorial(n):
    answer = 1
    for i in range(n, 1, -1):
        answer = answer * i
    return answer
```

non-recursive factorial

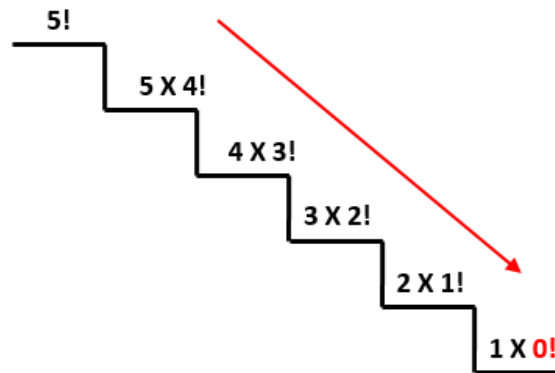
```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n-1)
```

recursive factorial

Let us consider how $5!$ would be evaluated recursively.

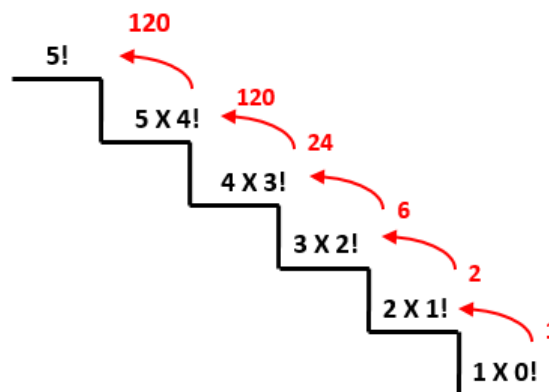
The evaluation can be considered in two stages. In the first stage (illustrated below) the number whose factorial is being sought is multiplied by the factorial of the previous number. This 'chain' of operations continues until $0!$ is reached.

- $5! = 5 \times 4!$
- $4! = 4 \times 3!$
- $3! = 3 \times 2!$
- $2! = 2 \times 1!$
- $1! = 1 \times 0!$
- $0! = 1$



In the second stage of the evaluation the chain is 'unwound' starting from the evaluation of $0!$. The result of this enables the completion of each successive step up the chain until $5!$ is reached. This is depicted as follows

- $0! = 1$
- $1 \times 0! = 1$
- $2 \times 1! = 2$
- $3 \times 2! = 6$
- $4 \times 3! = 24$
- $5 \times 4! = 120$
- $5! = 120$



The `plotly` library module can be used to create a visualisation of the factorials of the first six non-negative integers.

The chart generated by the program to the right is shown below.

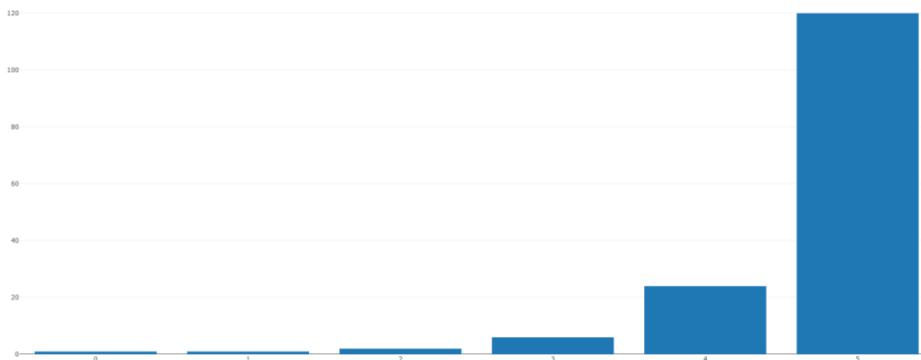
```

nums = []
factList = []

for i in range(6):
    nums.append(i)
    factList.append(factorial(i))

plotly.offline.plot({
    "data": [Bar(x=nums, y=factList)],
    "layout": Layout(title="Factorial")
}))

```



Program Context and Call stack

It is worth noting that recursion is considered to be *computationally expensive* because it requires a relatively large amount of memory to implement



KEY POINT: Recursive function are memory intensive

To understand why it is the case, it is first necessary to understand two concepts: the *program context* and the *call stack*.


The program context can be thought of as the internal program state used by Python as it executes a program. The program state consists of data such as the address of the current instruction, variables and their values. The call stack is an area of memory where Python saves the program context when a function is called. So, every time a function is called, Python makes a copy of the current program context and saves it on the call stack. When the flow of control eventually returns to the function, the contents of the call stack are loaded back in as the current program context. (This process is referred to as popping the stack.) In this way, a program can continue running with the same context it was using at the point when the call to the function was made.

The code shown here to the right is a recursive implementation of the `sumOfN` function described earlier. The purpose of the function is to return the sum of the first n integers.

```

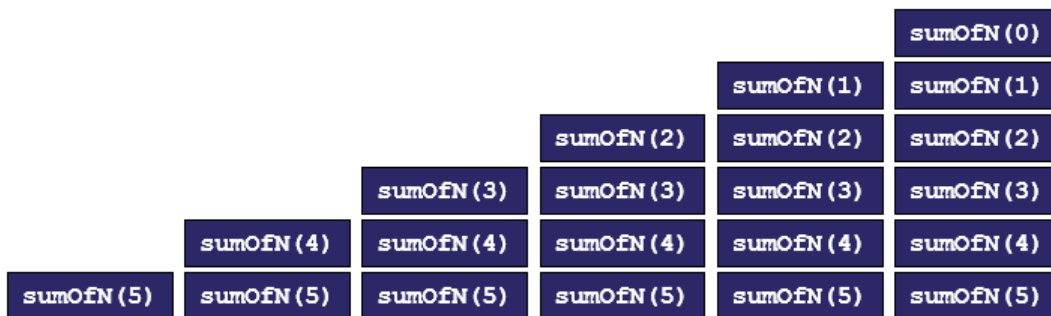
1. def sumOfN(n):
2.
3.     if n == 0:
4.         return 0
5.
6.     return n + sumOfN(n-1)
7.
8. answer = sumOfN(5)
9. print(answer)

```



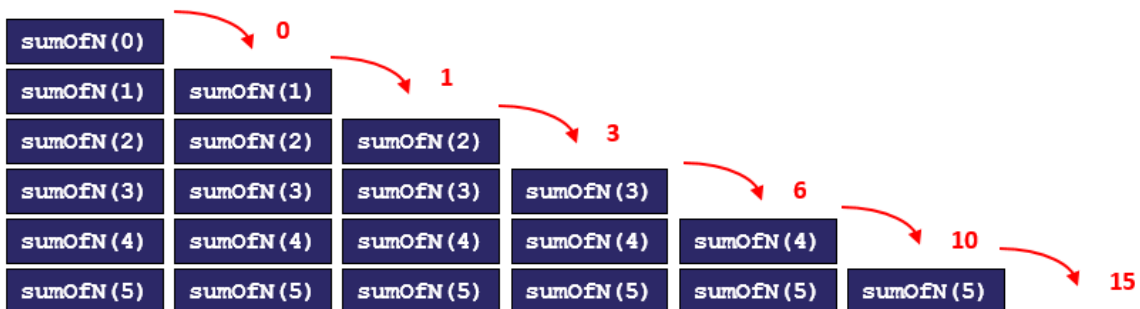
When line 8 of the program is executed the program context is saved at the bottom of the call stack – shown to left in the illustration below.

This is the program context at the point when `sumOfN(5)` was called. The call stack is progressively built up in response to the successive recursive calls to the `sumOfN` function made on line 6. This continues until `sumOfN` is called with an argument of zero.



The current program context is saved on top of the stack each time the function `sumOfN` is called

At this point the condition on line 3 evaluates to `True` and the stack begins to unwind on a last-in, first-out (LIFO) principle. The program context at the point of each call to `sumOfN` is restored in the reverse order to which it was saved. The addition operation on line 6 is completed for each call, eventually leading to an `answer` of 15.



The stack unwinds eventually leading to the `answer`, 15



Programming Exercises 6.3

1. Test your knowledge of Python by stating whether each of the following statements relating to function are `True` or `False`
 - a) The last character in a function header must always be a colon
 - b) A function definition must appear in a program before it can be called
 - c) Function names can begin with numbers
 - d) Function names cannot begin with the underscore character
 - e) A function name cannot be the same as a Python keyword
 - f) A function name can be the same as an existing Python built-in function
 - g) The function body must contain at least one statement
 - h) It is not necessary to indent every line in the function body
 - i) The statements inside a function body all must be indented to the same level
 - j) If a function contains only one statement, that statement can appear on the same line as the function header
 - k) Function names must be unique i.e. no two functions can have the same name
 - l) Every function must accept at least one argument
 - m) When a function is called, the number of arguments passed in must match the number of parameters specified in the function header
 - n) Not every function returns a value
 - o) A function call can be used as an argument to another function
 - p) The last line of every function must be a return statement
 - q) Every function must contain at least one return statement
 - r) A function can contain several return statements
 - s) A function can return multiple values
 - t) The value returned by a function must be assigned to a variable (in the calling statement)

Experiment!

It would be a good exercise to design a simple test program to verify each answer.


For example, to find out whether the last character in a function header must always be a colon simply write a short program with a function header that does not end in a colon. Run the program and see what happens.

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

```

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

```



<i>OUTPUT</i>

- a) Now key the program in and run it. Compare the predicted output with the **actual output**. Is the actual output the same as the expected output?
- b) How would the output of the program differ if lines 7 and 12 were removed? Try it.

- c) What do you think it would be a bad idea to insert a call to the function `bar` inside the function `foo`?

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

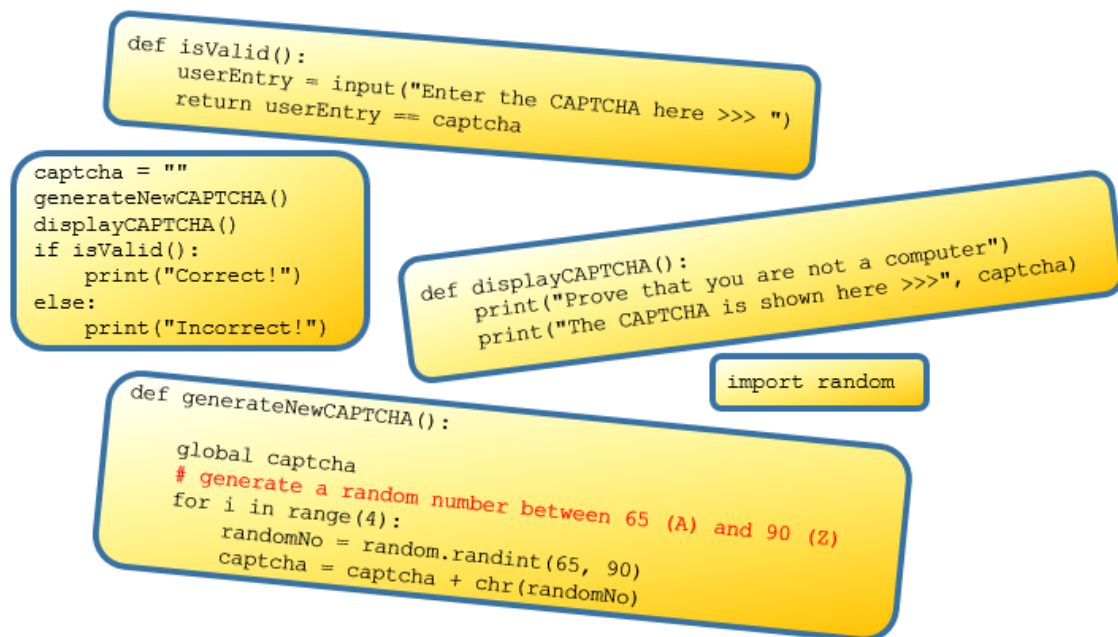
(i)
 Starting bar()
 Leaving bar()
 Starting foo()
 Leaving foo()
 Starting foobar()
 Leaving foobar()

(ii)
 Starting foobar()
 Starting foo()
 Leaving foo()
 Leaving foobar()
 Starting foo()
 Leaving foo()
 Starting bar()
 Leaving bar()

(iii)
 Starting foo()
 Starting bar()
 Leaving bar()
 Leaving foo()
 Starting bar()
 Leaving bar()
 Starting foobar()
 Starting bar()
 Leaving bar()
 Leaving foobar()

3. The acronym CAPTCHA stands for Completely Automated Public Turing Test(s) to tell Computers and Humans Apart.

Arrange the blocks of code shown below into the correct order so that it produces a program that generates a CAPTCHA, displays it to the user, prompts the user to enter this value, and displays *Correct!* if the value entered is the same as the computer generated CAPTCHA; and *Incorrect!* otherwise.



```

def isValid():
    userEntry = input("Enter the CAPTCHA here >>> ")
    return userEntry == captcha

captcha = ""
generateNewCAPTCHA()
displayCAPTCHA()
if isValid():
    print("Correct!")
else:
    print("Incorrect!")

def displayCAPTCHA():
    print("Prove that you are not a computer")
    print("The CAPTCHA is shown here >>>", captcha)

import random

def generateNewCAPTCHA():
    global captcha
    # generate a random number between 65 (A) and 90 (Z)
    for i in range(4):
        randomNo = random.randint(65, 90)
        captcha = captcha + chr(randomNo)
    
```

- modify the program so that CAPTCHAs of arbitrary lengths can be generated
 - suggest how the program might behave if the line `global captcha` was removed from the function `generateNewCAPTCHA`
-
-

4. Write a program that reads a password from the end-user and validates it according to the following constraints:
- Passwords must be between nine and twelve characters in length
 - Passwords can only contain uppercase letters, lowercase letters, digits, and special symbols.
 - The following special symbols are permitted: `_`, `+`, `-`, `*`, `/`, `!`, `?`, `&`, `@`, `^`.

5. The *Collatz sequence*¹⁰ is defined as follows for the set of positive integers:
- $$n \rightarrow n/2 \text{ (for all } n, \text{ even) and } n \rightarrow 3n + 1 \text{ (for all } n, \text{ odd)}$$

Using the rule above and starting with 6, we generate the following sequence:

$$6 \rightarrow 3 \rightarrow 10 \rightarrow 5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$$

It can be seen that this sequence contains 10 terms – starting at 6 and finishing at 1. Although it has not been proven yet, it is thought that all starting numbers finish at 1.

Implement the Collatz sequence algorithm in Python by converting the pseudo-code below.

```

Prompt the user to input a positive number – call it N
While N is not positive:
    Display an error message
    Prompt the user to input a positive number – call it N
While N is not 1:
    If N is even
        Compute N = N/2
    Else
        Compute n = 3n+1
    Output N
  
```

6. Design and implement a function that finds the sum of all the multiples of 3 or 5 below 1000. (Hint: the answer is 233,168)
7. A *perfect number* is a number whose factors (excluding the number itself) add up to the number (e.g. $6 = 3 + 2 + 1$). Write a program to list the first four perfect numbers.
8. An *amicable pair* consists of two integers for which the sum of proper divisors (the divisors excluding the number itself) of one number equals the other. The smallest pair of amicable numbers is (220, 284). They are amicable because the proper divisors of 220 are 1, 2, 4, 5, 10, 11, 20, 22, 44, 55 and 110, of which the sum is 284; and the proper divisors of 284 are 1, 2, 4, 71 and 142, of which the sum is 220.
- Implement a function `isAmicable(n1, n2)` to test whether `n1` and `n2` make an amicable pair.

¹⁰ <http://mathworld.wolfram.com/CollatzProblem.html>



BREAKOUT ACTIVITIES (Functions)

BREAKOUT 6.1: ATM System

Consider the following trimmed down version of the ATM menu first introduced in Section 1 (breakout 1.1) and developed further in Section 2 (breakout 2.2)

```
# Display an ATM menu
print("\t LCCS BANK LIMITED")
print("\t ATM Main Menu")
print("")
print("\t(1) Balance Enquiry")
print("\t(2) Cash Lodgement")
print("\t(3) Cash Withdrawal")
print("")
print("\t(0) Exit")
print("")
print("\t CHOOSE AN OPTION >> ")
```

Code to display an ATM menu

```
LCCS BANK LIMITED
ATM Main Menu

(1) Balance Enquiry
(2) Cash Lodgement
(3) Cash Withdrawal

(0) Exit

CHOOSE AN OPTION >>
```

The ATM menu

In this activity we will develop a modular program with functionality behind each of the menu options shown above. The system requirements are as follows:


- When the program is started the ATM menu is displayed and the user is prompted to choose an option
- Once the option is read it should be validated by the system. A valid option is an integer between 0 and 3 inclusive.
- If the user chooses Option 1 the system will display the account balance
The balance is stored in a variable called `balance` which is initialised to €0.00 at the start of every run
- If the user chooses Option 2 the system will prompt the user to enter an amount to lodge.
The amount entered will be added to `balance` whose value will be updated accordingly.
- If the user chooses Option 3 the system will prompt the user to enter an amount to withdraw. The amount entered will be subtracted from `balance` whose value will be updated accordingly

The system constraints are:

- Lodgements can only be made in multiples of €10
- Withdrawals are permitted only if the amount requested does not exceed the value stored in `balance`

The main program used to drive the ATM menu system is shown here.

```
# Main processing
1. balance = 0.0
2. print("Balance is %.2f" %balance)
3.
4. displayMenu()
5. menuOption = getChoice()
6. while menuOption != 0:
7.     if menuOption == 1:
8.         print("Balance is %.2f" %balance)
9.     elif menuOption == 2:
10.        amount = float(input("How much do you want to lodge? "))
11.        processLodgement(amount)
12.     elif menuOption == 3:
13.        amount = float(input("How much do you want to withdraw? "))
14.        processWithdrawal(amount)
15.
16.    displayMenu()
17.    menuOption = getChoice()
18.
19.
20. print("Thank you for banking with LCCS BANK LIMITED")
21. print("Balance is %.2f" %balance)
```




The initial task in this activity is to study the above code carefully. Use the prompts below to help you build up an understanding of the purpose of the code and how it works

a) Identify the names and purpose of the variables

b) Identify the names of the built-in functions used and the names of the user-defined functions required to make this program work.

(continued)

c) Explain how the `if` statement works (lines 7-14)

d) Explain how the `while` loop guard works (line 6)

e) Explain why lines 4 and 5 are repeated on lines 16 and 17

f) Predict what would happen if you keyed the code in and ran it

g) Key in the program (or download it from GitHub) and run it. What do you observe?

By this point it should be evident that the main program does not work as intended because the definitions (bodies) for the functions `getChoice`, `processLodgement` and `processWithdrawal` are all empty.

These empty function definitions – shown below - are known as a *stubs*. Stubs are placeholder functions used as part of the program construction process so that programs can be run without any syntax errors.

```
def getChoice():  
    return 0  
  
def processLodgement(lodgeAmount):  
    pass  
  
def processWithdrawal(withdrawalAmount):  
    return
```

h) Explain the use of `pass`¹¹ and `return` in the above code.

i) Why do you think the stub `getChoice` returns zero?

¹¹ `pass` is a Python keyword. When it is executed, nothing happens. It is useful as a placeholder when a statement is required syntactically, but no code needs to be executed.



The next stage of this activity is to incorporate full implementation of the four functions (shown below) into the main program. Before you do this you should first study each function carefully and use the space provided to explain how each one works

<pre># A function to process withdrawals def processWithdrawal(withdrawalAmount): global balance if (withdrawalAmount > balance): print("Insufficient Funds.") else: balance = balance - withdrawalAmount print("Please take your cash.") return</pre>	<pre># A function to process lodgements def processLodgement (lodgeAmount): global balance if (lodgeAmount%10 == 0): balance = balance + lodgeAmount else: print("Amounts must be multiples of 10.") return</pre>
<pre># A function to read and validate the menu option def getChoice(): validChoice = False while not validChoice: choice = input("\t CHOOSE AN OPTION >> ") if choice.isdigit(): choice = int(choice) if (choice >= 0) and (choice < 4): validChoice = True return choice</pre>	<pre># A function to display the menu def displayMenu(): print("\t LCCS BANK LIMITED") print("\t ATM Main Menu") print("") print("\t(1) Balance Enquiry") print("\t(2) Cash Lodgement") print("\t(3) Cash Withdrawal") print("") print("\t(0) Exit") print("")</pre>

Key in (or download) the four functions and make sure they run without any syntax errors. Experiment by re-arranging the functions into different orders. After every change make sure to test your program to make sure it still works properly

Test the system with some 'normal' use cases e.g. use the system to lodge €100 or withdraw €30. What's the balance before and after the transaction?



Experiment!
Investigate what happens when you attempt the following

a) try to lodge €35?

b) try to withdraw a negative amount?

c) try to withdraw an amount greater than the balance?

d) try to withdraw an amount equal to the balance?



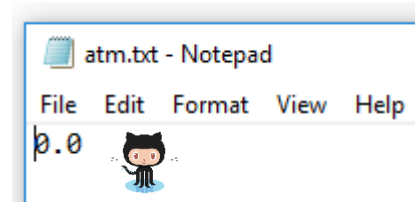
Evaluate.
Identify any features of the solution that you like/dislike.

Suggested Activities (Modifications)¹²

1. Modify the program so that the maximum amount that can be withdrawn in any one transaction is €200.

2. Modify the program so that the account balance can be read from and stored in a file. The following hints are designed to help

- a) You will need to create the initial file - call it `atm.txt` - to store the balance. Insert the value 0.0 the very first time it is created Initially



- b) The function shown below reads the value from the file into the variable `balance`.

```
# A function to read the balance from the file
def readATMFile():
    global balance

    bankFile = open("atm.txt", "r")
    balance = float(bankFile.readline())
    bankFile.close()
```

- c) The function shown below writes the contents of the variable `balance` to the file. Notice the use of the `with` keyword as an alternative way to reference file objects. The use of `with` is considered good practice.

```
# A function to write the balance to the file
def writeATMFile():
    global balance

    with open('atm.txt') as bankFile:
        bankFile.write(str(balance))
```

- d) Finally, you will need to insert the calls to these functions at the start and end of the main program block presented earlier.

3. Design and integrate a Personal Identification Number (PIN) security layer into the system. Feel free to come up with your own requirements but here's some ideas to get you started.

- The customer is given three attempts to enter the correct PIN
- A valid PIN is any four-digit number (i.e. a value between 0000 and 9999)
- The initial PIN to enter the system should be read from a file (e.g. `atm.txt`)
- The menu system should be extended to allow the user to change the PIN

¹² Before you attempt these activities you will need to download <https://github.com/pdst-lccs/lccs-python/blob/master/Section%206%20-%20Modular%20Programming/Breakout%206.1%20ATM%20System/atm%20-%20V1%20-%20initial%20solution.py>

- Global variables are variables that are visible to blocks of code outside the scope of where they are declared. In this example, the variable `balance` is *global*. This means that the value of `balance` is visible inside every function which uses the `global` keyword.

The use of *global* variables is generally considered to be poor programming practice. Remove the need for the `global` keyword in the system.

Hint: Consider how `balance` could be passed as an argument into the functions that need it, and returned out of functions that change it.

- The system requires the user to enter the lodgement/withdrawal amounts. What are the drawbacks of this? Can you think of any alternative ways these amounts could be captured by the system? Design a solution.



How has this specific activity extended your thinking in relation to the role of functions in designing a user interface?



Based on what you have learned by completing this activity, use the code below as an inspiration to develop another activity suitable for use in the LCCS classroom.

```
1 print("\nSelect the action that you want to perform:")
2 print("  1. Look up a phone number.")
3 print("  2. Add or change a phone number.")
4 print("  3. Remove an entry from your phone directory.")
5 print("  4. List the entire phone directory.")
6 print("  5. Exit from the program.")
7 print("Enter action number (1-5): ")
8
```

The prompts here may be used as a guide:

- Start off by listing the requirement and any constraints.
- What prior knowledge and skills are required to meet the requirements?
- What initial scaffolding would you provide to students?
- What areas would you ask students to investigate?
- How might students modify any code you provide?
- What extended activities based on the code would ask students to complete?
- With what reflection tasks should students be presented?

BREAKOUT 6.2: Summing Numbers

Earlier in this chapter we presented a function `sumOfN` to calculate and return the sum of the first n non-negative integers. The function worked by maintaining a running total as it iterated over every number up to and including n . The solution is considered relatively expensive because it requires n iterations (and n computations) for n numbers.

As a young schoolboy, the German mathematician Carl Friedrich Gauss (1777-1855) is reputed to have devised a much simpler and more elegant solution that cost just one simple calculation and did not require the use of iteration. The formula is developed as follows¹³:

We let S_1 be the sum of the integers from 1 *up* to n :

$$S_1 = 1 + 2 + 3 + \dots + (n - 2) + (n - 1) + n$$

We let S_2 be the sum of the integers from n *down* to 1

$$S_2 = n + (n - 1) + (n - 2) + \dots + 3 + 2 + 1$$

Since both sums are the same we can safely say $S_1 = S_2 = S$

The two sums can be added as follows to give $2S$

$$\begin{array}{r} S = 1 + 2 + 3 + \dots + (n - 2) + (n - 1) + n \\ S = n + (n - 1) + (n - 2) + \dots + 3 + 2 + 1 \\ \hline 2S = (n + 1) + (n + 1) + (n + 1) + \dots + (n + 1) + (n + 1) + (n + 1) \end{array}$$

We have $(n + 1)$ repeated n times i.e. $n \times (n + 1)$

The required sum S is arrived at by dividing both sides by 2 to give Gauss's formula:

$$S = \frac{n \times (n + 1)}{2}$$

¹³ See <https://brilliant.org/wiki/sum-of-n-n2-or-n3/>


Suggested Activities.¹⁴

1. A Python implementation of Gauss's formula is shown here to the right.

Write two additional functions to find the sum of the squares and the sum of the cubes of the first n non-negative integers using the respective formulae shown below

```
# Sum the first n integers
def gaussSumOfN(n):
    return n*(n+1)/2

result = gaussSumOfN(100)
```



$$\sum_{k=1}^n k^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{k=1}^n k^3 = \frac{n^2(n+1)^2}{4}$$

- a) Explain the subtle difference between finding the sum of the first n even numbers and finding the sum of the all even numbers less than n . Use the two listings below to help.

```
def sumOfEvenV1(n):
    sum = 0
    number = 0
    while number <= n:
        sum = sum + number
        number = number + 2

    return sum
```

```
def sumOfEvenV2(n):
    sum = 0
    number = 2
    count = 0
    while count < n:
        sum = sum + number
        number = number + 2
        count = count + 1

    return sum
```

- b) Exploit the pattern $2 + 4 + 6 + 8 + \dots = 2(1 + 2 + 3 + 4 + \dots)$ to devise a formula to solve the above problem. Implement your solution.
- c) Define a function that sums all the positive integers from x to y where both x and y are two numbers entered by the end-user. For example, if the end-user entered 8 and 13 the program would compute and display the result of $8 + 9 + 10 + 11 + 12 + 13$.
(Hint: *Sum of 8 to 13 = Sum of 1 to 13 minus the sum of 1 to 7*)

¹⁴ Refer to <https://betterexplained.com/articles/techniques-for-adding-the-numbers-1-to-100/> for interesting background reading



Reflect. How could the derivation of Gauss’s formula and/or the illustrations shown below be used explain core computational thinking concepts in the LCCS classroom?

$$1 + 2 + 3 = 6$$

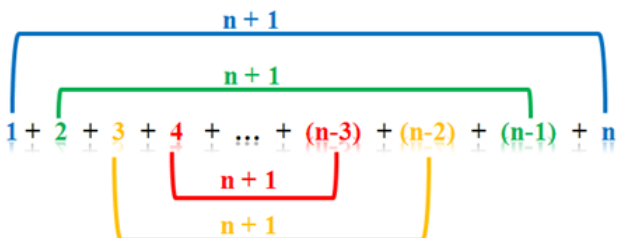
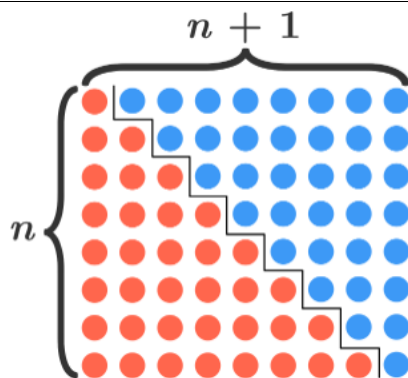
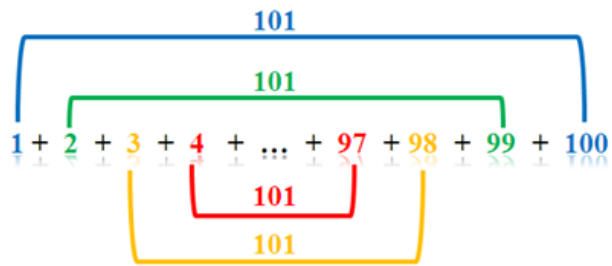
$$1 + 2 + 3 + 4 = 10$$

$$1 + 2 + 3 + 4 + 5 = 15$$

$$1 + 2 + 3 + 4 + 5 + 6 = 21$$

$$1 + 2 + 3 + 4 + 5 + 6 + 7 = 28$$

$$1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 = 36$$



Further Activities

1. Write a program to sum the first n odd numbers
(Write another program to sum all the odd numbers up to n).
2. The reciprocal of a number x is denoted by $\frac{1}{x}$ e.g. $\frac{1}{5}$ is the reciprocal of 5.
Define a function to sum the reciprocal of the first 10 positive integers i.e.

$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7} + \frac{1}{8} + \frac{1}{9} + \frac{1}{10}$$

3. Implement a function that finds an approximation for π using the formula provided

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \dots$$

4. Write a program that uses the formula shown to estimate a value for Euler’s constant, e

$$\sum_{n=0}^{\infty} \frac{1}{n!} = \frac{1}{1} + \frac{1}{1} + \frac{1}{2} + \frac{1}{6} + \frac{1}{24} + \frac{1}{120} + \dots = e.$$



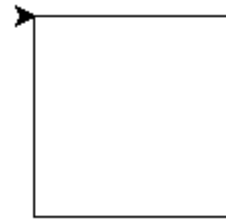
Explain how recursion could be used to complete any of the tasks covered in this section

BREAKOUT 6.3: Turtle Graphics and Functions – PART I

Consider the short program below to draw a square of 100 units

```
from turtle import *

# Draw a square
forward(100)
right(90)
forward(100)
right(90)
forward(100)
right(90)
forward(100)
right(90)
```



Suggested Activities

1. Key in (or copy+paste from GitHub) the code and run with it. Investigate what happens when you change the values (i.e. 100 and 90). What conclusions about the meanings of these values can you make?

2. Implement a function called `drawSquare` and modify the above code to use it.

```
def drawSquare():
```

3. Modify `drawSquare` so that it accepts the square's dimension as a parameter

```
def drawSquare(dimension):
```

4. The implementation of `drawSquare` shown here exploits the fact that a square is a special kind of rectangle (abstraction). Implement `drawRectangle`.

```
def drawSquare(dimension):
    drawRectangle(dimension, dimension)
```

5. Design and implement a function to draw a *n-sided polygon*. (Hint: study the three function definitions shown carefully – look for patterns and generalise.)

```
def drawSquare(size):
    for side in range(4):
        forward(size)
        left(90)
```

```
def drawTriangle(size):
    for side in range(3):
        forward(size)
        left(120)
```

```
def drawHexagon(size):
    for side in range(6):
        forward(size)
        left(60)
```

BREAKOUT 6.3: Turtle Graphics and Functions – PART II

Study the two listings below carefully and answer the questions that follow.


```
from turtle import *

# set the appearance of the turtle
hideturtle()
color('red')
pensize(5)

# Draw a 30 degree angle
backward(100)
left(30)
forward(100)

# Position the pen
penup()
setpos(200, 0)
setheading(0)
pendown()

# Draw a 60 degree angle
backward(100)
left(60)
forward(150)
```



Listing A


```
from turtle import *

# set the appearance of the turtle
def setAppearance():
    hideturtle()
    color('red')
    pensize(5)

# Draw angle
def drawAngle(size):
    backward(100)
    left(size)
    forward(100)

# Position the pen
def setPenPosition():
    penup()
    setpos(200, 0)
    setheading(0)
    pendown()

# Main program starts here
setAppearance()
drawAngle(30)
setPenPosition()
drawAngle(60)
```



Listing B



Compare and contrast the two listings. Which do you think is better and why? Describe any limitations of your preferred listing and explain how these might be overcome.

Suggested Activities

1. Download the code for Listing B, predict what it does and run it. What does the code do?

2. Modify the function `drawAngle` to accept the lengths of each angle arm as shown in the function header here.

```
def drawAngle(size, arm1Len, arm2Len):
```

3. Modify the function `setPenPosition` to accept the (x,y) co-ordinates as shown in the function header here.

```
def setPenPosition(x, y):
```

4. Modify the implementation of `drawAngle` so that it accepts the co-ordinates (as well as the angle size and arm lengths) of the position at which to place the angle.

```
def drawAngle(x, y, size, arm1Len, arm2Len):
```

5. Once you have the previous task complete you should evaluate the design decision to change the definition of `drawAngle`.
What are the wider implications to the rest of a program of changing a function header?

6. Investigate whether Python would allow the following two function definitions in the same program. Would this be a useful feature?

```
def drawAngle(size, arm1Len, arm2Len):
```

```
def drawAngle(x, y, size, arm1Len, arm2Len):
```



Reflection.

What were your main thoughts as you engaged with this activity?



Describe any connections this activity allowed you to make between computational thinking and functions.

Further Activities

Code refactoring is the process of restructuring computer code without changing its existing functionality. Code is usually refactored in order to improve readability and maintainability.

The functions defined in the code below each draw a corresponding shape shown to the right. Each function relies on the turtle being oriented to the right (i.e. in an eastward direction) in order to work properly. This is achieved with the call `setheading(0)`.

```
from turtle import *

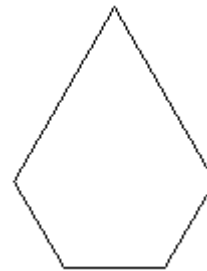
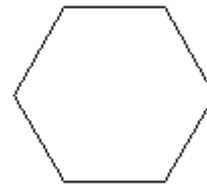
def drawShape1():
    for i in range(6):
        forward(50)
        left(60)

def drawShape2():
    right(60)
    for i in range(3):
        forward(50)
        left(60)
    forward(100)
    left(120)
    forward(100)

def drawShape3():
    right(60)
    for i in range(3):
        forward(50)
        left(60)
    left(60)
    forward(100)

def drawShape4():
    right(60)
    for i in range(3):
        forward(50)
        left(60)

hideturtle()
# Insert your code under here ...
setheading(0)
drawShape1()
#drawShape2()
#drawShape3()
#drawShape4()
```



1. The task here is to look for a common pattern in the shape (or code) and exploit this pattern to refactor the code without changing any functionality.

Hint: You will need to write a separate function that just draws the part of the shape that is common to all four shapes.

2. Read the code below carefully and key it in.
- a) Write the necessary code to call the functions defined so that each of the three shapes shown to the right are drawn - all lines are 50 units in length.

```

from turtle import *

def movePenTo(x, y):
    penup()
    setpos(x, y)
    pendown()

def drawVertLineUp():
    setheading(90)
    forward(50)

def drawVertLineDown():
    setheading(270)
    forward(50)

def drawHorizLineRight():
    movePenTo(xcor()+5, ycor())
    setheading(0)
    forward(50)
    movePenTo(xcor()+5, ycor())

def drawHorizLineLeft():
    movePenTo(xcor()-5, ycor())
    setheading(180)
    forward(50)
    movePenTo(xcor()-5, ycor())

hideturtle()
# Insert your code under here .

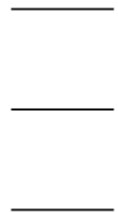
```



- b) Identify any repeating patterns in the above shapes and exploit this pattern to refactor the code without changing any functionality.
- c) Define three functions `draw2`, `draw3` and `draw9` in terms of the functions defined in the above code to display the digits as illustrated below.



draw2



draw3



draw9

BREAKOUT 6.4: Using check digits to verify codes

A **check digit** is a number whose purpose is to trap potential errors made by humans when manually entering a code (e.g. barcodes, ISBNs) to a system. It consists of a single digit – usually the rightmost digit - whose value is computed from the other digits in the number. The check digit of the barcode shown here to the right is 4



EAN-8 format barcode

Barcode check digit verification algorithms generally work by calculating a total using the formula shown below. If the total is evenly divisible by 10 then the check digit is deemed to be valid.

$$total = (sum\ of\ all\ the\ digits\ in\ the\ even\ positions) + 3 \times (sum\ of\ all\ the\ digits\ in\ the\ odd\ positions)$$

The digits are read from right to left – so the rightmost digit is always taken to be the first even-positioned digit. The following table shows the positions and the corresponding digit for the barcode shown at the top of the page.

Digit Position	0	1	2	3	4	5	6	7
Digit	4	5	2	3	3	8	5	6

The check digit of this barcode number can be verified as follows:

1. Add all the digits in the even-numbered positions (zeroth, second, fourth etc.)
2. Add the digits in the odd-numbered positions (first, third, fifth, etc.) together and multiply by three.
3. Add the two sub-totals.
4. Take the remainder of the total divided by 10 (modulo operation)

$$4 + 2 + 3 + 5 = 14$$

$$3 \times (5 + 3 + 8 + 6) = 3 \times 22 = 66$$

$$14 + 66 = 80$$

$$80 \% 10 = 0$$

The remainder after dividing by ten is zero (i.e. $80 \equiv 0 \pmod{10}$). Therefore, the check digit is correct.

The European Article Number or EAN (aka International Article Number) is a worldwide standard that describes the format for numbers that appear under the barcodes used to identify retail products. Two of the most commonly used EAN standards are the eight digit EAN-8 and thirteen digit EAN-13 – illustrated below.



Barcode using EAN-8



Barcode using EAN-13



Use the algorithm described on the previous page to verify that both of the above barcodes are valid



Outline any programming strategies you could use to extract the even/odd digits as part of a Python implementation of the check digit verification algorithm?

Developing an algorithm to extract digit from a number

The three listings below extract the individual digits from any two- digit, three- digit and four- digit number respectively

In all of these examples the rightmost digit is always stored in the variable `d1`.

The value is the remainder after dividing whatever number was entered by 10 (e.g.

$83 \% 10 = 3$, $835 \% 10 = 5$ etc.).

```
n = int(input("Enter a 2 digit number: "))
d1 = n%10      # %10 extracts the last digit
d2 = n//10    # //10 chops off the last digit
print(d1, d2)
```



Sample Input: 83

Output: 3 8

In the above example the leftmost digit is extracted by dividing by 10 (i.e. $83 // 10 = 8$). This is stored in the variable `d2`.

```
n = int(input("Enter a 3 digit number: "))
d1 = n%10      # %10 extracts the final digit
d2 = (n//10)%10 # //10 chops off the last digit
d3 = (n//100)%10 # //100 chops off the last two digits
print(d1, d2, d3)
```



Sample Input: 835

Output: 5 3 8

Here, the middle digit is extracted in two steps – first, divide by 10 to leave the first two digits (i.e. $835 // 10 = 83$) and then remainder 10 (i.e. $83 \% 10 = 3$).

This is stored in the variable `d2`.

The leftmost digit is extracted by dividing by 100 (e.g. $835 // 100 = 8$). This is stored in the variable `d3`. (Note that taking the remainder 10 of does not affect the value)

In the next (and final) example we examine the code to extract the individual digits of any four-digit number - `d4`, `d3`, `d2` and `d1`.

```
n = int(input("Enter a 4 digit number: "))
d1 = n%10      # %10 extracts the final digit
d2 = (n//10)%10 # //10 chops off the last digit
d3 = (n//100)%10 # //100 chops off the last two digits
d4 = (n//1000)%10 # //1000 chops off the last three digits
print(d1, d2, d3, d4)
```



Sample Input: 8352

Output: 2 5 3 8

Let's look at the process for extracting the second digit in from the right, d_2 . The sample input is 8352 and we wish to extract the 5. The procedure is to divide by 10 – this gives 835 - and then use remainder 10 on this to extract the final digit.

A similar procedure can be used to extract the third digit in from the right (d_3) i.e. extract the 3 from 8352. This time we divide by 100 – this gives 83 - and then use remainder 10 to extract the final digit.

Finally the leftmost digit, d_4 , can be extracted simply by dividing by 1000 i.e. $8352 // 1000 = 8$. Applying remainder 10 to this does not change the value.



Describe any patterns you recognise emerging from the three examples above?



Devise an algorithm to extract the i^{th} digit from any number n . (Take the rightmost digit to be at position 0.) Implement your solution in a Python function – header defined as follows:

```
def extractDigit(i, n):
```



Explain how the remainder (modulus) operator could be used to ensure a randomly generated number is within a specific range

Suggested Activities

These activities are based on the short program below which implements an algorithm to validate an EAN-8 barcode number

```
def extractDigit(i, n):
    return (n//pow(10, i))%10

def isValidEAN8(n):

    sum = 0
    for i in range(9):
        digit = extractDigit(i, n)

        if i%2 == 0:
            sum = sum + digit
        else:
            sum = sum + digit*3

    return sum%10 == 0

print(isValidEAN8(53912343)) # true
```

1. Key in (or copy+paste from GitHub) the above code and make sure it runs without any syntax errors.
2. Test the program with a range of valid and invalid EANs
3. Take a close look at the function `extractDigit`. What would happen if an *out of range index* was passed into it?

For example what would happen if the following calls were made to the function - `extractDigit(-1, 869)` or `extractDigit(3, 869)`?

4. Suggest ways by which any problem(s) identified in the previous question could be fixed and discuss whether any of these solutions need to be implemented.
5. Extend the code with a function to check the validity of an EAN-13 number
6. Implement a function `isValidEAN` that works for both EAN-8 and EAN-13 numbers.
Hint you will need to do a range test to determine the size of the number
7. Modify the code so that it could work using string representation of the EAN (as opposed to a numeric representation).

Test your program with the call: `print(isValidEAN8("53912343"))`

8. Design (and implement) a program to generate a check digit for an EAN-13 barcode (i.e. given the first 12 digits compute the 13th)
9. Write a recursive function to sum the individual digits of a number



Reflection.

***Describe your experience of engaging with the tasks in this activity.
Consider the following prompts for a learner's perspective***

When I saw a problem for the first time what was my thinking?

What computational thinking skills, if any, did I employ?

What worked well for me? What didn't work so well?

Were there any limitations to any of my solutions?

What programming skills did I improve/learn?



Looking back at the tasks in this activity (as a teacher) what programming constructs would it be necessary for students to understand before engaging in a similar task.

Further Activities (ISBNs and Credit Cards)

1. An International Standard Book Number (ISBN) is a unique number used to identify books worldwide. Before 2007 ISBNs were made up of 10 digits and this was extended to 13 digits from 1 January 2007. The former standard is known as ISBN-10 and the latter is known as ISBN-13.



Use the information provided below to derive and implement an algorithm to validate any ISBN-10

ISBN-10: 1-350-05711-8

$checksum = (1 \times 10) + (3 \times 9) + (5 \times 8) + (0 \times 7) + (0 \times 6) + (5 \times 5) + (7 \times 4) + (1 \times 3) + (1 \times 2) + (8 \times 1)$

$checksum \% 11$ must be zero

2. Just like barcodes and ISBNs, credit card numbers contain several pieces of information for performing validity tests. For example, Visa card numbers are always 16 digits long and always begin with 4. A valid Visa card number also passes a digit-sum test known as the Luhn checksum algorithm. Luhn's algorithm states that if you sum the digits of the number in a certain way, the total sum must be a multiple of 10 for a valid Visa number.

Systems that accept credit cards perform a Luhn test before contacting the credit card company for final verification. This lets the company weed out many fake or incorrect credit card numbers.

The algorithm for summing the digits is the following. For digits at even indexes (the 0th digit, 2nd digit, etc.), simply add that digit to the cumulative sum. For digits at odd indexes (index 1, 3, etc.), double the digit's value, then if that doubled value is more than 10, add its digits together to make a number that is smaller than 10, then add this result into the sum.



Use the information provided below to derive and implement Luhn's algorithm to validate any credit card

Credit Card Number: 4408 0412 3456 7893

$checksum = (8) + 4 + (0) + 8 + (0) + 4 + (2) + 2 + (6) + 4 + (1 + 0) + 6 + (1 + 4) + 8 + (1 + 8) + 3$

$checksum \% 10$ must be zero

NOTES

Section 7

Dictionaries

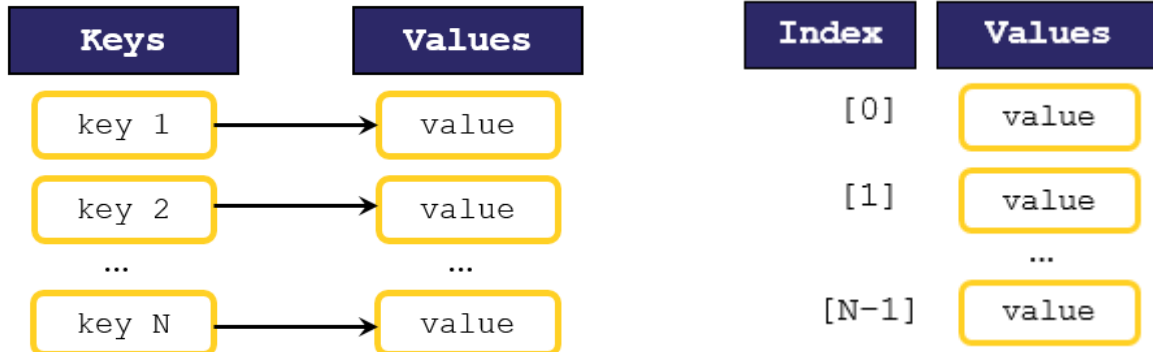
Introduction

In plain English a dictionary is something that is used to find the meaning or translation of a word. In Python a dictionary is a data structure which can be used to store values that can be looked up and retrieved using a unique identifier known as a key.

Python dictionaries are very similar to Python lists but, as we will soon see, there are some important differences.

Like lists, dictionaries are useful because they provide a means by which a collection of data can be manipulated using a single variable. Each element in a dictionary consists of two parts – a **key**, which must be unique, and an associated **value**. These are commonly referred to as **key-value pairs** (and also name-value pairs).

The graphic on the left below depicts the mapping between keys and their associated values. This can be contrasted with the graphic on the right which depicts a Python list of length N.



A dictionary with N elements. Each element is referenced by a unique key which is usually a string or an integer

A list with N elements. Each element is referenced by a zero based positional offset known as an index

Recall from Chapter 4 that lists are ordered data structures. This means that Python maintains the relative ordering of every element in a list. Because the list index is a positional offset position, operations such as slicing and concatenation make sense for lists.

Dictionaries on the other hand are unordered data structures. Python does not maintain the order of dictionary elements. This is because elements are retrieved using a key (as opposed to an index). There is no guarantee that dictionary elements are maintained in the

same order as they were created. Consequently, operations that depend on order such as slicing and concatenation are not supported for dictionaries.



KEY POINT: A dictionary can be thought of as an unordered list which uses a key to retrieve elements instead of an index.

Dictionaries are useful for storing a list of values when the values can be identified by some string or integer (i.e. the key). Keys must be unique and can be of any immutable datatype (e.g. string, integer) – and values can be of any datatype (simple or compound).

Dictionary Definitions – some examples


The examples below are all given in the form of assignment statements in which a dictionary appears to the right of the assignment operator and the variable to which the dictionary is being assigned appears on the left.

Example 1

In this first example, we define a dictionary called `glossary`. The dictionary shown below contains four elements and each element is made up of two parts – a key and a value – delimited by a colon. The name of the first element in this dictionary is *Analyse* and the value of this element is *to study or examine something in detail*.

```

glossary = {
    "Analyse" : "to study or examine something in detail",
    "Annotate" : "to add brief notes of explanation to a diagram or graph",
    "Appraise" : "to evaluate, judge or consider text or a piece of work",
    "Appreciate" : "to recognise the meaning or have a practical understanding of",
}
  
```



In this dictionary each individual word is the key. The key can be used to lookup the value - in this case the meaning of the word.

Note that in each of the examples in this section the dictionary data is enclosed by curly braces. The opening brace, (`{`), tells Python that this is the start of a dictionary definition and the closing brace, (`}`), signals the end of the dictionary definition.




KEY POINT: The elements of a dictionary are enclosed by curly braces; each element in a dictionary is a **key-value** pair; the key and the value must be separated by a colon.

Example 2

In this example the dictionary comprises six elements. Note that dictionary elements are separated by commas and also that a comma is allowed at the end of the last element.

```
capitals = {
    "Ireland" : "Dublin",
    "Scotland" : "Edinburgh",
    "England" : "London",
    "Wales" : "Cardiff",
    "France" : "Paris",
    "Italy" : "Rome",
}
```



The individual elements provide a mapping from a unique country name (e.g. Ireland) to a capital city (e.g. Dublin).

The above dictionary could be used to look up a capital city for a given country.

As was the case with the last example the datatype of the names and values are all strings and the elements are all enclosed in curly braces.




KEY POINT: No two dictionary elements can have the same key i.e. the keys *must* be unique. Values however, can be duplicated.

Example3

Here, the dictionary `numbers` can be used to lookup the Irish word for each of the numbers from one to ten.

```
numbers = {
    1 : "aon",
    2 : "dó",
    3 : "trí",
    4 : "ceathair",
    5 : "cúig",
    6 : "sé",
    7 : "seacht",
    8 : "ocht",
    9 : "naoi",
    10: "deich"
}
```



In this example, the datatype of the keys are all integers and the datatype of the corresponding values are all string.



KEY POINT: The datatype of a dictionary key *must* be immutable (e.g. string, integer) but values can be of *any* datatype (including lists and other dictionaries).

STUDENT TIP

Consider using a dictionary in situations when you need to associate a unique identifier (key) with one or more variables (value)

Example 4

This example defines a dictionary called `capacity` which stores the maximum number of people that seven well known sports stadia can safely accommodate.

Note in this case the datatype of the keys is string and the datatype of the values is integer.

```

capacity = {
    "Croke Park" : 82300,
    "Aviva Stadium" : 51700,
    "Páirc Uí Caoimh" : 45000,
    "Thomond Park" : 25600,
    "Pearsa Stadium" : 34000,
    "Kingspan Breffni" : 26000,
    "Casement Park" : 32600,
}

```

This dictionary could be used to look up the capacity of a given stadium.

Example 5

In examples 1-4 the keys and values store the *same kind of information* i.e.

- in example 1 the key is a word and the lookup value is the meaning of that word
- in example 2 the key is a country and the values is its capital
- in example 3 the key is a number and the value is its Irish translation
- in example 4 the key is the name of a stadium and the value is its capacity

It is quite common for keys to be unrelated to one another. Notice how in the following examples the keys all refer to different things and also the mixture of datatypes in the values.

```

student = {
    'id': "123456",
    'name': "Mary Bloggs",
    'gender' : "Female",
    'age' : 17,
    'gpa' : 76.2,
    'ambitious': True,
    'grant' : False,
    'ambitious': True,
}

```

This dictionary is used to hold student details. Note the mixture of datatypes in the values.

```

nctBooking = {
    'car_reg': "131 CN 6439",
    'date': "10/12/2018",
    'time': "21:20",
    'centre': "Cavan",
    'fee': 78.00
}

```

nctBooking is a dictionary used to store the booking details for a National Car Test (NCT)

Dictionaries like these two are useful for representing **structured data** i.e. data that is organised in a tabular format in such a manner that values can be identified by unique names (e.g. field names in a database, column headers in a spreadsheet/csv file)

It is worth noting that dictionaries are referred to as records/structures in other programming languages.

Creating Dictionaries - syntax

Python supports several different syntaxes for creating dictionaries – so far we have looked at just one i.e. enclosed in curly braces with key-value pairs delimited by colons and separated by commas. The general form of this syntax is shown below.

```

<dict-variable> = {
    <key 1> : <value>,
    <key 2> : <value>,
    ...
    <key N> : <value>,
}

```

Note the use of:

- curly braces to enclose the dictionary elements
- the colon to delimit the key-value pairs and
- the comma to separate elements


Another way to create dictionaries is to use the built-in function, `dict`. There are several variations on the use of `dict` but for the sake of brevity we will look at just two.

In this first variation (shown below) the `dict` function is used to create a dictionary called `abbr1` with three key-value pairs. The general form of this syntax is shown to the right. Note the similarity to the syntax of the earlier examples – basically, this syntax just passes the entire dictionary (including the curly braces) as an argument into the `dict` function.

```

abbr1 = dict( {
    "OMG" : "Oh My God!",
    "LOL" : "Laugh out Loud",
    "IMHO" : "In My Humble Opinion",
})

```



```

<dict-variable> = dict( {
    <key 1> : <value>,
    <key 2> : <value>,
    ...
    <key N> : <value>,
})


```

In this next example three keyword arguments are passed into `dict`. Each argument is *assigned* an associated value. This syntax requires that each key is a valid Python identifier i.e. it cannot begin with a digit, must be made up of alphanumeric characters or underscore etc. Note the absence of curly braces in this syntax.

```

abbr2 = dict(
    OMG = "Oh My God!",
    LOL = "Laugh out Loud",
    IMHO = "In My Humble Opinion",
)

```



```

<dict-variable> = dict(
    <key 1> = <value>,
    <key 2> = <value>,
    ...
    <key N> = <value>,
)

```

When created in this way, the datatype of the dictionary keys is always a string.

The dictionaries resulting from both examples, `abbr1` and `abbr2` are identical i.e.

```
{'OMG': 'Oh My God!', 'LOL': 'Laugh out Loud', 'IMHO': 'In My Humble Opinion'}
```

Empty Dictionaries

In practice data is usually not hardcoded into programs as illustrated in the examples used thus far. Rather, data typically enters a running program from some external source such as the end-user, file(s) or a database. In such cases it can be useful to start off with an empty dictionary. The three statements below all define an empty dictionary called `d`.

```
d = {}
```

```
d = dict({})
```

```
d = dict()
```

As a program runs, data can be added to, or removed from the dictionary in accordance with the needs of the underlying computational model.



What have you learned about dictionaries from the preceding examples?



What one question about dictionaries would you like to have answered right now?

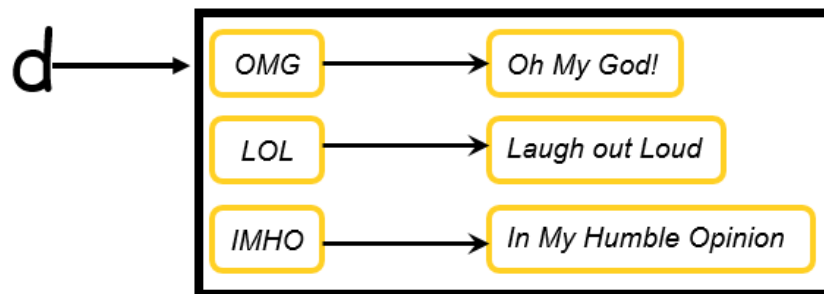


Define a dictionary of your choice to contain five elements.

Indexing Dictionaries

Recall from earlier that *indexing* is a technique used to access/retrieve the elements of strings (section 3) and lists (section 4). Dictionary values can also be indexed but, unlike strings and lists which both use a zero based integer as a positional offset to access elements, dictionaries use the key as the index.

Let's say we have a dictionary called `d` defined with key-value pairs depicted as shown here.



The code below displays the three values stored in the above dictionary:

```

print(d['OMG'])
print(d['LOL'])
print(d['IMHO'])
  
```



```

Oh My God!
Laugh out Loud
In My Humble Opinion
  
```

Dictionary values are accessed by using the keys as the index

The output displayed by the code

The general syntax to access elements of a dictionary is:

`<dict-name>[key]`

where, `dict-name` is the name of the dictionary and `key` is the lookup value to use in order to retrieve the required element. Note that the key may be a string or an integer.



KEY POINT: Dictionary values are accessed by using their key as the index (i.e. in square brackets)

The example below demonstrates that dictionary values can be accessed in any order. The output of each `print` statement is shown to the right.

```

print(d['IMHO'])
print(d['OMG'])
print(d['LOL'])
  
```

```

In My Humble Opinion
Oh My God!
Laugh out Loud
  
```



SYNTAX CHECK #1:


Attempts to access a dictionary using a key that does not exist result in Python displaying a `KeyError`

This error is demonstrated by the code shown here to the right.

The last line (highlighted in red) causes the Python interpreter to display the `KeyError` shown below.

```
d = {
    "OMG" : "Oh My God!",
    "LOL" : "Laugh out Loud",
    "IMHO" : "In My Humble Opinion",
}

print(d['LOLL'])
```



```
Traceback (most recent call last):
  File "C:\PDST\Work in Progress\Python Workshop\Manual
    print(d['LOLL']) # syntax error: Key does not exist
KeyError: 'LOLL'
```


The problem is that Python cannot find a key entry with the name `LOLL` in dictionary `d` and the probable cause is a typing error made by the programmer.



SYNTAX CHECK #2:

```
d = {
    "OMG" : "Oh My God!",
    "LOL" : "Laugh out Loud",
    "IMHO" : "In My Humble Opinion",
}

print(d['omg'])
```



Key names are case sensitive so the listing below will also result in a `KeyError`. The error message is shown below.

```
Traceback (most recent call last):
  File "C:\PDST\Work in Progress\Python Workshop\Manual and
    print(d['omg']) # syntax error: Keys are case sensitive
KeyError: 'omg'
```



Explain the possible causes for a `KeyError` in Python.

Keys can be variables

It is useful to be aware that the index used to lookup a dictionary value can be stored in a variable. Consider the code below.

```
1. d = {
2.     "OMG" : "Oh My God!",
3.     "LOL" : "Laugh out Loud",
4.     "IMHO" : "In My Humble Opinion",
5. }
6.
7. key = "OMG"
8. value = d[key]
9. print(value)
```



Line 7 declares a variable called `key` and assigns the string `OMG` to it. Line 8 uses the contents of `key` as the index to retrieve its corresponding value from the dictionary `d`. The result of the lookup is assigned to the variable `value`. Finally, line 9 causes the contents of `value` to be displayed.



SYNTAX CHECK #3:

If Python does not recognise the identifier used to lookup a dictionary it will display a `NameError`.



Experiment!

Key in (or copy+paste from GitHub) the code below and explain why both result in a `NameError` when they are run.

```
1. d = {
2.     "OMG" : "Oh My God!",
3.     "LOL" : "Laugh out Loud",
4.     "IMHO" : "In My Humble Opinion",
5. }
6.
7. print(d[OMG])
```



```
1. d = {
2.     "OMG" : "Oh My God!",
3.     "LOL" : "Laugh out Loud",
4.     "IMHO" : "In My Humble Opinion",
5. }
6.
7. key = "OMG"
8. value = d[keyy]
9. print(value)
```



The code below defines a dictionary called `capitals` and prompts the user to enter one of the six nation countries.

```
capitals = {
    "Ireland" : "Dublin",
    "Scotland" : "Edinburgh",
    "England" : "London",
    "Wales" : "Cardiff",
    "France" : "Paris",
    "Italy" : "Rome"
}

country = input("Enter one of the six nations : ")
print("The capital of "+country+" is "+capitals[country])
```

If the user enters one of the country names that is stored as a key in the dictionary, the program will display the corresponding capital city. The contents of the variable `country` is used as the key to lookup the dictionary `capitals`.



Experiment!

What happens when the user enters the name of a country that is not in the dictionary `countries`? Explain.



Experiment!

What is the purpose of the variable `i` in the code below?

Change the for loop to `for i in range(10)` : and explain what happens

```
numbers = {
    1 : "aon",
    2 : "dó",
    3 : "trí",
    4 : "ceathair",
    5 : "cúig",
    6 : "sé",
    7 : "seacht",
    8 : "ocht",
    9 : "naoi",
    10: "deich"
}

for i in range(1, 11):
    print(numbers[i])
```

Operations to access values

Indexing is not the only technique that can be used to retrieve dictionary values. In fact, Python dictionaries support a number of operations that can be used for this purpose - two of these are `get` and `pop`

<code>get (key)</code>	This call returns the value in the dictionary that corresponds to key. If the key does not exist the call returns <code>None</code> (unless a default value has been specified for the key). Therefore, this command never results in a <code>KeyError</code> .
<code>pop (key)</code>	This call returns the value in the dictionary that corresponds to key. If the value is found the element is removed from the dictionary. Otherwise the call results in a <code>KeyError</code> .

The use of `get` is highlighted in the code listing shown here.

In the first sample run the user enters *Maths* and the value *H4* is returned and assigned to `result`.

In the second sample run, the call to `get` returns `None` as the key entered by the user (i.e. *sdf*) is not found.

```
grades = {
    "Irish" : "H3",
    "English" : "O3",
    "Maths" : "H4",
    "Computer Science" : "H1",
}

subject = input ("Enter subject name: ")
result = grades.get(subject)
print("Result for %s was %s" %(subject, result))
print(grades)
```

Sample Run 1

```
Enter subject name: Maths
Result for Maths was H4
{'Irish': 'H3', 'English': 'O3', 'Maths': 'H4', 'Computer Science': 'H1'}
```

Sample Run 2

```
Enter subject name: sdf
Result for sdf was None
{'Irish': 'H3', 'English': 'O3', 'Maths': 'H4', 'Computer Science': 'H1'}
```

Notice from both sample runs that the contents of the dictionary `grades` remain unchanged after the call to `get`.

This can be contrasted with the use of `pop` which removes the element from the dictionary as shown here.

```
grades = {
    "Irish" : "H3",
    "English" : "O3",
    "Maths" : "H4",
    "Computer Science" : "H1",
}

subject = input ("Enter subject name: ")
result = grades.pop(subject)
print("Result for %s was %s" %(subject, result))
print(grades)
```

When the key is found its value is returned and the element is deleted from the dictionary.

```
Enter subject name: Maths
Result for Maths was H4
{'Irish': 'H3', 'English': 'O3', 'Computer Science': 'H1'}
```

When the key is not found Python raises a `KeyError`

```
Enter subject name: sdf
Traceback (most recent call last):
  File "C:\PDST\Work in Progress\Python Workshop\s
    result = grades.pop(subject)
KeyError: 'sdf'
```

One way of avoiding this type of error is to use the `in` keyword to test the key for membership before attempting the call to `pop`. This is exemplified below.

```
grades = {
    "Irish" : "H3",
    "English" : "O3",
    "Maths" : "H4",
    "Computer Science" : "H1",
}

subject = input ("Enter subject name: ")
if subject in grades:
    result = grades.pop(subject)
    print("Result for %s was %s" %(subject, result))
else:
    print("Result for %s does not exist. %s not found" %(subject, subject))

print(grades)
```

This time when a key which does not exist is entered, a meaningful message is displayed and the dictionary is left unaltered.

```
Enter subject name: sdf
Result for sdf does not exist. sdf not found
{'Irish': 'H3', 'English': 'O3', 'Maths': 'H4', 'Computer Science': 'H1'}
```

Adding, Changing, and Deleting Dictionary Elements


The technique of indexing can be used to add, change and delete elements. These operations are now discussed in turn

Adding new elements to a dictionary

New elements are added to a dictionary when a key that does not already exist is used as the index in an assignment statement.

Line 7 in the code shown adds a new (fourth) element to the dictionary, d.

```
1. d = {
2.     "OMG" : "Oh My God!",
3.     "LOL" : "Laugh out Loud",
4.     "IMHO" : "In My Humble Opinion",
5. }
6.
7. d['WYD'] = "What are you doing"
8. print(d)
```



The output displayed is:

```
{'OMG': 'Oh My God!', 'LOL': 'Laugh out Loud', 'IMHO': 'In My Humble Opinion', 'WYD': 'What are you doing'}
```

The general syntax for adding elements to a dictionary is:

<dict-name>[key] = value

where, `dict-name` is the name of the dictionary and `key` and `value` specify the key-value pair of the new element to add.



Get Coding!

Add the following key-value pairs to the dictionary, d defined above

```
"MUA" : "Make Up Artist"
"SWAG" : "Stuff We All Get"
"WTP" : "What's the plan"
```

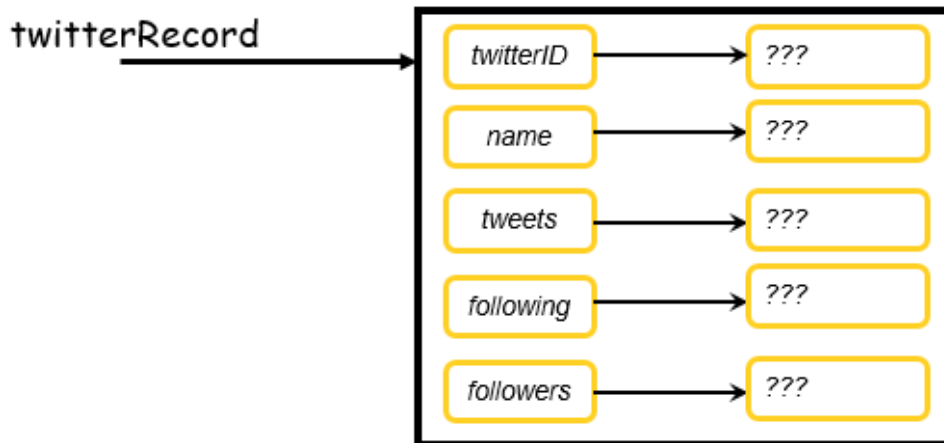


Experiment!

What happens when you try and add a new value to a dictionary using a key that already exists?

Building up a simple empty dictionary

Imagine we were asked to write a simple program for Twitter that would build up a dictionary to store information entered about a person/user. Specifically, let's say we were asked to build up, and populate a dictionary with the following structure:



The keys (i.e. handle, name, tweets etc.) are shown on the left hand side and the corresponding values on the right will be entered by the end-user when the following program is run.

```

1. # Create the initial (empty) dictionary
2. twitterRecord = {}
3.
4. # Gather the data from the end user
5. handle = input("Enter twitter handle :")
6. twitterRecord['twitterID'] = handle
7. name = input("Enter name : ")
8. twitterRecord['name'] = name
9. tweets = int(input("Enter number of tweets : "))
10. twitterRecord['tweets'] = tweets
11. following = int(input("Enter number for following : "))
12. twitterRecord['following'] = following
13. followers = int(input("Enter number of followers : "))
14. twitterRecord['followers'] = followers
15.
16. print(twitterRecord)

```



Lines 6, 8, 10, 12, and 14 all assign whatever value the user enters for the specified key into the dictionary.

Note that decisions about what information a system should store are usually made (by the system analysts and designers) before the programmer(s) begin coding. In this case, it was decided to store the person's handle and name as well as the number of tweets they sent, the number of people they are following and the number of followers they have.

The next example builds up a dictionary of student results. The results are stored as key-value pairs made up of a subject name and a corresponding mark.

In this first version of the program the dictionary stores the student name and one result. A sample run is shown here to the right

```
1. # Version 1. A dictionary to store a student's result
2. results = {}
3.
4. name = input("Enter student name: ")
5. results['name'] = name
6. subject = input ("Enter subject name: ")
7. mark = input ("Enter percentage mark for "+subject+": ")
8. results[subject] = mark
9. print(results)
```



```
Enter student name :Joe
Enter subject name: Irish
Enter percentage mark for Irish 45
{'name': 'Joe', 'Irish': '45'}
```

Notice on line 8 that the identifier `subject` is not enclosed in quotes. This is how the name of the subject entered by the end-user becomes the key for this dictionary entry.

The obvious limitation of the above program is that it only works for one subject. In this next version we use a `while` loop to allow multiple subjects to be entered and stored in the dictionary.

```
# Version 2a. A dictionary to store multiple results for a student
results = {}

name = input("Enter student name: ")
results['name'] = name
while True:
    subject = input ("Enter subject name: ")
    if subject == "":
        break
    mark = input ("Enter percentage mark for "+subject+": ")
    results[subject] = mark

print(results)
```



A sample run of this code (version 2a) shows results for Irish and Maths being entered and stored in the dictionary.

```
Enter student name: Joe
Enter subject name: Irish
Enter percentage mark for Irish: 67
Enter subject name: Maths
Enter percentage mark for Maths: 90
Enter subject name:
{'name': 'Joe', 'Irish': '67', 'Maths': '90'}
```

The next block of code, version 2b is *logically equivalent* to version 2a i.e. if they are given the same input they will both produce the same output.

Notice however, the subtle difference between the `while` loops in the two versions.


In version 2a the `break` statement causes the loop to terminate. This happens whenever the user presses return for the subject name.

In version 2b the user is prompted to enter the subject name *before* entering the `while` loop for the first time, and then again at the end of each execution of the loop body. Whenever the user enters an empty subject name (by pressing return) the loop guard i.e. `subject != ""` is evaluated to `False` and the loop terminates.

```
# Version 2b. A dictionary to store multiple results for a student
results = {}

name = input("Enter student name: ")
results['name'] = name
subject = input ("Enter subject name: ")
while subject != "":
    mark = input ("Enter percentage mark for "+subject+": ")
    results[subject] = mark
    subject = input ("Enter subject name: ")

print(results)
```



In situations such as this – when there is a choice between more than one logically equivalent solution to the same problem – the decision as to which implementation to use usually rests with the programmer and very often boils down to a matter of personal programming style.



Modify the code in either of the previous two example (i.e. version 2a or 2b) so that the dictionary stores the results of one single subject for multiple students. A sample run of the desired program is shown below.

```
Enter subject name: Computer Science
Enter student name: Joe
Enter percentage mark for Joe: 67
Enter student name: Mary
Enter percentage mark for Mary: 76
Enter student name: Alex
Enter percentage mark for Alex: 98
Enter student name:
{'subject': 'Computer Science', 'Joe': '67', 'Mary': '76', 'Alex': '98'}
```

The solution is available by clicking here:



Changing dictionary values

The simplest way to change (update) dictionary values is to use the assignment statement. Line 7 in the code below illustrates how the dictionary value for key entry *LOL* could be changed from *Laugh out Loud* to *League of Legends*.

```
1. d = dict( {
2.     "OMG" : "Oh My God!",
3.     "LOL" : "Laugh out Loud",
4.     "IMHO" : "In My Humble Opinion",
5. })
6.
7. d['LOL'] = "League of Legends"
8. print(d)
```



The expression on the right hand side of the assignment operator ('=') in line 7 is assigned as the new value for the element in *d* that is matched to the key 'LOL'.

The output displayed is:

```
{'OMG': 'Oh My God!', 'LOL': 'League of Legends', 'IMHO': 'In My Humble Opinion'}
```



Experiment!

What happens when you try to change a value using a key that does not exist (e.g. typo error in key)?



Experiment!

Key in (or copy+paste from GitHub) the code below and use the output to explain what the *update* command used in lines 12 and 13 does.

```
1. d1 = {
2.     "OMG" : "Oh My God!",
3.     "LOL" : "Laugh out Loud",
4.     "IMHO" : "In My Humble Opinion",
5. }
6.
7. d2 = {
8.     "LOL" : "League of Legends",
9. }
10.
11. print(d1)
12. d1.update(IMHO = "In My Honest Opinion")
13. d1.update(d2)
14. print(d1)
```




Deleting dictionary elements

Dictionary elements can be deleted using the `del` keyword.

The use of `del` is exemplified on line 7 below which removes the entry indexed by `LOL` from the dictionary.

```
1. d = {  
2.     "OMG" : "Oh My God!",  
3.     "LOL" : "Laugh out Loud",  
4.     "IMHO" : "In My Humble Opinion",  
5. }  
6.  
7. del d['LOL']  
8. print(d)
```



The effect is to delete the second element, and in doing so, reduce the number of elements in the dictionary from three to two. The output displayed is:

```
{'OMG': 'Oh My God!', 'IMHO': 'In My Humble Opinion'}
```

The general syntax to delete a dictionary element is:

```
del <dict-name>[key]
```

where, `dict-name` is the name of the dictionary and `key` is the lookup value to use in locating the element to delete. If the key is not found Python raises a `KeyError`

If `del` is used on a dictionary without specifying an index, the dictionary reference is removed from the program's namespace. The effect of the following statement is to delete the variable represented by `dict-name` from the program.

```
del <dict-name>
```

Finally, the `clear` command deletes all elements from a dictionary. The resulting dictionary is left empty.



Get Coding!

Write code to remove all the elements from the dictionary, `d` defined above



Experiment!

What is the relationship between *d1* and *d2* at the end of this program?

```

1. d1 = {
2.     "OMG" : "Oh My God!",
3.     "IMHO" : "In My Humble Opinion",
4. }
5.
6. d2 = {
7.     "LOL" : "League of Legends",
8. }
9.
10. d1.clear()
11. del d2['LOL']
    
```



Explain what the *clear* command used on line 10 does.



Experiment!

What is the relationship between *d1* and *d2* at the end of this program?

```

1. d1 = {
2.     "OMG" : "Oh My God!",
3.     "IMHO" : "In My Humble Opinion",
4. }
5.
6. d2 = {
7.     "LOL" : "League of Legends",
8. }
9.
10. d1.clear()
11. del d2
    
```



Explain what the *del* command used in line 11 does.



Experiment!

Explain what happens when you run the following code.

```
1. d = {  
2.     "OMG" : "Oh My God!",  
3.     "LOL" : "Laugh out Loud",  
4.     "IMHO" : "In My Humble Opinion",  
5. }  
6.  
7. del d['WYD']  
8. print(d)
```





Experiment!

Explain what happens when you run the following code.

```
1. d = {  
2.     "OMG" : "Oh My God!",  
3.     "LOL" : "Laugh out Loud",  
4.     "IMHO" : "In My Humble Opinion",  
5. }  
6.  
7. del d  
8. print(d)
```





Programming Exercises 7.1¹⁵

1. Study the dictionary definition shown in part b) carefully and answer the questions that follow:

a) What is the name of the dictionary? _____

b) Identify the keys and values in the dictionary

```
car = {
    'reg': "131 CN 6439",
    'make': "Audi",
    'model' : "A6",
    'year' : 2013,
    'kms' : 52739,
    'colour': "Silver",
    'diesel': True,
}
```


Keys: _____

Values: _____

c) Suggest name for two additional key-value pairs that could be added to `car`

d) Provide an alternative definition for `car` using the `dict` function by completing the code inside the red box shown.

```
car = dict( {
```



e) Predict what output would be generated by each of the following `print` statements.

(Note: some of these statements generate an error.)

(i) `print(car['make'])` _____

(ii) `print(car[model])` _____

(iii) `print(car['miles'])` _____

(iv) `print(car['colour'][0])` _____

(v) `print(car['diesel'][0])` _____

(vi) `print(car['reg'][4:5])` _____

¹⁵ See Appendix G for solutions

f) Assume that `currentYear` is a variable that has been assigned the value of the current year (e.g. `currentYear = 2018`). What do you think the statement `print(car['kms']/(currentYear - car['year']))` would output?

g) What information does this output convey to the end user?

h) Now create a dictionary definition for `car` so that the `print` statements shown below can run without errors. Use the space provided to record the actual output.

- (i) `print(car['make'])` _____
- (ii) `print(car['model'])` _____
- (iii) `print(car['kms'])` _____
- (iv) `print(car['colour'][0])` _____
- (v) `print(car['diesel'])` _____
- (vi) `print(car['reg'][4:5])` _____
- (vii) `currentYear = 2018`
`print(car['kms']/(currentYear - car['year']))` _____



What have you learned about dictionaries by completing this exercise?

2. The dictionary `colours` contains a mapping of five colours in English to Irish. Study the definition carefully (ignoring the deliberate translation errors for the moment) and answer the following questions.

- a) Write an alternative definition for `colours` in the space provide on the right.
(Hint: use the `dict` function.)

```
colours = {
    'white': "bán",
    'red': "dearg",
    'green' : "gorm",
    'blue' : "glas",
    'black' : "dubh",
}
```

- b) Predict what happens when the following lines of code are added (individually)
(Hint: None of these statements generate a syntax error.)

(i) `colours['yellow'] = 'buí'`

(ii) `colours['pink'] = 'bán agus dearg'`

(iii) `colours['green'] = 'glas'`

(iv) `colours['blu'] = 'gorm'`

(v) `colours.update(blue='gorm')`

(vi) `colours.update(green='gorm', blue='dubh')`

(vii) `del colours['white']`

c) In each of the following blocks of code line 9 reads a value from the end-user and line 11 attempts to use that value as a key to lookup the dictionary, `colours`. The idea is to assign the translated colour to the variable, `translation`.

Each block illustrates a different technique used to retrieve a value from a dictionary.

A.	Indexing	<pre>8. # Prompt the user to enter a colour 9. colour = input("Enter a colour: ") 10. # Look up the translation of colour 11. translation = colours[colour]</pre>
B.	get	<pre>8. # Prompt the user to enter a colour 9. colour = input("Enter a colour: ") 10. # Look up the translation of colour 11. translation = colours.get(colour)</pre>
C.	pop	<pre>8. # Prompt the user to enter a colour 9. colour = input("Enter a colour: ") 10. # Look up the translation of colour 11. translation = colours.pop(colour)</pre>

For each block (i.e. A, B and C) predict the value of `translation` given the inputs of *white* and *orange* for `colour` (six separate runs).

	white	orange
A.		
B.		
C.		

d) Key in (or copy+paste from GitHub) the code and run it to check all the predictions you made as part of this exercise.



Explain the differences between indexing, get and pop as techniques to retrieve values from a dictionary.

- Write a program that defines a dictionary called `book`. The keys should be `isbn`, `title` and `author`. You should make up your own values.

Now add a fourth key-value pair to your dictionary.

- The code below prompts the end user to enter a month (`month_name`) and then calls a function to return the number of days in that month to the variable `numDays` which is then displayed.

```
def daysInMonth(month_name):
    if month_name == "Feb":
        return 28
    elif month_name in ("Apr", "Jun", "Sep", "Nov"):
        return 30
    elif month_name in ("Jan", "Mar", "May", "Jul", "Aug", "Oct", "Dec"):
        return 31
    else:
        return

print("List of months: Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec")
month_name = input("Input the name of Month: ")
numDays = daysInMonth(month_name)
if numDays == None:
    print("Invalid month entered")
else:
    print("No. of days: %d" %numDays)
```



Define a dictionary called `days` so that the following implementation of `daysInMonth` can be used instead of the one shown above. The two programs will be logically equivalent.

```
def daysInMonth(month_name):
    return(days[month_name])
```




Which of the two implementations do you think is better and why?

Iterating over dictionaries

Occasionally the need arises to write code to perform some processing on each individual dictionary element. Such situations can be dealt with by traversing (or iterating over) the dictionary. A simple example would be to display every key and/or value in a separate formatted message. Let's say we have a dictionary defined as follows (both definitions are equivalent):

```
results = dict({
    'Joe': 67,
    'Mary': 76,
    'Alex': 72,
    'Sarah': 82,
    'Fred': 64,
    'Pat': 91,
})
```



```
results = {
    'Joe': 67,
    'Mary': 76,
    'Alex': 72,
    'Sarah': 82,
    'Fred': 64,
    'Pat': 91,
}
```

Python supports a small number of commands which can be used in conjunction with a `for` loop to iterate over the elements of a dictionary. These are summarised in the table below.

<code>keys()</code>	This call returns a list of all the keys in a dictionary
<code>values()</code>	This call returns a list of all the values in a dictionary
<code>items()</code>	This call returns a list of all the key-value pairs in a dictionary
<i>[Technically <code>keys()</code>, <code>values()</code> and <code>items()</code> actually return what Python calls a view object. For the purpose of this manual it is fair to think of view objects and lists as equivalent.]</i>	

The use of `keys()` is illustrated below – the output is shown on the right. Note that, on each iteration of the loop, the variable `k` takes on (i.e. is assigned) the value of the next key in the list of keys returned by the call to `results.keys()`

```
for key in results.keys():
    print("Name: %s" %key)
```

```
Name: Joe
Name: Mary
Name: Alex
Name: Sarah
Name: Fred
Name: Pat
```

The pattern is so common that the use of `keys()` can be omitted entirely. Both loops are logically equivalent.

```
for key in results:
    print("Name: %s" %key)
```

The built-in function `sorted` can be used to sort the keys – again the output is displayed on the right:

```
for key in sorted(results):
    print("Name: %s" %key)
```

```
Name: Alex
Name: Fred
Name: Joe
Name: Mary
Name: Pat
Name: Sarah
```

Since dictionaries are unordered the use of `sorted` can have no effect on the internal ordering of the dictionary elements.

The following code exemplifies the use of `values()` to iterate over the individual values held in the `results` dictionary. This time the variable `v` takes on (i.e. is assigned) the next value in the list of values returned by the call to `results.values()`

```
for k in results.values():
    print("Name: %s" %k)
```

```
Value: 67
Value: 76
Value: 72
Value: 82
Value: 64
Value: 91
```

The code to the right shows how to calculate the mean value by:

- adding up all the values
- storing their sum in `total` and
- dividing by the number of values (note the use of the `len` built in function).

```
total = 0
for v in results.values():
    total = total + v

mean_result = total/len(results)
print("Mean result: %d " %mean_result)
```



Code to calculate the mean of a set of values.

The output generated is:

```
Mean result: 75
```

In typical Python fashion there's usually simpler solution! In this case, the same result could have been achieved by importing and using the `mean` function as follows:

```
from statistics import mean
... # define results here

mean_result = mean(results.values())
print("Mean result: %d " %mean_result)
```



Finally, the use of `items` to iterate over and display the keys and values in the `results` dictionary is illustrated in the code below:

```
for k, v in results.items():
    print("Name: %s Result: %d" % (k,v))
```

```
Name: Joe Result: 67
Name: Mary Result: 76
Name: Alex Result: 72
Name: Sarah Result: 82
Name: Fred Result: 64
Name: Pat Result: 91
```

Notice that there are two variables – `k` and `v` – in the `for` loop. The `items` function actually returns a list of pairs known as **tuples**¹⁶. There is a one-to-one correspondence between the tuples and the key-value pairs in the dictionary. On each iteration of the `for` loop the variable `k` takes on the first element of the tuple and the variable `v` takes on the second.

Once again we can use the `sorted` function to sort the list of items. The following code shows how the dictionary can be sorted on its keys.

```
for k, v in sorted(results.items()):
    print("Name: %s Result: %d" % (k,v))
```

```
Name: Alex Result: 72
Name: Fred Result: 64
Name: Joe Result: 67
Name: Mary Result: 76
Name: Pat Result: 91
Name: Sarah Result: 82
```

Finally, the code to sort the dictionary by values is shown below. This code is shown for completeness only and an explanation is beyond the scope of this manual.

```
import operator

results = dict({
    'Joe': 67,
    'Mary' : 76,
    'Alex' : 72,
    'Sarah' : 82,
    'Fred': 64,
    'Pat' : 91,
})

for k, v in sorted(results.items(), key=operator.itemgetter(1)):
    print("Name: %s Result: %d" % (k,v))
```



```
Name: Fred Result: 64
Name: Joe Result: 67
Name: Alex Result: 72
Name: Mary Result: 76
Name: Sarah Result: 82
Name: Pat Result: 91
```

¹⁶ A tuple is another Python datatype. Tuples behave very like lists except that their elements cannot be changed i.e. they are immutable sequences. For more information on tuples see the official Python reference at <https://docs.python.org/3.6/library/stdtypes.html#tuple>

Dictionaries and Lists.

By this stage it should be evident that dictionaries and lists are conceptually very similar data structures. The main similarities are:

- Both are **compound datatypes**. This means that they can be used to store (and retrieve) multiple values using a single variable. (Compound datatypes can be contrasted with simple datatypes such as, integer, float and Boolean. Variables of these simple types can only store a single value at any one time.
- Both are **mutable** meaning that they can grow and shrink dynamically as new elements are added and old elements are deleted.
- Both support **indexing** to access elements. However, the semantics of how exactly lists and dictionaries use indexing is perhaps the greatest distinguishing feature between the two data structures. This semantics is described as follows.

Lists are indexed using a zero-based positional offset. Dictionaries are indexed using a key. The datatype of a list index must be integer whereas, with dictionaries the datatype of the index must match the datatype of the key in the dictionary.



KEY POINT: Dictionary values are retrieved using a key (and not an offset position as used by lists).

Another significant difference between lists and dictionaries is that lists are ordered collections whereas dictionaries are not. Dictionary values are mapped by a key and consequently can exist anywhere in the dictionary. List values, on the other hand exist at a fixed position defined by its index.

The main implication of this is that certain list operations that depend on ordering (e.g. slicing, concatenation, `append`, `insert` and `remove`) have no meaning for dictionaries. Errors which we associate with lists resulting from using an out of range index (i.e. `IndexError`) also have no meaning for dictionaries (they have their own types of errors).

STUDENT TIP

It can be useful to think of a dictionary as a list whose elements can be retrieved using a key as opposed to a zero-based positional index.

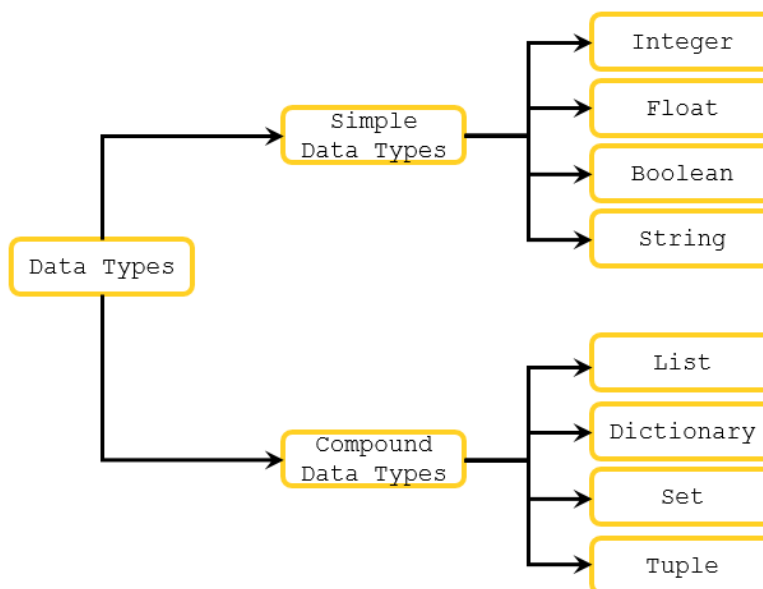
A word on datatypes and usage notes

Lists and dictionaries are among the most powerful data structures offered by Python and the main reason for this lies in their flexibility – especially when it comes to the wide range and type of values that they can be used to store. The values stored in lists and dictionaries can be of any datatype – there are effectively no restrictions.



KEY POINT: There are virtually no limitations to what can be stored in lists and dictionaries.

Before exploring this point in any more detail it is worth looking at the following graphic which classifies Python datatypes into two broad groups – simple and compound (aka composite).



Note that Python contains many more datatype than those illustrated and also that the classification is for illustrative purposes only¹⁷. A simple datatype can be thought of as atomic in the sense that values cannot be sub-divided into further sub-types, whereas a compound datatype can contain values which themselves can be either simple or compound.

In terms of the types of values that lists and dictionaries can store, we have a very broad spectrum. At one end of this spectrum values can be uniform and simple, while at the other end at the other end it is possible for elements to be of different datatypes and also compound.

¹⁷ Python represents everything as objects so, strictly speaking, it has no simple datatypes.



BREAKOUT ACTIVITIES (Dictionaries)

BREAKOUT 7.1: Frequency Counters

1. Key in (or copy+paste from GitHub) the following code and run it a number of times before attempting the questions and activities that follow:

```
1. sentence = input("Enter a sentence: ")
2. chars = {}
3. for char in sentence:
4.     if char in chars:
5.         chars[char] = chars[char] + 1
6.     else:
7.         chars[char] = 1
8. print(chars)
```



- a) Describe what the program does.

- b) Insert the line `print(char)` between lines 3 and 4 (indented) and run the program again. Use your observations to state the purpose of `char`.

- c) What type of data structure is `chars`?

- d) What information do the key-value pairs hold in `chars`?

The `chars` keys are:

The `chars` values are:

- e) Replace lines 4-7 of the original program with the following single line of code. Make sure it is indented. Run the program again and use the space below to note any changes in the way it behaves.

```
chars[char] = chars.get(char, 0) + 1
```

- a) Browse to the official reference for the `get` dictionary operation and describe what it does in the context of the previous question.

<https://docs.python.org/3.6/library/stdtypes.html#mapping-types-dict>

- g) *Modify the program so that it only maintains a count of each vowel in the input sentence. A sample run is shown below.*

Hint: As each character is being processed check whether it is a vowel or not; if the character is a vowel add it to the dictionary and update the counter.

```
Enter a sentence: This is a test input sentence  
{'i': 3, 'a': 1, 'e': 4, 'u': 1}
```

- h) *Modify the program so that it maintains total count of vowels and consonants.*

Hint: The dictionary needs only two keys – vowels and consonants.

A sample run would look like this:

```
Enter a sentence: This is another test  
{'consonants': 14, 'vowels': 6}
```

- i) *Use the structure and patterns contained in the original program to develop a program that counts the number of occurrences of words in a piece of text. A sample run of the desired program is shown below.*

```
Enter a sentence: THE DAY IS A VERY VERY VERY NICE DAY SO IT IS  
{'THE': 1, 'DAY': 2, 'IS': 2, 'A': 1, 'VERY': 3, 'NICE': 1, 'SO': 1, 'IT': 1}
```

- j) **Add the following four lines of code to the end of the original program (no indentation) and run it. Use the space provided below to describe what the code does.**

```
from collections import Counter
c = Counter(chars)
max_pair = c.most_common()[0]
print("%s occurs most often %d times" %(max_pair[0], max_pair[1]))
```

- b) **Experiment! What happens if there are more than one most frequently occurring letters? How might the code be altered to display all the most frequently occurring letters?**

- k) **The code below shows a logically equivalent solution to that shown in part j) above. Which solution do you prefer and why? How might the code be altered to overcome the limitation highlighted in the last question?**

```
max_key = ""
max_value = 0
for k, v in chars.items():
    if v > max_value:
        max_value = v
        max_key = k

print("%s occurs most often %d times" %(max_key, max_value))
```


2. The short program prompts the end-user to enter a sentence and then displays the frequency count of every word in the sentence. A sample run of the program yields the following:

```
Enter a sentence: THE DAY IS A VERY VERY VERY NICE DAY SO IT IS
{'THE': 1, 'DAY': 2, 'IS': 2, 'A': 1, 'VERY': 3, 'NICE': 1, 'SO': 1, 'IT': 1}
```

Key the program in and run it until you are satisfied you understand what it does.

```
1. # Count the number of words in a sentence (v1)
2. sentence = input("Enter a sentence: ")
3. sent_list = sentence.split()
4. words = {}
5. for word in sent_list:
6.     if word in words:
7.         words[word] = words[word] + 1
8.     else:
9.         words[word] = 1
10. print(words)
```



Now attempt the following. (The knowledge gained from completing part 1 of this exercise should be helpful)

a) Predict the output of the above program for the following sentences

Have a very merry Christmas and a very very merry New Year

Expected Output:

happy birthday dear Mary happy birthday to you

Expected Output:

Ho ho ho!

Expected Output:

b) Describe the influence of case and punctuation on the way the program works.

c) Suggest ways by which the limitations caused by case and punctuation marks in the program could be overcome.

d) Design and implement a solution to your suggestion from the previous question.

e) Replace lines 6-9 with a single statement that is logically equivalent i.e. each word encountered (i.e. the dictionary value) has one added to its frequency counter if the word already exists in the dictionary; otherwise the word is added to the dictionary with an initial frequency count set to 1.

f) Modify the program so that it performs the same word frequency analysis on text read from a .txt file (as opposed to a single sentence typed in by the user).

g) Change this program so that it ignores words of length greater than 3.

h) Use the program to calculate the average word length in the entire text.

i) Suggest further activities that could be based on this example.

3. We saw in chapter 5 (breakout activity 5.1) how the program shown here could be used to display a bar chart with the ten most frequently occurring words from a piece of text contained in a text file called `book.txt`.

```

import collections
import plotly
from plotly.graph_objs import Bar, Layout

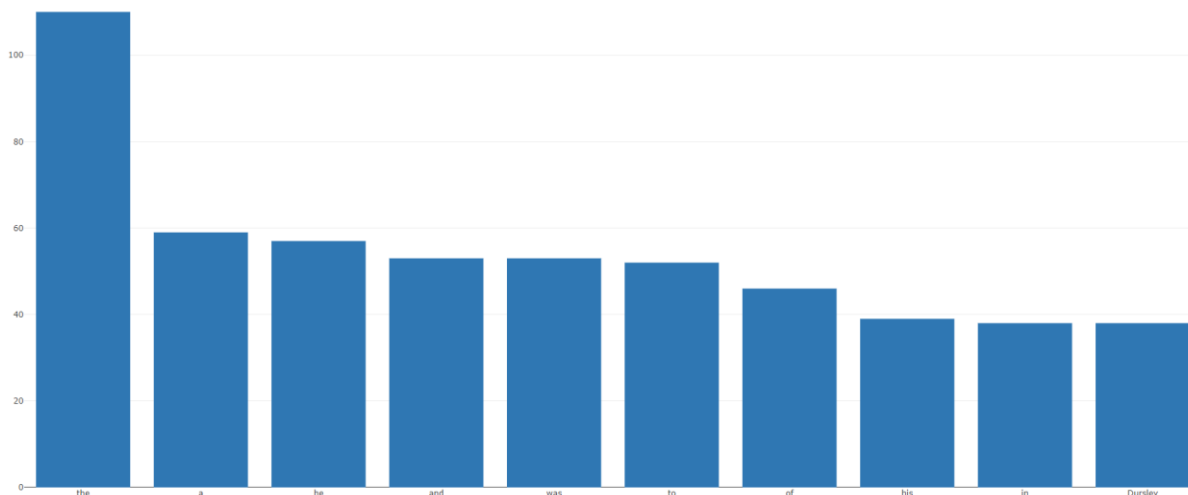
# IMPORTANT: Make sure books.txt is in the runtime directory
bookFile = open("book.txt","r") # Open the file
textList = bookFile.read().split()
bookFile.close()

c = collections.Counter(textList)

words = []
wordCount = []

print ('Most common:')
for word, count in c.most_common(10):
    words.append(word) # append the word to the words list
    wordCount.append(count)
    print ('%s: %d' % (word, count))

# Plot the results (x and y values must be in lists)
plotly.offline.plot({
    "data": [Bar(x=words, y=wordCount)],
    "layout": Layout(title="word count")
})
  
```



The task here is to provide an alternative implementation of the above program that incorporates the use of dictionaries.

NOTES:

Appendices

Appendix A: Python Keywords

False	break	else	if	not	while
None	class	except	import	for	with
True	continue	finally	in	pass	yield
and	def	for	is	raise	
as	del	from	lambda	return	
assert	elif	global	nonlocal	try	

Python 3.6.2 keywords

Appendix B: Python Built-in Functions

See <https://docs.python.org/3/library/functions.html>

abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

Appendix C: Python Assignment Operators

Operator	Name	Example	Same as	Result (x)
=	Simple Assignment	x = y	N/A	10
+=	Increment Assignment	x += y	x = x + y	10
-=	Decrement Assignment	x -= y	x = x - y	4
*=	Multiplication Assignment	x *= y	x = x * y	21
%=	Remainder Assignment	x %= y	x = x % y	1
/=	Division Assignment	x /= y	x = x / y	2.33333
//=	Floor Division Assignment	x //= y	x = x // y	2
**=	Power Assignment	x **= y	x = x ** y	343

Python Assignment Operators

Appendix D: Python Arithmetic Operators

Operator	Description	Example	Result
+	Addition	x + y	10
-	Subtraction	x - y	4
*	Multiplication	x * y	21
%	Remainder	x % y	1
/	Division	x / y	2.33333
//	Floor Division	x // y	2
**	Power	x ** y	343

Python arithmetic operators

Appendix E: Python Relational Operators

Operator	Description	Example	Result
>	Greater than	7 > 5	True
>=	Greater than or equal to	7 >= 5	True
<	Less than	7 < 5	False
<=	Less than or equal to	7 <= 5	False
==	Equal to (the same as)	7 == 5	False
!=	Not equal to (not the same as)	7 != 5	True

Python relational operators


















Appendix F: Truth Tables for not, and, and or


















A	not A
False	True
True	False



















A	B	A and B
False	False	False
False	True	False
True	False	False
True	True	True

















A	B	A or B
False	False	False
False	True	True
True	False	True
True	True	True






Appendix G: Sample Solutions to Selected Problems

Programming Exercise/ Breakout Activity	Solution
Breakout 1.2 Games Programming with <code>pygame</code> (chequer board)	
Breakout 2.1 Turtle Graphics – Question 3 (increase angle size)	
Breakout 2.1 Turtle Graphics – Question 4 (pseudo-code)	
Breakout 2.1 Turtle Graphics – Question 5 (random walk)	
Breakout 2.2 Automated Teller Machine (ATM) Menu System	
Breakout 2.3 Data Processing (average height) – Question 6	
Programming Exercise 3.1 – Question 1 (match index operation to the correct output)	
Programming Exercise 3.2 – Caesar cipher	
Programming Exercise 3.3 (match string method to the correct output)	
Breakout 3.1 School Survey Web Page – Question 2	
Breakout 3.2 RSS Feed Analysis – Questions 2, 4 and 6	
Breakout 4.1 Random Sentence Generator (partial solution)	
Breakout 4.2 Data Processing (heights) – Question 3	
Programming Exercise 5.1 (re-order the code)	
Breakout 5.1 Task Development (some ideas for the classroom)	
Breakout 5.2 Automated Teller Machine (ATM) Menu System (partial solution)	
Programming Exercise 5.2 (<code>for</code> loops) – Question 3 (chequer board)	

Programming Exercise/ Breakout Activity	Solution
Programming Exercise 6.1 (Q1 - list factors)	
Programming Exercise 6.1 (Q2 - greatest common divisor)	
Programming Exercise 6.1 (Q3 - valid date)	
Programming Exercise 6.1 (Q4 - ordinal numbers)	
Programming Exercise 6.1 (Q5 - long date)	
Programming Exercise 6.1 (Q6 - Easter)	
Programming Exercise 6.2 (Fibonacci)	
Programming Exercise 6.3 (Q2 - foobar)	
Programming Exercise 6.3 (Q3 - CAPTCHA)	
Programming Exercise 6.3 (Q4 – validate password)	
Programming Exercise 6.3 (Q5 – Collatz sequence)	
Programming Exercise 6.3 (Q6 – multiples of 3 or 5)	
Programming Exercise 6.3 (Q7 – perfect numbers)	
Programming Exercise 6.3 (Q8 – amicable pairs)	
Breakout 6.1 (ATM System) – initial solution	
Breakout 6.1 ATM System – suggested activities) Q1 – maximum withdrawal of €200	
Breakout 6.1 ATM System – suggested activities) Q2 – file processing	

Programming Exercise/ Breakout Activity	Solution
Breakout 6.1 ATM System – suggested activities) Q3 – PIN processing	
Breakout 6.1 ATM System – suggested activities) Q4 – no global variables	
Breakout 6.2 Summing Numbers (suggested activities) Q1(b) – sum of even	
Breakout 6.2 Summing Numbers (suggested activities) Q1(c) – sum from x to y	
Breakout 6.2 Summing Numbers (further activities) Q1 – sum of odd	
Breakout 6.2 Summing Numbers (further activities) Q2 – sum of reciprocals	
Breakout 6.2 Summing Numbers (further activities) Q3 – approximate pi	
Breakout 6.2 Summing Numbers (further activities) Q4 – approximate e	
Breakout 6.3 Turtle Graphics Part I Q2 – draw a square – no parameters	
Breakout 6.3 Turtle Graphics Part I Q3 – draw a square – with parameters	
Breakout 6.3 Turtle Graphics Part I Q4 – draw a rectangle	
Breakout 6.3 Turtle Graphics Part I Q5 – draw an n-sided polygon	
Breakout 6.3 Turtle Graphics Part II Q2 – draw angle	
Breakout 6.3 Turtle Graphics Part II Q3 – set pen position	
Breakout 6.3 Turtle Graphics Part II Q4 – draw angle passing in co-ordinates as parameters	
Breakout 6.3 Turtle Graphics Part II Q6 – investigate whether two drawAngles are allowed	
Breakout 6.3 Turtle Graphics Further Activities Q1 – refactor code	
Breakout 6.3 Turtle Graphics Further Activities Q2 (c) – draw the digits 2, 3 and 9	

Programming Exercise/ Breakout Activity	Solution
Breakout 6.4 Using check digits to verify codes. Further Activities (ISBNs and Credit Cards) Q2 (Luhn's Algorithm to validate a credit card number)	
Programming Exercise 7.1 Q1 (b) – identify the key-value pairs	
Programming Exercise 7.1 Q1 (d) – alternative dictionary definition	
Programming Exercise 7.1 Q1 (e) – predict the output	
Programming Exercise 7.1 Q1 (h) – predict the output	
Programming Exercise 7.1 Q2 (a) – alternative dictionary definition	
Programming Exercise 7.1 Q2 (b) – predict the output	
Programming Exercise 7.1 Q2 (b) – predict the output (translation)	
Programming Exercise 7.1 Q3 – define a dictionary called <code>book</code>	
Programming Exercise 7.1 Q4 – a dictionary to display the number of days in a month	
Breakout 7.1 Frequency Counters Q1(e) – count letters	
Breakout 7.1 Frequency Counters Q1(g) – count vowels	
Breakout 7.1 Frequency Counters Q1(h) – count vowels and consonants	
Breakout 7.1 Frequency Counters Q1(i) – count words	
Breakout 7.1 Frequency Counters Q1(j) – count most common letters	
Breakout 7.1 Frequency Counters Q1(k) – count most common letters (version 2)	

Programming Exercise/ Breakout Activity	Solution
Breakout 7.1 Frequency Counters Q2(e) – count words	
Breakout 7.1 Frequency Counters Q2(f) – count most common words	
Breakout 7.1 Frequency Counters Q2(g) – words of length greater than 3	
Breakout 7.1 Frequency Counters Q2(h) – average word length	
Breakout 7.1 Frequency Counters Q3 – display bar chart with 10 most common words	

BLANK PAGE