

Python Programming

GCSE Computer Science

OCR Spec: J277



Contents: links to Chapters

Contents: links to Chapters	1
Chapter 1. The Basics	2
Chapter 2. Variables and Assignment Statements	4
Chapter 3. Data Types	6
Strings & String Formatting	8
Chapter 4. Functions & Procedures	14
Chapter 5. Structured Programming	19
OCR Exam Reference Language	21
Chapter 6. Importing Modules	37
Chapter 7. Scope of Variables in Functions	40
Chapter 8. Arrays/Lists	42
Chapter 9. Reading and Writing Files	52
Chapter 10. Validation Techniques	56
Chapter 11. Dealing with Errors	59
Chapter 12. Test Plans and Test Data	62

Chapter 1. The Basics

Using Python as a Calculator

Below is a list of the **mathematical operators** you need to know for OCR J277.

Operator (Python)	Operator (OCR Reference Language)	Name	Example	Output
+	+	Addition	4+5	9
-	-	Subtraction	8-5	3
*	*	Multiplication	4*5	20
/	/	Division	19/3	6.33
**	^	Exponent	2**4	16
%	MOD	MOD (Modulo arithmetic gives remainder only from division)	19%3	1
//	DIV	DIV (Whole-number division gives the "floored quotient")	7//2 (rounded down)	3

Expressions

In mathematics an **expression** means the numbers, symbols and operators that are grouped together to show the value of something.

```
>>> 19%3
1
>>> 2**4
16
>>> 7//2
3
```

Order of Operation

Python automatically understands the order of operation of any mathematical calculations; i.e. BIDMAS.

B rackets
I ndices
D ivision
M ultiplication
A ddition
S ubtraction

1 Exercise: Calculate Using Python

1.1. Use print statements to show the result of these operations (predict the output each time!):

1. $50-5*6$
2. $(50-5*6)/4$
3. $21\%6$
4. $17//5$
5. $5*3+2$
6. $5**2$
7. $1//2$

1.2 Copy and complete this program for a cash machine (ATM):

- Use the MOD operator to add a layer of validation in the blank (____), which says whether the amount input can be dispensed or not.

```
print("We only dispense 10s and 20s. Input what cash you would like: ")
```

```
cash = int(input())
```

```
if ____:  
    print("That is fine.")
```

```
else:  
    print("We cannot dispense that amount.")
```

- EXT: Tell them what notes they will receive.
- EXT: Can you stop the program crashing if the user enters a word instead of a number? ([Jump to error handling](#) for an idea!)

1.3 Ask a user for a **time in minutes** and output how many **hours and minutes** that is (using DIV and MOD).

Chapter 2. Variables and Assignment Statements

Variables are reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

Constants are reserved memory locations to store values but they cannot change. Constants can be declared in JavaScript, Java, C# and C++ (e.g. `const pi = 3.142`), but not in Python. The convention in Python is to use ALL CAPS for constants, e.g. `PI = 3.142`

The data used in a program **must be stored in main memory** while the program is running. We use names for these areas in memory so we can refer to them in our programs.

- When we create a variable, the name we use refers to the location of the data item in memory.
- Each variable will have a data type (see Data Types). The data type determines what we can do with the data and how much space it needs in memory.

[From here I will refer to variables and constants as variables, but be aware of the difference in OCR Reference Language and other high level languages]

Variables Names

When we create variables and choose a name for them, there are some basic rules to follow:

1. Variable names **must not** have spaces

e.g. `myName` ✓ `my_name` ✓ `my_Name` ✓ `my name` ✗

2. Variable names **must not** use any reserved keywords (see below)

e.g. `break` ✗ `else` ✗

3. **Must not** start with a number or operator (but ok with a letter or underscore).

e.g. `3` ✗ `3things` ✗ `+++things` ✗ `top3Things` ✓ `_3things` ✓

4. **Do choose** a consistent naming style and stick to it! e.g. **camelCase** or **use_underscore**

5. **Do remember:** variable names are **case-sensitive** (this means you must use the same CAPITALS or lower case wherever you use the variable name)

6. **Do** use a name that **makes sense** (in the context of the program)

e.g. `student_Name` ✓ `sn` ✗ `x` ✗

The reserved keywords have a special meaning and purpose in Python; you can check what they are by typing `help('keywords')`.

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

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

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

When we **assign** a value to a variable, it is important that the **variable name is created before** the value is assigned.

```
>>> 5=shoe_size
...
SyntaxError: cannot assign to literal here.
```

When we assign values to variables, we can use the variable names in calculations instead of the values. This is good practice in coding!

```
>>> #Calculate the area of a rectangle
>>> length = 13.6
>>> width = 5.7
>>> area = length * width
>>> print(area)
77.52
>>>
```

2 Exercise: Variables and Assignment

1. Which of these variable names and assignment statements will NOT cause a syntax error?

variable names and assignment statements

```
continue_list = 15          #(a)
red Counter = True          #(b)
154.2 = length              #(c)
_1234 = 'password'          #(d)
my_best_friend = "Thomas"  #(e)
```

2. Write a program to calculate the area of a circle with a radius of 6.7 cm and print the result of the calculation.
 - a. Declare the value of π as 3.142
 - b. The area of a circle is $\pi \times \text{radius squared}$!
3. Write a program to store the following variables:
 - a. Forename
 - b. Surname
 - c. Age

Use **one print function** to display the above information in a sentence.

Chapter 3. Data Types

Introduction

Choosing the correct data type to use is important because it determines what actions we can perform on the data, e.g. we cannot divide a text string by an integer. Every value you store will have one of the following data types:

String and Character	<ul style="list-style-type: none">• A string is a series of characters, in quotation marks. These can be single or double quotation marks.• A “character” is a string constrained to a single character. This is not a separate data type in Python, but it is in <i>OCR Reference Language</i>!• We can “add” (concatenate) them together, e.g. ‘birth’ + ‘day’ would give you ‘birthday’. (Note that some languages, including <i>OCR Reference Language</i>, do not use commas to concatenate... but you can in Python unless it is in an <code>input()</code> function!
Integer	<ul style="list-style-type: none">• An integer is a whole number.• It can be positive or negative.
Float	<ul style="list-style-type: none">• A number with a fractional part• A float (short for “floating point number” because of the way it is handled in binary) is also called a “real number”.
Boolean	<ul style="list-style-type: none">• A Boolean value can be True(1) or False (0).• Named after the mathematician George Boole, a Boolean should be used when there are only 2 possible values.

Data type errors (TypeError)

Look at this example which causes a data type error (or “*TypeError*”) when the code is executed. Can you explain what the error message means:

```
>>> legal_age = 17
>>> student_age = input("Enter your age: ")
Enter your age: 18
>>> if student_age >= legal_age:
    print("You are old enough to drive")
else:
    print("You cannot drive yet, you are too young")

Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    if student_age >= legal_age:
TypeError: unorderable types: str() >= int()
>>>
```

Casting Data Types & Getting Input

Python assumes that all data entered using the `input()` built-in function is a **string** unless the data type is **cast** into a different data type. This means telling Python to treat the data input as an **integer** or **float**.

In this code the data input is **cast** to an **integer** when the input is requested. This could be done like this:

```
>>> student_age = input("Enter your age: ")
Enter your age: 18
>>> student_age = int(student_age)
>>> type(student_age)
<class 'int'>
```

Used to show the data type has changed



The example below is more **efficient** as only one line of code has been used to cast the data type, rather than two lines of code as shown above.

Efficient Code:

```
>>> legal_age = 17
>>> student_age = int(input("Enter your age: "))
Enter your age: 18
>>> if student_age >= legal_age:
    print("You are old enough to drive")
else:
    print("You cannot drive yet, you are too young")

You are old enough to drive
```

Casting: The process of forcing a value into a certain data type, e.g. digits entered at an input prompt changed from string to integer using `int()`. Sometimes called “type conversion”.

To cast as different data types:

Data type required	How to cast
String	<code>str(variable_name)</code>
Integer	<code>int(variable_name)</code>
Float	<code>float(variable_name)</code>

Strings & String Formatting

```
#These are all the same

my_string = 'Hello world'
print(my_string)
my_string = "Hello world"
print(my_string)
my_string = '''Hello world'''
print(my_string)
my_string = """Hello world"""
print(my_string)

#triple quotes allow the string to cover several lines

my_string = """
    The string "Hello world" is the
    first text that people learning
    to code output to screen"""
print(my_string)
```

```
>>>
Hello world
Hello world
Hello world
Hello world

    The string "Hello world" is the
    first text that people learning
    to code output to screen
```

Which one should I use?

- Using **single** quotation marks means that you need to use escape characters in the string if you also want to use characters like a backslash, apostrophe or double quotation marks.
- Using **double** quotation marks means that you do not need to use escape characters.
- Using **triple** quotation marks means that text can span several lines. These are used for “docstrings” and multiple-line comments.

Useful “Escape” Characters (** is a backslash, above right shift)

Escape Character	What it does
\'	Allows the use of an apostrophe inside a single-quotation-marks string
\"	Allows the use of double quotation marks inside a single-quotation-marks string
\n	New line
\t	Tab (indents your text string)

Examples:

```
>>> print("The quick brown \nfox jumped over\nthe lazy dog")
The quick brown
fox jumped over
the lazy dog
>>>
>>> print("Please choose from\n\t1) Play Game\n\t2) Quit")
Please choose from
        1) Play Game
        2) Quit
```



```
>>> print('Don't make this mistake!!')
...
SyntaxError: unterminated string literal (detected at line 1)
```

String Operations

Strings are immutable; this means that once we have created a string variable we cannot alter it or edit it.

There are a number of string operations we can perform on a string variable that will be useful when writing your code:

1. `len(myString)` – returns the number of characters in the string, including spaces
2. `myString.upper()` – returns the string in upper case
3. `myString.lower()` – returns the string in lower case
4. `myString[x:y]` – returns a substring of the original string starting at character x and ending before character y

Note:

Python is case sensitive, so in this example I am trying to test whether the two strings are the same by using the equality operator.

```
>>> text_1 = "Paris"
>>> text_2 = "PARIS"
>>> text_1==text_2
False
>>> text_1.upper()==text_2
True
>>>
```

If I test equality **without** converting the first string to upper case, the result of the comparison is false; Python does not recognise that the strings are the same.

Formatting Strings

When we need to use variables inside a print statement, there are a number of different ways we can do this:

```
>>> teacher = input("Please input your teacher's name: ")
Please input your teacher's name: Mr Jones
>>> print("Hello ",teacher)
Hello Mr Jones
>>> print("Hello "+teacher)
Hello Mr Jones
>>>
>>> print("Hello {0}".format(teacher))
Hello Mr Jones
>>>
```

1. Use a comma to add the variable into the print statement.
2. Concatenate the string with the variable inside the print statement.
3. Use the string method **.format()**.

The last method may seem more time-consuming than the first two but it allows a lot more flexibility.

Example:

```
>>> euro = 1.39
>>> cash = 250.0
>>> print("£{0} will buy {1} Euros at today's rate: {2}".format(cash,euro*cash,euro))
£250.0 will buy 347.5 Euros at today's rate: 1.39
>>>
```

3.1 Exercise: String formatting

1. Declare a string: "Computer Science is fun!"

Print the outputs from the following string manipulations:

- `len(myString)`
- `myString.upper()`
- `myString[3:9]`
- `myString[3:]`
- `myString[:3]`

Read the section: [String manipulation in OCR Reference Language](#), and copy the above string manipulations into your notes under *2.2 Programming Fundamentals*, giving the **OCR Reference Language equivalents**.

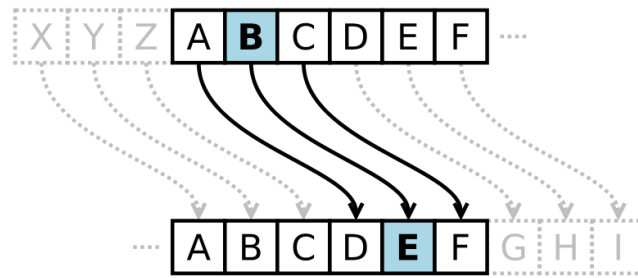
2. Prompt the user to enter two whole numbers. Output a message 'The average of your two numbers, (number 1) and (number 2) is (average)', **using the .format method**.
3. Use **one** print statement, with tabs and new lines, to print the following to screen:

There are 2 papers for GCSE Computer Science:

1. Computer Systems
2. Computational Thinking, Algorithms and Programming

Using ord() and chr()

These two built-in functions are used when creating a Caesar cipher program to encrypt a “plaintext” string. A Caesar cipher substitutes the actual character in the string with another letter a certain number of spaces further on in the alphabet.



The **ord()** function allows us to represent each letter as an ordinal number; we can then add the required number of letters to shift by, (which is usually between 1 and 26, and wraps around, to prevent gibberish and non-printing characters), to find the value of the new number.

The new number can be changed back to a letter by using the **chr()** function. The numbers used to represent each upper- or lower-case letter in the alphabet are based on the ASCII character set.

Note that in OCR reference language...

chr(...) is **CHR(...)**

and

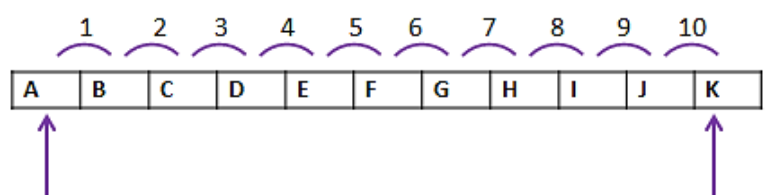
ord(...) is **ASC(...)**.

The diagram below shows the first 10 capital letters with their ASCII values and how that is represented in binary in the computer:

LETTER	ASCII VALUES	BINARY VALUES
A	65	01000001
C	67	01000011
D	68	01000100
E	69	01000101
F	70	01000110
G	71	01000111
H	72	01001000
I	73	01001001
J	74	01001010
K	75	01001011

So, an example in Python:

```
>>> ord('A')
65
>>> ord('a')
97
>>> letter = 'A'
>>> newLetter = ord(letter)+10
>>> chr(newLetter)
'K'
```



This simple example shows how we can use these two built-in functions to write a basic Caesar cipher. The key is 7 and **z** is 7 letters away from **s** in the alphabet.

```
password = "secret"
encrypted = '' # empty string for encrypted password

#encrypt password

key = 7
for each in password:
    newLetter = ord(each)+key
    encrypted +=chr(newLetter)

print(encrypted)
```

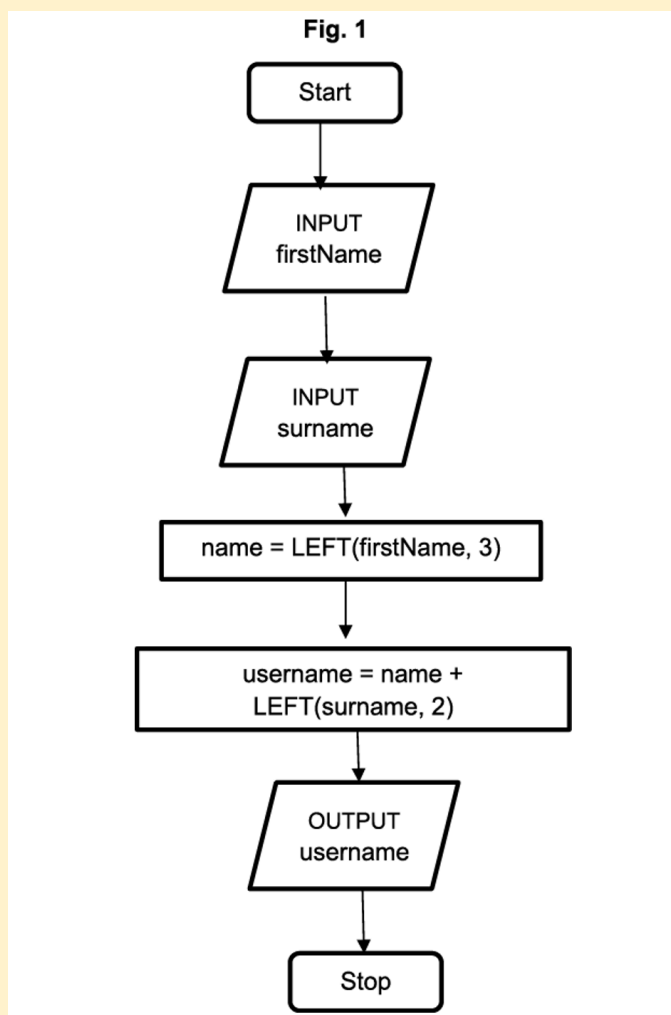
```
>>>
zljyl{
>>>
```

3.2 Exercise: Encryption

Prompt the user to enter a string for encryption and also enter a key value. Output the encrypted string with a suitable message. (*Don't look it up if you have done this before!*)

3.3 OCR String Operations Question

Johnny is writing a program to create usernames. The first process he has developed is shown in the flowchart in Fig. 1.



For example, using the process in Fig. 1, Tom Ward's user name would be TomWa.

(a). State, using the process in Fig. 1, the username for Rebecca Ellis.

[1]

(b). Johnny has updated the process used to create usernames as follows:

- If the person is male, then their username is the last 3 letters of their surname and the first 2 letters of their first name.
- If the person is female, then their username is the first 3 letters of their first name and the first 2 letters of their surname.

(i) What would be the username for a male called Fred Biscuit using the updated process?

[1]

(ii) Write an algorithm, using Python or pseudocode, for Johnny to output a username using the updated process.

[6]

[First: write it/ type it, but also... can you code this in Python?]

Chapter 4. Functions & Procedures

What are Functions and Procedures?

Functions and procedures are “sub programs”: blocks of code that perform a specific task. We can reuse them to make our code more **efficient**. Python has many ready-made functions that we can use – these are called built-in functions – and you will already have used the `print()` and `input()` functions.

Why should I use functions or procedures?

- They make your code **easier to read** as you can see what tasks each block of code performs (providing you use a suitable name).
- Functions and procedures can be **reused** instead of repeating code.
 - *If changes are needed, they can be made to just one block of code.*
 - *It is easier to organise the order in which the program runs.*



Functions vs Procedures

In Python, the blocks of code we write are all known as functions. However, in programming terms you will need to know the difference between a function and a procedure.

- A **procedure** is a self-contained block of code that performs a task and can be called on by the main program.
- A **function** is a type of procedure which will “**return** a value”.

Writing a Procedure

When we write a procedure or a function in Python the structure and layout of the block of code is exactly the same. If the block of code returns a value, it is known as a function, if it does not it is known as a procedure.

Here is an example of a very simple procedure which **does not return a value**:

```
File Edit Format Run Options Window Help
#simple procedure to print a message

def print_a_message():
    '''prints out a string'''
    print("Hello world!")
```

- We use the **def** statement to tell Python we are defining a procedure or function.
- The name should be what the procedure or function *does* – keep it simple and use underscores between words.
- The name is followed by two brackets (**parentheses**); these will hold any variables we want the procedure to use. These are called **parameters**.
- After the brackets we add a **colon**.
- The body of the procedure or function is then **indented** below (use the **tab** key).
- A triple-quoted “docstring” can be used to add detail to explain what the procedure does

In order to execute or ‘run’ the procedure or function and see the result of the code in the interpreter window, we must ‘call’ the procedure or function.

Calling the Procedure or Function

At the moment this function will not run until we instruct Python to execute the code by ‘calling’ the function:

```
File Edit Format Run Options Window Help
#simple procedure to print a message

def print_a_message():
    '''prints out a string'''
    print("Hello world!")

print_a_message()
```

4.1 Exercise: Basic Procedure

1. Write **procedures** to do the following:

Procedure 1	Write a procedure to add two numbers together and print the answer. Use the following variable names and values: x = 17 y = 22 Call the procedure
Procedure 2	Write a procedure to multiply x and y together and divide by z. Use the following variables: x = 6 y = 4 z = 8 Call the procedure

Extending the basic procedure

Look at this example:

```
File Edit Format Run Options Window Help
def add_numbers(x,y):
    '''Add variables x& y, print the result'''
    result = x + y # this produces the result
    print(result)

add_numbers(15,22)
add_numbers(37,98)
```

Procedures or functions are 'called' with arguments supplied instead of placeholders

Parameters are the 'placeholders' used when the procedure or function is defined; in this example these are x and y.

Arguments are the *actual values* we use when we 'call' the procedure or function.

4.2 Exercise: Extend Basic Procedure

1. Develop Procedure 2 from the previous exercise to use **parameters** and supply the following arguments:
x = 15, y = 13, z = 5

Returning a value from a function

When we *return a value* from a function, we can then use that value as an argument for another function or procedure.

Look at this example:

```
File Edit Format Run Options Window Help
#ask for the user name and print a welcome message

def get_name():
    '''get name input and return'''
    name = input("Please enter your name: ")
    return name

def print_greeting(name):
    '''prints greeting with name variable'''
    print("Hello {0}, welcome to my greeting program.".format(name))

name = get_name()
print_greeting(name)
```

This time when we call the `get_name()` function we must tell Python where the variable 'name' is created *before* we use it as an argument in the `print_greeting()` procedure.

4.3 Exercise: Returning Values

1. Write functions that will:
 - a. **Function 1:** e.g. `get_student_name()`... ask for the student name and return it
 - b. **Function 2:** ask for the name of their Computer Science teacher and return it
 - c. **Function 3:** ask for marks out of 9 for the last three homework tasks and return an average mark. (*Calculate the average inside the function*).
 - d. **Function 4:** create a `teacher_comment` function with 3 parameters. It will use an `if` statement to display the following message, depending on the average mark:
 - i. *average >=8. 'Well done, [student_name], [teacher_name] is very pleased with your effort.'*
 - ii. *average >=6 <8. 'A good effort, [student_name]. [teacher_name] thinks you should check your work carefully.'*
 - iii. *average <=5. '[student_name] this is poor. [teacher_name] has asked you to try harder.'*
 - e. Declare variables for the first 3 function calls and finally call the `teacher_comment` function, supplying those 3 variables as arguments.

Chapter 5. Structured Programming

When you create your own programs, it is important to structure them so that they are easy to **maintain**:

- Use **subprograms** to break up your code, allow you to debug in one place and avoid repeating chunks.
- Use sensible **identifiers** for variables and functions
- Use **comments** or docstrings to explain your code
- Use **whitespace**

Programming problems are easier to solve by breaking them down into a series of smaller steps which can be easier to understand and solve (this is called **decomposition**). The total solution is created when all the smaller subproblems have been solved.

The three **constructs** that we use in structured programming are:

- **Sequence**
- **Selection**
- **Iteration**

Sequence

In structured programming, the **sequence** in which instructions are executed is the same order as they appear in the code:

```
# Sequence Example

Item_Quantity = int(input("Please enter the quantity required: "))
Item_Price = float(input("Please enter the item price: "))
Total = Item_Quantity*Item_Price
print("The total cost is £{0}.".format(Total))
```

When trying to solve problems we can also represent them using either **pseudocode** or **flow charts**:

Pseudocode:

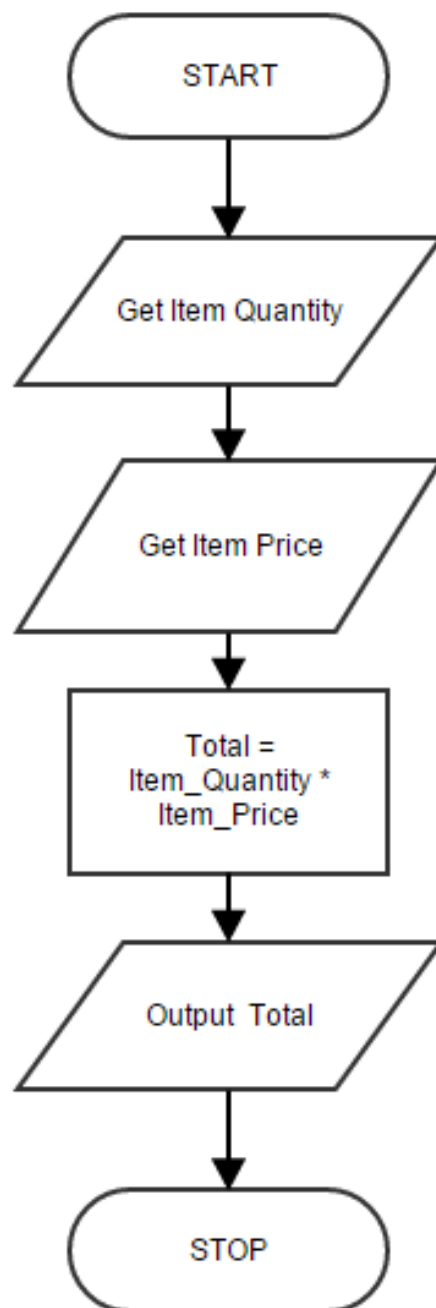
Pseudocode means “pretend code” and is an unambiguous algorithm but written in a generic, non-specific high-level language. For example:

```
Item_Quantity <-- USERINPUT
Item_Price <-- USERINPUT
Total <-- Item_Quantity * Item_Price
OUTPUT 'The total cost is £',Total
```

Flow chart:

An algorithm expressed as a diagram.

For example:



OCR Exam Reference Language

Note that in your exam papers you will see algorithms written in a **type of pseudocode** called “**OCR Exam Reference Language**”. This is very similar to Python, but has a few features which are more similar to Javascript and other languages. For example:

Meaning	Python	OCR Exam Reference Language
Operators	<code>%</code> <i>...modulo: remainder only</i> <code>//</code> <i>... floored quotient: integer only</i>	MOD <i>...modulo: remainder only</i> DIV <i>... floored quotient: integer only</i>
Comments	<code>#</code>	<code>//</code>
Assigning a constant	<code>VAT = 0.2</code> <i>...(capitalisation as convention in Python)</i>	<code>const vat = 0.2</code>
Casting as a real number	<code>float ("4.52")</code>	<code>real ("4.52")</code>
FOR loop ("count-controlled")	<code>for i in range (9):</code> <code>print ("Loop this")</code> <i>...prints "Loop this" 9 times</i>	<code>for i = 1 to 9</code> <code>print ("Loop this")</code> next i <i>or</i> <code>for count = 1 to 9</code> <code>print ("Loop this")</code> next count <i>...prints "Loop this" 9 times ("count" is inclusive!)</i>
FOR loop with step	<code>for i in range (2, 11, 2):</code> <code>print (i)</code>	<code>for i = 2 to 10 step 2</code> <code>print (i)</code> next i
WHILE loop ("condition-controlled")	<code>while answer != "Correct"</code> <code>answer = input("New answer")</code>	<code>while answer != "Correct"</code> <code>answer = input("New answer")</code> endwhile
"DO...UNTIL" WHILE loop	<i>...same as above in Python</i>	do <code>answer = input("New answer")</code> until answer == "Correct"
SELECTION (IF-THEN-ELSE)	<code>if answer == "Yes":</code> <code>print("Correct")</code> <code>elif answer == "No":</code> <code>print("Wrong")</code> <code>else:</code>	<code>if answer == "Yes" then</code> <code>print("Correct")</code> <code>elseif answer == "No" then</code> <code>print("Wrong")</code> <code>else</code>

	<pre>print("Error")</pre>	<pre>print("Error") endif</pre>
SELECTION (Switch-Case)	<pre>if day == "Sat": print("Saturday") if day == "Sun": print("Sunday") else: print("Weekday") ...switch-case N/A in Python</pre>	<pre>switch day: case "Sat": print("Saturday") case "Sun": print("Sunday") default: print("Weekday") endswitch</pre>
String handling	<pre>subject = "ComputerScience" len(subject) ...gives the value 15 subject[3:9] ...returns characters at index 3 to 8 inclusive, i.e. "puter" subject[0:4] ...returns "Comp" subject[-3:] ...returns "nce" or subject[13:] ...returns "nce" CONCATENATION: print("Hello, your name is: " + name) or print("Hello, your name is:", name) subject.upper() ... gives "COMPUTERSCIENCE" subject.lower() ... gives "computerscience" ord("A") ... returns 65 (ASCII character number) chr(97) ...returns 'a' (character at that ACSII number)</pre>	<pre>subject = "ComputerScience" subject.length ...gives the value 15 subject.substring(3,5) ...starts at character index 3 and counts 5 characters i.e. "puter" subject.left(4) ...returns "Comp" subject.right(3) ...returns "nce" CONCATENATION: print("Hello, your name is: " + name) ", " not used to concatenate by OCR subject.upper ... gives "COMPUTERSCIENCE" subject.lower ...gives "computerscience" ASC(A) ...returns 65 CHR(97) ...returns 'a'</pre>
File handling	<pre>myFile = open("sample.txt", "a")</pre>	<pre>newFile("sample.txt")</pre>

	<pre>myFile.close()</pre> <p><i>... adds line at end of the file</i></p> <pre>lines = ["line1", "\n"line2"] myFile.writelines(lines)</pre> <p><i>... appends multiple lines at end of the file</i></p> <pre>myFile.readline()</pre> <p><i>... returns the next line in the file</i></p> <pre>print(myFile.read())</pre> <p><i>... returns and outputs every line in the file</i></p>	<pre>myFile = open("sample.txt") myFile.close()</pre> <p><i>myFile.writeLine("Add this") ...adds line at end of the file</i></p> <p>N/A</p> <pre>myFile.readLine()</pre> <p><i>...returns the next line in the file</i></p> <pre>while NOT myFile.endOfFile() print(myFile.readLine()) endwhile</pre> <p><i>...returns and outputs every line in the file</i></p>
<p>ARRAYS</p> <p>Declaring an array</p> <p>Selecting items in 2D arrays</p>	<p><i>Note: "arrays" have a fixed length and every element of the array must be of the same data type. By contrast, Python "lists" can have items appended or removed. They also allow different data types to be added.</i></p> <pre>names = [""]*3 total = [0]*3</pre> <pre>studentResults [2] [4]</pre> <pre>len(array)</pre> <p><i>...gives the number of items in the list</i></p>	<pre>array names = [3] array total = [3]</pre> <pre>studentResults [2,4]</pre> <pre>array.length</pre> <p><i>...gives the number of items in the array</i></p>
Sub programs: procedures & functions	<pre>def procedureName():</pre> <pre>def funuctionName():</pre> <pre>...</pre> <pre> return variableName</pre>	<pre>procedure procedureName()</pre> <pre>...</pre> <pre>endprocedure</pre> <pre>function functionName()</pre> <pre>...</pre> <pre> return variableName</pre> <pre>endfunction</pre>
Random numbers	<pre>myVariable = random.randint(1,6)</pre> <p><i>... creates a random integer between 1 and 6 inclusive.</i></p> <pre>myVariable = random.uniform (-1.0, 10.0)</pre>	<pre>myVariable = random(1,6)</pre> <p><i>... creates a random integer between 1 and 6 inclusive.</i></p> <pre>myVariable = random (-1.0, 10.0)</pre>

	<i>... creates a random real number (float) between -1.0 and 10.0 inclusive.</i>	<i>... creates a random real number (float) between -1.0 and 10.0 inclusive.</i>
--	--	--

Comparison and Logical Operators

Comparison Operators (also referred to as *Relational Operators*)

Operator	What it means	Example
==	Equality operator checks whether both values are the same	<pre>>>> 7==6 False</pre>
!=	Not equal to	<pre>>>> 8!=2 True</pre>
>	Greater than	<pre>>>> 65>12 True</pre>
<	Less than	<pre>>>> 21<12 False</pre>
>=	Greater than or equal to	<pre>>>> 15>=12 True</pre>
<=	Less than or equal to	<pre>>>> 34<=35 True</pre>

Logical Operators (also referred to as *Boolean Operators*)

AND	Logical AND checks whether both conditions are true or false	<pre>>>> x = 6 >>> x > 0 and x < 7 True</pre>
OR	Logical OR checks whether EITHER of the conditions is true	<pre>>>> x = 5 >>> y = 8 >>> x/2==2 or y/4==2 True</pre>
NOT	Logical NOT reverses a Boolean value. In the example $x > y$ evaluates to False; using the logical NOT reverses the evaluation to True.	<pre>>>> x = 5 >>> y = 8 >>> not (x>y) True</pre>

5.1 Exercise in class (no need to code!): Comparison and Logical Operators

- Calculate the following (True or False):
 - $23 \neq 15$
 - $5 + 3 < 10$
- If $a = 3$ and $b = 8$, what are the results of the following statements?
 - $a < b$
 - $6 \geq a$
- If $c = 11$ and $d = 11$, what are the results of the following statements?
 - $c==d$ and $c>21$
 - $\text{not } (c>d)$

Selection

IF Statement

Selection or conditional statements execute a block of code based on the result of some test or condition we have set in the code. We are testing to see whether the result is TRUE or FALSE and we can set different actions depending on the result of our test.

```
if 76 > 23:
    print("76 is greater than 23")

if 15 > 23:
    print("15 is greater than 23")
```

```
>>>
76 is greater than 23
>>>
```

Example:

In this example, the test in the first block of code evaluates to **True** so the print statement is executed. The test in the second example evaluates to **False** so the print statement is NOT executed.

Note: Remember that it is important that your code is correctly indented to avoid syntax errors.

This can be used in simple examples like this, where we are asking for input from the user and checking this against a present condition in our code:

```
mark = int(input("Enter test score: "))
if mark >=50:
    print("Pass")
```

```
>>>
Enter test score: 57
Pass
>>>
```

What happens if I enter a value below 50? We need the code to be able to deal with BOTH **True** and **False** inputs.

IF ELSE Statement

Using an IF ELSE statement I can now use a False code block so that something happens if the test condition does not evaluate to True.

Example:

```
mark = int(input("Enter test score: "))
if mark >=50:
    print("Pass")
else:
    print("Test failed, resit please")
```

```
>>>
Enter test score: 49
Test failed, resit please
>>>
```

I may want my code to check several conditions and proceed with the condition which is true. For example, a different score in the test used above will result in a different grade.

IF and ELIF Statements

```
mark = int(input("Enter test score: "))

if mark <= 49:
    print("Grade D: Please attend resit")
elif mark > 50 and mark <56:
    print("Grade C-needs improvement")
elif mark >=56 and mark <65:
    print("Grade B-good work")
elif mark >=65 and mark <70:
    print("Grade A-well done")
else:
    print("Grade A*- excellent!")
```

The structure of the IF/ELIF statement should follow these rules:

```
If Condition 1 = True:
    execute Code 1
elif Condition 2 = True:
    execute Code 2
else:
    execute Code 3
```

You can test any number of conditions using this method; your code does not have to include an ELSE but, if it does, it must be the last statement.

Nesting IF Statements

We can also use IF/ELIF statements to check sub-conditions in a program:

```
x = 5
y = 8

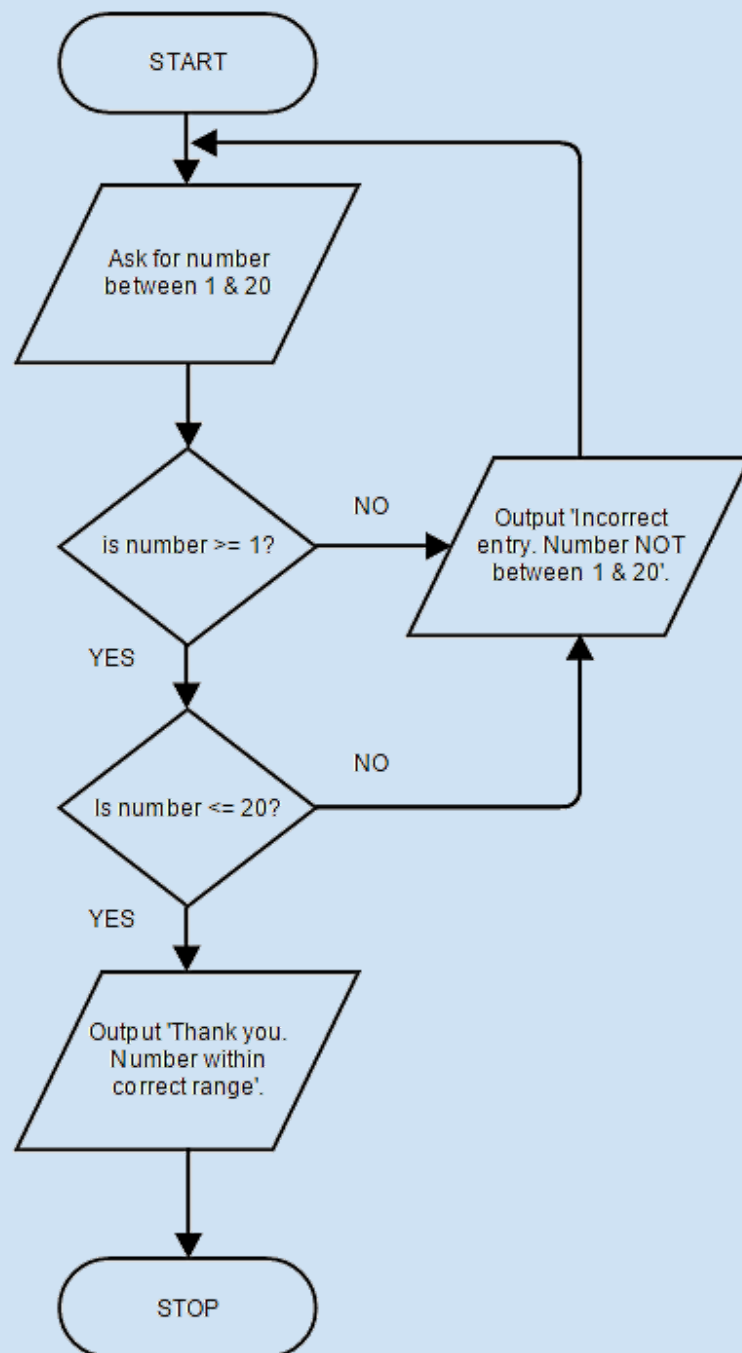
if x == y:
    print("x and y are equal")
else:
    if x < y:
        print("x is less than y")
    else:
        print("x is greater than y")
```

```
>>>
x is less than y
>>>
```

In this example, condition 1 evaluates to False so the ELSE part of the IF statement is executed. Condition 2, in the first part of the nested IF statement, evaluates to true as 5 is less than 8 and the print statement is executed.

5.2 Exercise: IF/ELIF statements

1. **Code the program** shown in the flow chart algorithm.

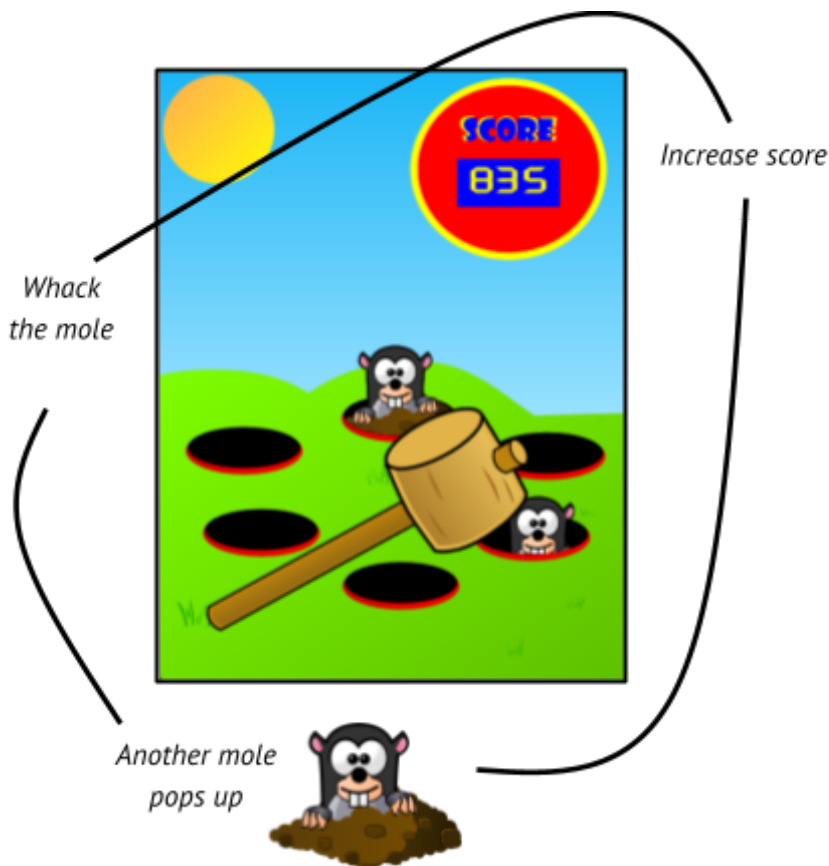


2. **Write a program** which asks for the names of two football teams playing against each other and their scores. Your program should calculate how many points each team gets (3 for a win, 1 for a draw, 0 if they lose).

Iteration

In programming, iteration means repeating instructions or processes over and over again; this commonly known as 'looping'.

Example:



If you do not hit the mole within a certain time limit, the game is over.

In Python there are two types of loop that can be used: a FOR loop and a WHILE loop.

While Loops

WHILE loops are also known as **condition-controlled loops** as they will continue to iterate or loop until a condition, which you have set in your code, is met. It is important to make sure that you have written code that will allow your loop to finish and avoid an infinite loop.

Example:

```
def numberLoop():  
    """ while loop example """  
    number = 1          # initial value of the variable  
    while number <= 10: # the condition to exit the loop  
        print(number)  
        number += 1     # incrementing the value of the  
                        # variable  
  
numberLoop()
```

If the value of the variable 'number' is not incremented by 1 each time the loop iterates, the condition to exit the loop will never be reached as 'number' will always be less than 10.

It can also check a condition entered by a user:

```
def whileInput():
    """while loop example input"""
    answer = 'n'
    while answer != 'y':
        answer = input("Are we there yet? Enter y/n: ").lower()
    print("At last!")

whileInput()
```

```
>>>
Are we there yet? Enter y/n: Almost
Are we there yet? Enter y/n: Nearly
Are we there yet? Enter y/n: Very soon
Are we there yet? Enter y/n: Y
At last!
>>>
```

As you can see, the loop continues until the condition is met and the final print statement is then executed. Remember that use of the `.lower()` built-in function changes my capital letter Y into lower-case y.



We can also use a WHILE loop to check whether a valid input has been entered; this is an ideal way to ensure that your code is robust.

```
3 def menu():
4     '''Display menu choice: Enter Player Names, Play Game ,Quit'''
5
6     menuOptions = ['E', 'P', 'Q']
7     invalidMenuChoice = True
8     print('Please choose from the following options:\n
9         \tPress 'E' to enter players names\n
10        \tPress 'P' to play the game\n
11        \tPress 'Q' to quit\n')
12     menuChoice = input('>> ').upper()
13
14
15     while invalidMenuChoice:
16         if menuChoice in menuOptions:
17             invalidMenuChoice= False
18         else:
19             print('That was not in the menu,\nplease choose E,P or Q to continue')
20             menuChoice = input('>> ').upper()
21
22     return menuChoice
23
24 def main():
25     """runs all functions"""
26
27     menuChoice = menu()
28     if menuChoice == 'E':
29         print("You have chosen to enter player names")
30     elif menuChoice == 'P':
31         print("You have chosen to play the game")
32     else:
33         print("You have chosen to quit the game")
34
35 main()
```

```
>> x
That was not in the menu,
please choose E,P or Q to continue
>> y
That was not in the menu,
please choose E,P or Q to continue
>> q
You have chosen to quit the game
>>>
```

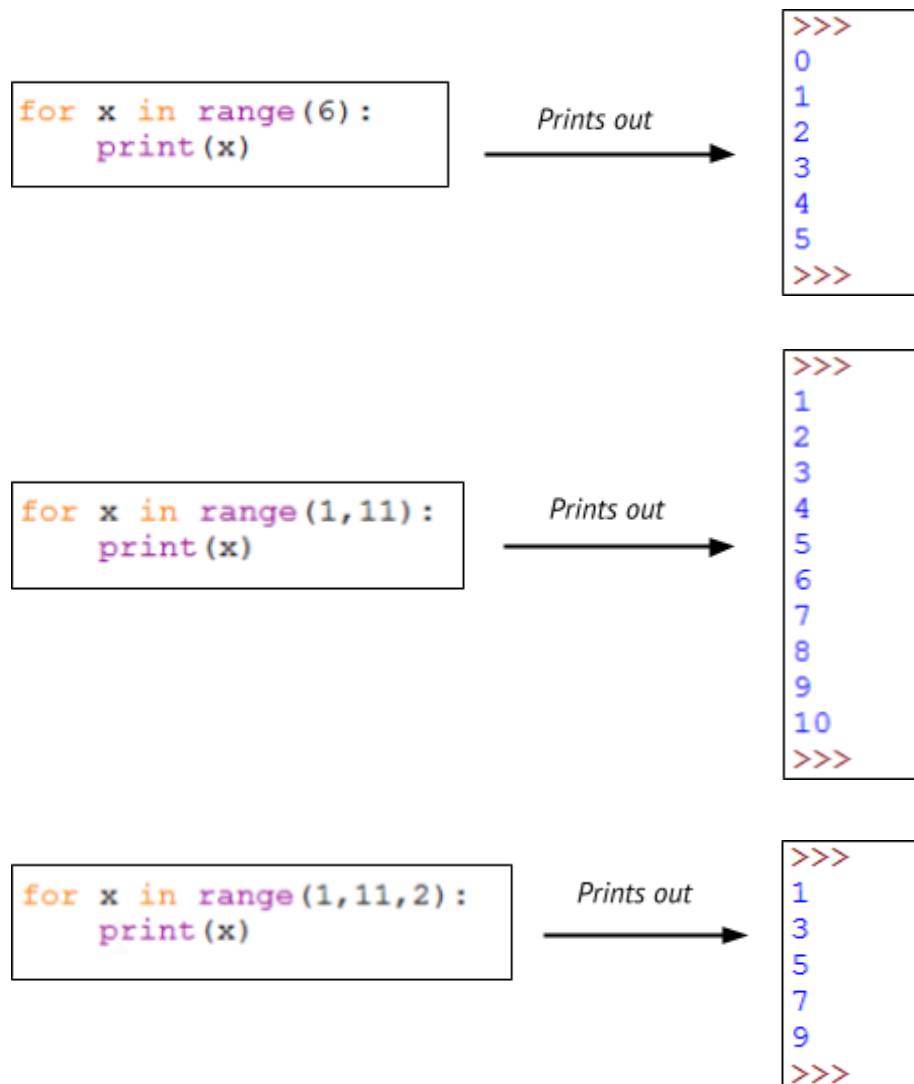
- On Line 7 the variable 'InvalidMenuChoice' is set to **True**.
- On Line 15 the WHILE loop will continue while the value of the variable remains **True**. This could also be written as:

```
while invalidMenuChoice == True:
```
- Line 16 checks whether the input is in the list 'Menu Options'; if that condition evaluates to True then a **valid** menu choice has been entered.
- Line 17 then **changes the value of the variable 'InvalidMenuChoice' to False** so that the program will now **stop looping** through lines 15 to 20.

For Loops

The FOR loop , also known as a **count-controlled loop**, will loop for a set number of times, which can be a number range, e.g. from 1 to 10, or items in a sequence such as a string or a list.

A common way to use a FOR loop with numbers is to use the range() built-in function. When we use this built-in function we usually supply the starting number in the range and the number to go up to, **but not include**, in the range. Look at these examples:



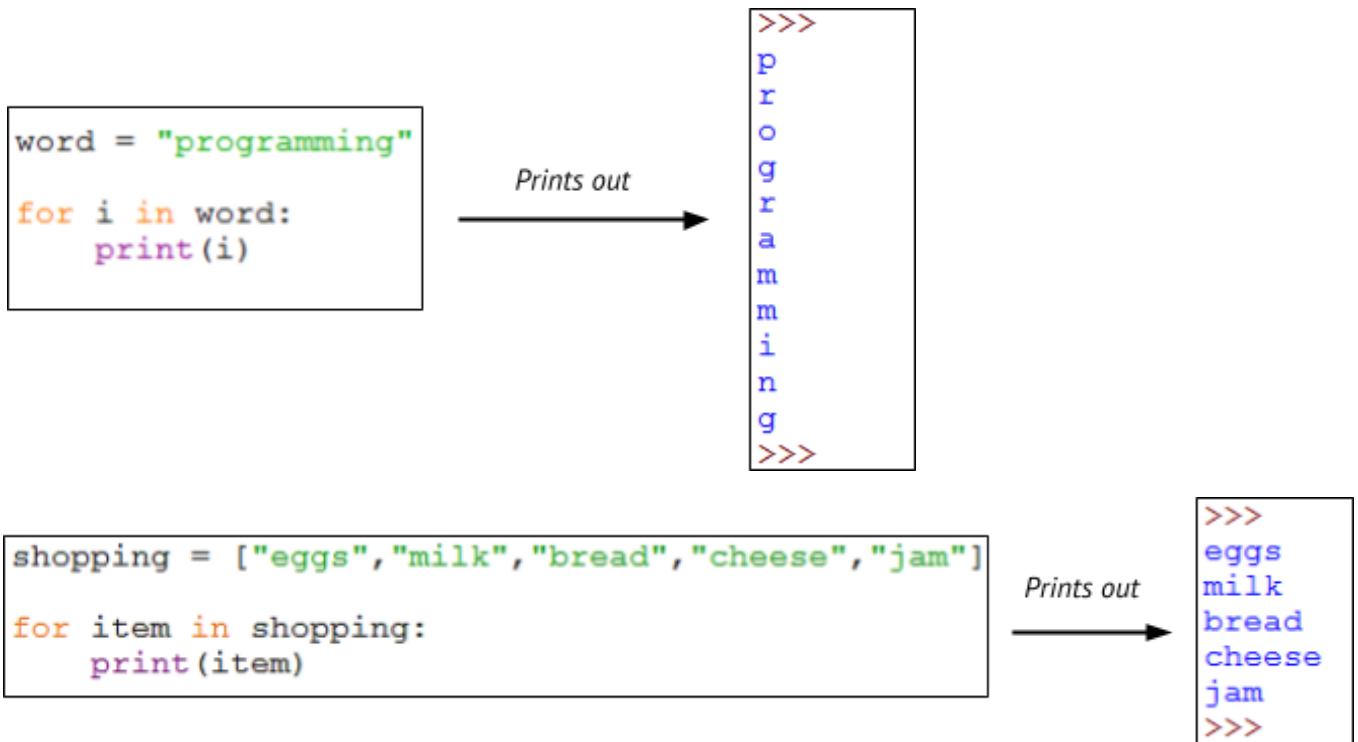
The first example has no starting point for the range of numbers so uses the default of 0 to print out six numbers from 0 to 5.

In the second example I have supplied the starting point 1 and the end of the range as 11; this means up to but not including the last number.

The third example sets the start and end of the range but also specifies that the numbers must increase in steps of 2.

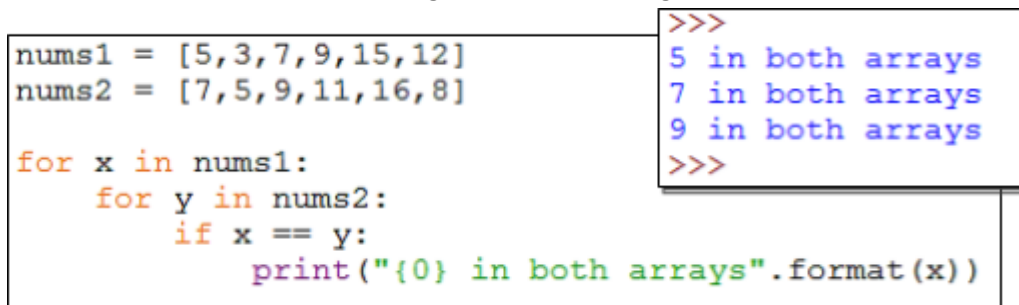
For Loops Using Strings and Lists

Sometimes we want to iterate over a data structure such as a string or a list; we can do this by using a variable 'i' or 'item' to iterate over the characters in the string or the elements in the list/array.



Nested Loops

Sometimes we want to loop through two sets of integers to compare them. For example:



This example finds the integers that appear in both lists; although they are not in the same order, we can identify which are duplicated. We may be writing a program to find duplicate values, so these can be processed in some way to solve a problem.

Example 2:

In this example x is used to count through the range from 3 to 5, and y is used to count through the range from 1 to 13. I have used the string `format()` built-in function to print out a simple times table.

```
for x in range(3,5):  
    for y in range(1,13):  
        print("{0} * {1} = {2}".format(x,y,x*y))
```

```
>>>  
3 * 1 = 3  
3 * 2 = 6  
3 * 3 = 9  
3 * 4 = 12  
3 * 5 = 15  
3 * 6 = 18  
3 * 7 = 21  
3 * 8 = 24  
3 * 9 = 27  
3 * 10 = 30  
3 * 11 = 33  
3 * 12 = 36  
4 * 1 = 4  
4 * 2 = 8  
4 * 3 = 12
```

Example 3:

This example shows how you might run a game loop in a program.

```
3 noWinner = True
4 playerGo = 1
5
6 while noWinner: # outer loop
7     while playerGo == 1: # inner loop
8         name = input("Please enter your name: ")
9         choice = input("Please enter a vowel {0}: ".format(name))
10        if choice.upper() == "I":
11            noWinner = False
12            print("Well done {0}, you have won!".format(name))
13            break
14        else:
15            print("You have not won the game, your turn is over")
16            playerGo = 2 # playerGo variable changed to 2
17    while playerGo == 2:
18        name = input("Please enter your name: ")
19        choice = input("Please enter a vowel {0}: ".format(name))
20        if choice.upper() == "I":
21            noWinner = False
22            print("Well done {0}, you have won!".format(name))
23            break
24        else:
25            print("You have not won the game, your turn is over")
26            playerGo = 1 # playerGo variable changed to 1
27
28 print("Game over")
```

The two conditions that are being checked in each WHILE loop, noWinner and playerGo, are set at the start on lines 3 and 4.

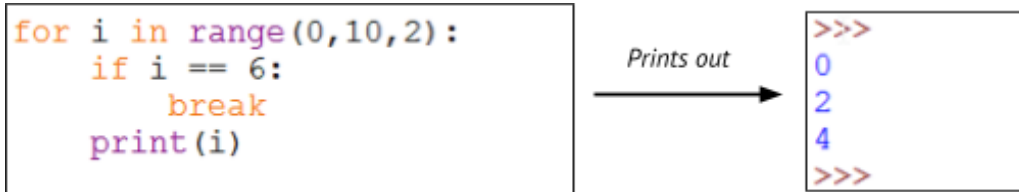
- Line 6 – the **outer loop** continues to check whether **noWinner** is still **True**.
- Line 7 – the inner **while** loop, to control the turn for each player, asks for their name and their choice. This is then compared with the answer on Line 10.
- If the choice entered matches the answer, the game has been won.
- Line 11 – the variable **noWinner** is set to **False**, a message is printed and the '**break**' command forces the code out of the inner loop.
- The condition for the outer loop is **no longer True** and the code jumps to Line 28.
- If the choice entered does not match the answer, the '**else**' part of the selection statement is executed, a message is printed on Line 15 and the playerGo variable is changed to 2 on Line 16.
- The **second inner loop then works in exactly the same way**.

Breaking out of Loops

A break statement will allow us to 'break' out of a WHILE loop if a test condition is met. In Example 3 above this happens if the player correctly guesses the vowel.

Here are two more examples:

Example 1:



This FOR loop should print from 0 to 8, in steps of 2, but is set to break out of the loop if the iterator 'i' is equal to 6.



In this example the use of True in the WHILE loop means that until the input meets the criteria to break out of the loop, the program will continue to ask for an input. In the example above, no input was entered apart from hitting the ENTER key.

5.3 Exercise: While and For loops

1. Write a program, using at least two functions AND parameter passing, which will allow the user to input a number between 1 and 12 (**validate** it is in that range!) and print the **times table** for that number.
2. Write a program, using at least two functions AND parameter passing, that will add together a series of numbers until the user enters 0. The program will then display the total.

Remember parameter passing, e.g. below, look how "name" was returned and then declared with a function call:

```
File Edit Format Run Options Window Help  
#ask for the user name and print a welcome message  
  
def get_name():  
    '''get name input and return'''  
    name = input("Please enter your name: ")  
    return name  
  
def print_greeting(name):  
    '''prints greeting with name variable'''  
    print("Hello {0}, welcome to my greeting program.".format(name))  
  
name = get_name()  
print_greeting(name)
```

5.4 OCR Iteration Question:

a). A programmer has written an algorithm to output a series of numbers. The algorithm is shown below:

```
01   for k = 1 to 3
02       for p = 1 to 5
03           print (k + p)
04       next p
05   next k
```

i. Give the first **three** numbers that will be printed by this algorithm.

[1]

ii. State how many times line **03** will be executed if the algorithm runs through once.

[1]

b). Identify **one** basic programming construct that has been used in this algorithm.

[1]

c). Describe what is meant by a variable.

[1]

d). Identify a variable used in the algorithm above

[1]

Chapter 6. Importing Modules

When we are creating our programs we may need to use additional libraries of code which are not available as standard modules. Examples are:

- time and datetime
- random

In order to use these additional modules we need to import them into our program by adding an import statement, by convention at the very top of the code.

Import time and datetime

There are a number of options but these are likely to be more useful for programming project tasks.

- time – measures time in hours, minutes and seconds
- datetime – manipulates dates as days, months and years

```
import time
import datetime

today = datetime.date.today()

print("Today's date : {0}".format(today))

# Ideally I want to format the date as day,month,year
today = datetime.datetime.now().strftime('%d-%m-%Y')
print("Today's date : {0}".format(today))

#Find out the day of the week you were born
birthday = datetime.date(1999,4,10)
print("I was born on a {0}".format(birthday.strftime('%A')))
```

```
>>>
Today's date : 2015-11-27
Today's date : 27-11-2015
I was born on a Saturday
>>>
```

The date is displayed using the `strftime()` format and there are a number of additional 'directive' options for displaying dates and times (*you do not need to remember these, but may be useful in a project!*):

<code>strftime()</code> directive	How it is displayed
<code>%Y</code>	Year as <i>yyyy</i> , e.g. 2024
<code>%y</code>	Year as <i>yy</i> , e.g. 24
<code>%m</code>	Month as <i>mm</i> , e.g. 04
<code>%B</code>	Month name, e.g. April
<code>%b</code>	Month name shortened to 3 characters, e.g. Apr
<code>%d</code>	Day as <i>dd</i> e.g. 31
<code>%A</code>	Day name, e.g. Monday
<code>%a</code>	Day name shortened to 3 characters, e.g. Mon

We can also use the `time.sleep()` function to add a delay by using the current system time on your computer:

```
import time

#create a countdown timer

delay = int(input("How many seconds delay?: "))

start = time.time()
finish = start + delay # the current time + the delay

x = 1
while time.time() < finish: #uses the current time on your computer
    print("{0}...".format(x))
    x+=1
    time.sleep(1) # delay for 1 second
print("Time's up!")
```

You can also add delays using `time.sleep()` and the number of **seconds** delay you want:

```
print("Hello")
time.sleep(2)
print("world!")
```

6.1 Exercise: Using Date and Time

Write a program that will display the day of the week that a user was born on....

- Use functions if you can, to ask for the date, month and year (as integers).
- Display the date to the user in this format: 15/Jan/97, and tell them what day of the week they were born on.

Import Random

You can use the `random.randint()` function to generate a random number from a given range:

```
1 import random
2
3 def diceRoll():
4     x = random.randint(1,6)
5     print("Your dice says", x)
6
7 diceRoll()
```

Your dice says 6

You can use the `random.randint()` function to choose an item from a list using the index value.

Example:

```
1 import random
2
3 names = ["Bob", "Dave", "Stuart"]
4
5 print(names[random.randint(0,2)])
```

Stuart



Below shows an example of “*future proofing*” your code, i.e. writing robust code. I may want to change my program later, adding items to the menu. Can you explain why `(0, len(myList)-1)` is used?

Example:

```
import random

menu = ['ham', 'eggs', 'cheese', 'tomatoes', 'bread', 'butter']

choice = random.randint(0, len(menu)-1)

print(menu[choice])
```

6.4 Exercise: Random Options

Write a program for a restaurant where the customers are given a random meal.

You will need **lists** for the *starters* and *main course* with at least 3 items in each of your lists.

The program must generate a random meal with a choice from each menu option and display this to the customer in a neat way.

Chapter 7. Scope of Variables in Functions

The scope of a variable, constant or function in a computer program is the place where it can be used.

There are two different types of scope that you need to know about:

Local	This means the variable (or constant) is only in scope, and available for use, inside the sub program where it is defined.
Global	This means the variable is available for use throughout the program.

When you are writing code you should try to avoid using global variables as it is not good programming practice.

The first two variable assignments below, $x = 7$ and $y = 9$, are in global scope for the whole of the program as they have been created outside of a function or procedure.

The second set of variables, x and y , are assigned values inside the function and have local scope in that block of code.

When we run the code, Python will always look to see whether there are local variables before using global variables.

Example 1:

```
#These variables are in GLOBAL scope
x = 7
y = 9

def sum(x,y):
    '''adds x & y and prints result'''

    #These variables are in LOCAL scope
    x = 65
    y = 24
    print("The total of local variables x & y is {0}".format(x+y))

sum(x,y)

print("The total of global variables x & y is {0}".format(x+y))
```

```
>>>
The total of local variables x & y is 89
The total of global variables x & y is 16
>>>
```


Let's look at this example where only global variables are used:

Example 2:

```
#These variables are in GLOBAL scope
x = 7
y = 9

def sum(x,y):
    '''adds x & y and prints result'''

    #These variables are in LOCAL scope
    ##    x = 65
    ##    y = 24
    print("The total of local variables x & y is {0}".format(x+y))

sum(x,y)

print("The total of global variables x & y is {0}".format(x+y))
```

The local variable assignment is now commented out so when the code is executed, Python will not find any local variables inside the function. The function parameters, x and y, will be the global variables assigned at the start of the code.

The results given in the print statements are now the same, as the function uses the global variables:

```
>>>
The total of local variables x & y is 16
The total of global variables x & y is 16
```

7 Exercise: Function with local variables

Create a function (which returns a value) which converts miles to kilometres.

(There are 1.61 km in a mile!)

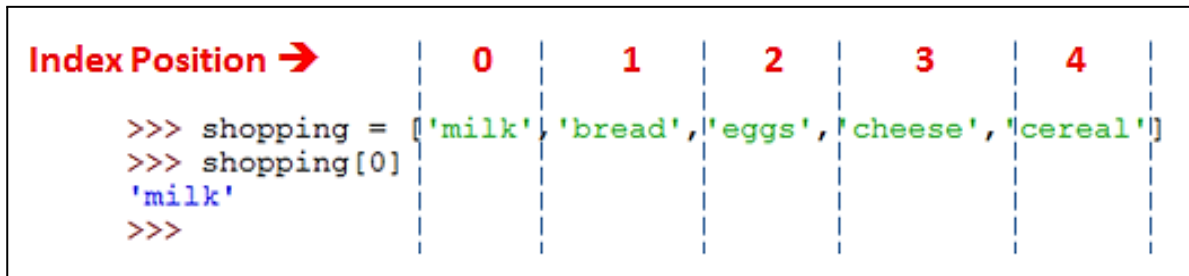
It should ask for a number of miles and output the number of km in an appropriate print statement.

It should only use local variables!

Chapter 8. Arrays/Lists

An array is a data structure that allows us to store multiple items using just one identifier. An array in Python is known as a **list** and there are a number of different methods we can use to manipulate and access data inside the list.

We can think about the data items in our list being contained in separate cells in computer memory; this means we can directly access each element in the list using its **index position**.



- The index position starts counting at 0.
- An “**array**” must contain only one data type. Python only has **lists** which can contain different data types such as strings, integers and floats.

Other ways to access the data in a list:

Slice Lists

Selects the items from the start of the list to INDEX POSITION 3

```
>>> shopping = ['milk', 'bread', 'eggs', 'cheese', 'cereal']  
>>> shopping[:3]  
['milk', 'bread', 'eggs']
```

Selects the items from INDEX POSITION 2 to the end

```
>>> shopping = ['milk', 'bread', 'eggs', 'cheese', 'cereal']  
>>> shopping[2:]  
['eggs', 'cheese', 'cereal']
```

Selects the items in the list from INDEX POSITION 1 and up to (but not including) INDEX POSITION 4

```
>>> shopping = ['milk', 'bread', 'eggs', 'cheese', 'cereal']  
>>> shopping[1:4]  
['bread', 'eggs', 'cheese']
```



8.1 Exercise: Slices

In Python, create a list of the months of the year using abbreviations 'Jan', 'Feb', etc.

SLICE the list to create the following results, using 4 different `print()` statements:

```
>>>
['Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug']
['Jan', 'Feb', 'Mar', 'Apr']
['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov']
['Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']
```

We can use a range of **list operations**, e.g.:

- Concatenate (add two or more lists together)
- Insert items into a list (using the index position)
- Add items to the end of a list (append)
- Check whether items are in a list
- Find the length of the list
- Delete an item from a list
- Sort a list

Concatenate Lists

```
>>> a = [3, 7, 45, 98]
>>> b = [13, 17, 2, 1, 9]
>>> c = a+b
>>> c
[3, 7, 45, 98, 13, 17, 2, 1, 9]
```

Add Items to a List

Append means to add at the end!

```
>>> all_friends.append('Leah')
>>> all_friends
['Aaron', 'Billy', 'Charlie', 'Claire', 'Danny', 'Emily', 'Jamie', 'Jenny',
 'Keiran', 'Sam', 'Tom', 'Leah']
...
```

Check Whether an Item is in the List

This uses the `'in'` operator to find out whether the value is in the list. Make sure you match the case:

```
>>> all_friends
['Aaron', 'Billy', 'Charlie', 'Claire', 'Danny', 'Emily', 'Jamie', 'Jenny',
 'Keiran', 'Sam', 'Tom', 'Leah']
>>>
>>> 'Charlie' in all_friends
True
>>> 'CHARLIE' in all_friends
False
```

Find the Length of a List

Use the `len()` built-in function. Although the **index position** will start counting from 0, the **number of items** in the length of the list will start counting at 1.

```
>>> students = ['Tom', 'Leah', 'Jamie', 'Holly', 'Charlie', 'Zoe', 'Aaron']
>>> len(students)
7
```

Delete an Item from a List

This will delete the first occurrence of the item from your list:

```
>>> students = ['Tom', 'Leah', 'Jamie', 'Holly', 'Charlie', 'Zoe', 'Aaron']
>>> students.remove('Zoe')
>>> students
['Tom', 'Leah', 'Jamie', 'Holly', 'Charlie', 'Aaron']
```

Sort a List

We can use the same built-in function to sort lists with strings and integers or floats:

```
>>> students = ['Tom', 'Leah', 'Jamie', 'Holly', 'Charlie', 'Aaron']
>>> students.sort()
>>> students
['Aaron', 'Charlie', 'Holly', 'Jamie', 'Leah', 'Tom']

>>> numbers = [7, 89, 3, 4, 56, 71, 10, 23]
>>> numbers.sort()
>>> numbers
[3, 4, 7, 10, 23, 56, 71, 89]
```

This also works with a list of mixed integers and floats.

```
>>> mix = [3.6, 78, 34.6, 2, 19.25]
>>> mix.sort()
>>> mix
[2, 3.6, 19.25, 34.6, 78]
```

8.3 Exercise: List practice

Ask a user to enter numbers until they enter 0.
Add the numbers into a list (you will need to declare an empty list to append into).
Sort the numbers into order and print the sorted list.

Using Lists in Programming Projects: Storing Menu Choices

You may be asked to create a menu that is displayed when the program is run and provides the user with options. Here is a simple example:

```
File Edit Format Run Options Window Help
# use print to display the menu

print("\t\tGame Menu\n")
print("\t\tA - Enter Name\n\t\tB - Play Game\n\t\tC - Quit")

selection = input("Please enter your choice: ")
```

Questions: 1. What does `\n` do? 2. What does `\t` do?

The current menu will allow me to enter any value but I want to restrict this to A, B or C only.

I can use a list to check whether the user input is a valid menu option by using the 'in' operator:

```
valid_option = ['A', 'B', 'C']

if selection in valid_option:
    print("That is a valid choice")
else:
    print("That is not a valid choice")
```



8.4 Exercise: Checking items in a list

1. Open a new python file as 'simple_menu' and **copy the code** above (both bits). Enter an **invalid** option to test it works....

Notice that the menu only works **once...** so we cannot make another choice in the case of an invalid selection initially.. The solution is to use a **loop**.

```
1 # use print to display the menu
2
3 print("\t\tGame Menu\n")
4 print("\t\tA - Enter Name\n\t\tB - Play Game\n\t\tC - Quit")
5
6 valid_option = ['A','B','C']
7
8 while True:
9     selection = input("Please enter your choice: ").upper()
10    if selection in valid_option:
11        print("That is a valid choice")
12        break
13    else:
14        print("That is not a valid choice")
```

The program will continue to ask for a choice until one of the three items in the `valid_option` list has been selected. When "A", "B" or "C" is entered, the print statement on Line 11 is executed followed by the **break** statement on Line 12 which **stops** the WHILE loop.

Why is a WHILE loop used here and not a FOR loop?

2. **Amend** your "simple_menu" to run the menu **inside a function** and **return** the selection variable.
3. Use a `main()` function to run the following program:
 - a. When **A** is selected, the program should ask for your **name** and **print a welcome message** including the user's name. The menu should then be **displayed again**.
 - b. When **B** is selected, the program should simply print '**Game is starting**'. The menu should then be **displayed again**. (We will add more functionality here in a bit....!)
 - c. When **C** is selected, the program should print '**Thank you for playing**' and **finish**.

8.5 Exercise: Role-play game inventory management.

Role-play games are very popular and often involve moving around a location, solving puzzles and collecting items in your inventory.. These items can then be stored to use later to help you in the game.

Now add some functionality to the “Play game” option so that you can manage an inventory for the game. Your program must:

- be able to **add** items to your inventory by choosing that option from the menu
- be able to **display** all the items you have in the inventory by choosing that option from the menu
- allow you to **‘get’** an item from the inventory (this means it is no longer in the inventory) by choosing that option from the menu
- display the menu again after each choice until you enter ‘q’

Example:

```
***** Options Menu*****
Enter 'p' to print inventory
Enter 'a' to add item to inventory
Enter 'g' to get item out of inventory
Enter 'q' to quit
Please enter choice: >>
```

Ideally you will use separate functions for each option, passing the updated inventory between the functions.

8.6 OCR Array Question

OCR High School uses a computer system to store data about students’ conduct. The system records good conduct as a positive number and poor conduct as a negative number. A TRUE or FALSE value is also used to record whether or not a letter has been sent home about each incident.

An example of the data held in this system is shown below in Fig. 1:

StudentName	Detail	Points	LetterSent
Kirstie	Homework forgotten	–2	FALSE
Byron	Good effort in class	1	TRUE
Grahame	100% in a test	2	FALSE
Marian	Bullying	–3	TRUE

Fig. 1

A single record from this database table is read into a program that uses an array with the identifier `studentdata`. An example of this array is shown below:

```
studentdata = ["Kirstie", "Homework forgotten", "-2", "FALSE"]
```

The array is zero based, so `studentdata[0]` holds the value "Kirstie".

Write an algorithm that will identify whether the data in the `studentdata` array shows that a letter has been sent home or not for the student. The algorithm should then output either "sent" (if a letter has been sent) or "not sent" (if a letter has not been sent).

[4]

2D Lists/Arrays

So far we have just looked at a “one-dimensional” array. We can also create **two-dimensional** arrays, i.e. **a list of lists**. This is also referred to as a “**matrix**”.

Although “**arrays**” are restricted to a **single data type**, in **Python** you can **mix data types within a 2D list**, making them more versatile. Look at this example:

```
File Edit Format Run Options Window Help
studentResults = [ ['Name', 'HW1', 'HW2', 'HW3', 'HW4'],
                   ['Steven', 8, 7, 9, 4], ['Lauren ', 7, 9, 8, 6]]
```

Accessing Items in a 2D List

We can think of a 2D list as a table with rows and columns.

- Each individual list is a row or “record”.
- Each item in the list is a column or “field”.

To access a particular element in a 2D list, use the index position of the list inside the outer list, and then the location of the data item inside the nested list,

i.e. `list_name [outer_list_index] [inner_list_index]`

Look at the example below:

	0	1	2
studentResults =	['Name', 'HW1', 'HW2', 'HW3', 'HW4']	['Steven', 8, 7, 9, 4]	['Lauren ', 7, 9, 8, 6]
		0 1 2 3 4	

<pre>print(studentResults[1][0]) print(studentResults[2][4])</pre>	<pre>>>> Steven 6 >>></pre>
--	---

8.6 Exercise: 2D arrays.

1. Declare a 2D array (or “matrix”) with dimensions 3x3 and enter some values.
2. Print the entire matrix.
3. Ask a user to specify a position in the matrix, and correctly return the element at that position (example output: “You selected the second list, third item, which is: ____”)

8.7 OCR 2D Array Question

OCR Tech is an online shop that sells electronics such as TVs and game consoles.

Customers can use a discount code to reduce the price of their purchase. Valid discount codes and their value (in pounds) are stored in a global two-dimensional (2D) array with the identifier discount. The following table shows part of this 2D array.

	0	1
0	PVFC7	10
1	CPU5	5
2	BGF2	15

For example, `discount[2,0]` holds discount code BGF2 and `discount[2,1]` holds the discount of 15 pounds.

A function searches through the 2D array and applies the discount to the price. The price and discount code are passed in as parameters. The algorithm design is not complete.

1. Complete the design for the algorithm.

```
function checkdiscount(price, code)

    newprice = price

    size = len(discount) - 1

    for x = 0 to .....

        if discount[x,0] == ..... then

            newprice = ..... - discount[.....]

        endif

    next x

    .....

endfunction
```

[5]

2. Write a program that:

- asks the user for an item price and discount code
- uses the `checkdiscount()` function from part 1 to calculate the price of the item after any discount has been applied
- repeats bullet points 1 and 2 until a price of 0 is entered
- outputs the total cost of all items entered, after any discounts have been applied.

[6]

8.8 Exercise: Using a loop to add into a 2D array.

An 11 year old wants a program which will ask for the names of 5 friends to invite to her party and ask the food they will bring. The program will add the names and food items to a 2D list. At the end it will print the entire matrix as neatly as possible! Try:

1. Declare an empty `party_list`
2. Use a loop to ask for 5 names and 5 items
3. In the loop, create a list containing the name and item, and then append that list to the main `party_list`
4. Print final party list

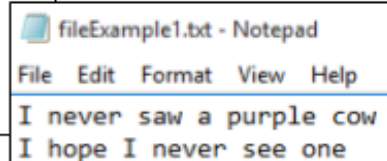
Chapter 9. Reading and Writing Files

None of the programs we have written will store any data **permanently**; the data only exists while the program is running (in RAM). In order to save data from our programs to use again, we need to write the data to a file and save it in secondary storage.

Writing to a File

When we want to write to a file in Python we need to create a 'file object' using the built-in function: `open()`. The `open()` function has two **parameters**: the **file name** and the **mode**. The mode determines whether the file is *read from* or *written to*.

```
def writeFile():  
    """write data to a file"""  
    myFile = open('fileExample1.txt', 'w') #writing mode  
    myFile.write("I never saw a purple cow")  
    myFile.write("\n")  
    myFile.write("I hope I never see one")  
    myFile.write("\n")  
    myFile.close() # ensure that the file is closed
```



fileExample1.txt - Notepad
File Edit Format View Help
I never saw a purple cow
I hope I never see one

Example 1, above:

1. `fileExample1.txt` was created!
2. Lines were written to it with the `write()` function (`writeLine()` in OCR)
3. The file was closed, so it doesn't remain in memory!

Modes for reading from and writing to a file:

Mode	Meaning
'w'	Write mode : used when writing to a file; any existing file with the same name will be deleted
'a'	Append mode : used when opening an existing file to write data; the data is automatically appended to the end of the file
'r'	Read mode : used when the file is to be read into your program

9.1 Exercise: Writing to a file

Define a procedure called "write_to_a_file", which creates a new text file called "`file_handling_rules.txt`" and writes the following 4 lines to it:

File handling rules:

Step 1: open a file object, in the correct mode, and assign it to a variable.

Step 2: write or append lines to, or read lines from, the file.

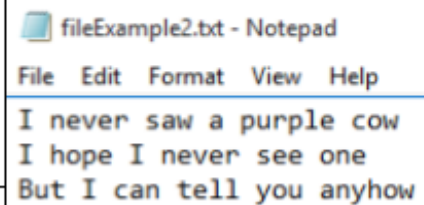
Step 3: remember to close the file!

Call the procedure!

There is a second method you could use which will **automatically close the file object** for us... example 2 shows writing to a file, and example 3 shows appending to a file.....

Example 2. This example is in “write”:

```
def writeFile2():  
    """write data to a file using with statement"""  
    with open('fileExample2.txt','w') as myFile:  
        myFile.write("I never saw a purple cow")  
        myFile.write("\n")  
        myFile.write("I hope I never see one")  
        myFile.write("\n")  
        myFile.write("But I can tell you anyhow")  
        myFile.write("\n")
```



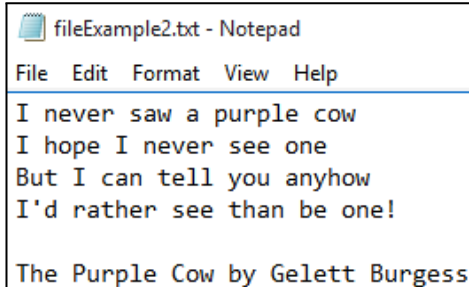
fileExample2.txt - Notepad

File Edit Format View Help

I never saw a purple cow
I hope I never see one
But I can tell you anyhow

Example 3. This example shows the **append mode** of writing to an existing file:

```
def writeFile3():  
    """write data to a file using with statement"""  
    with open('fileExample2.txt','a') as myFile:  
        myFile.write("I'd rather see than be one!")  
        myFile.write("\n")  
        myFile.write("\n")  
        myFile.write("The Purple Cow by Gelett Burgess")  
        myFile.write("\n")
```



fileExample2.txt - Notepad

File Edit Format View Help

I never saw a purple cow
I hope I never see one
But I can tell you anyhow
I'd rather see than be one!

The Purple Cow by Gelett Burgess

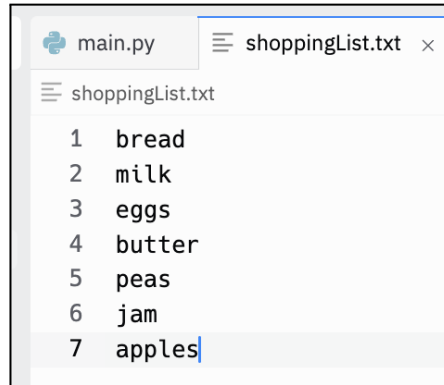
9.2 Exercise: Appending to a text file

Using this “with open... as...” process, append the following line to the external file you created earlier called “file_handling_rules.txt”:

Note, you could use the “with open... as...” structure to open a file, but OCR doesn’t do that!

Reading from a File

Here is a shopping list in a text file:



```
main.py shoppingList.txt x
shoppingList.txt
1 bread
2 milk
3 eggs
4 butter
5 peas
6 jam
7 apples
```

To handle this using Python, we need to create a 'file object', just as we did when writing to a file. Here is some code which reads lines from the file (note mode "r", and puts them into a list in Python called shoppingListArray:



```
main.py
1 with open("shoppingList.txt", "r") as File2:
2     shoppingListArray = File2.readlines()
3
4 print(shoppingListArray)
```

The print statement on line 4 outputs this:

```
['bread\n', 'milk\n', 'eggs\n', 'butter\n', 'peas\n', 'jam\n', 'apples']
```

You could iterate through the list printing each item like this:

```
4 for each in shoppingListArray:
5     print(each)
6
```

```
bread
milk
eggs
butter
peas
jam
apples
```

Not only does this include the newline (`\n`) character from the text file which was then added to the list,, but it includes an extra, default, `\n` character meaning a line is missed each time! This can be replaced with `**nothing**` like this:

```
4 for each in shoppingListArray:
5     print(each, end="")
6
```

```
bread
milk
eggs
butter
peas
jam
apples
```

9.3 Exercise: Reading from files

1. **Create** a text file with the first names of the students in your class and save it as '*students.txt*'.
Read the file into a list
Sort the list into alphabetical order (`list.sort()` will suffice!)
Print out the list items on separate lines (with no extra return characters... as above!)

9.4 OCR External Files Question

A library gives each book a code made from the first three letters of the book title in upper case, followed by the last two digits of the year the book was published.

For example, "Poetry from the War", published in 2012 would be given the code POE12.

- (a) (i) Complete the following pseudocode for a function definition that will take in the book title and year as parameters and return the book code.

```
01 function librarycode(title, _____ )
02     parta = title.substring(0, _____ )
03     partb = year.substring(2, 2)
04     _____ parta.upper + partb
05 endfunction
```

[3]

- (ii) Write an algorithm that does the following :

- Inputs the title and year of a book from the user.
- Uses the `librarycode` function above to work out the book code.
- Permanently stores the new book code to the text file `bookcodes.txt`

[6]

Chapter 10. Validation Techniques



When you are writing your own programs, it is important to ensure that the data entered by the end user is **reasonable** or **sensible** to ensure that your program does not crash or produce unexpected results; this technique is called validation.

Example: You may have registered with a website where you have been asked for your date of birth. This is commonly achieved with drop-down lists which restrict your data entry to the number of days in a month, months in the year and a range of years for people of all ages; however, it will not allow you to be 150 years old as that is not reasonable or sensible!

Validation does not check if the data entered is *correct*, only that the values are reasonable and within any boundaries you have set; e.g. an email address must include the @ symbol.

What could be checked?

- Data entered is the correct data **type**, e.g. a person's age is an integer and a person's name is a string.
- Data entered is within the correct **range**, e.g. that a person's age is not a negative number.
- Data entered is the correct **length**, e.g. telephone numbers.
- Data entered is in the right **format**, e.g. upper case letters for a postcode. We can use these built-in functions to help:
 - `isalpha()` – returns true if all the characters in a string are letters
 - `isdigit()` – returns true if characters in a string are numbers
 - `isupper()` – returns true if characters in a string are upper case
 - `islower()` – returns true if characters in a string are lower case

In the example below, **format** and **length** validation is used. The code checks that a name input from the user is **letters only** and that the string is at least **two characters in length**.

```
1 def getName():
2     """asks for user's name"""
3     invalidName = True                #variable to control the while loop
4     while invalidName:
5         name = input("Enter name :")  #ask for input inside the loop
6         if not name.isalpha():
7             print("Your name can only contain letters.")
8         elif len(name)<2:
9             print("Your name must have more than 2 letters")
10        else:
11            print("Valid name entry")
12            invalidName = False
13            break
14    return name
15
16 def main():
17     """runs all functions"""
18     name = getName()
19     print("Hello {}".format(name))
20
21 main()
```

In this example I have used a variable to control the WHILE loop on Line 4; the loop will continue until the string entered has passed the two validation checks, i.e. that the string for the username is more than two letters and has no numbers in it. On Line 6 I have used `isalpha()` with the logical operator NOT; this will evaluate to True if the string contains letters and numbers.

When those **two checks** have been passed in the ELSE part of the selection statement, I need to change the Boolean value of the variable controlling the loop to False and break out of the loop at that point.

Notice on Line 4 that I do not need to use any **equality operator** in my WHILE statement when I am using **Boolean values**.

The `isdigit()` built-in function **only** applies to strings; if you wanted an integer or float input you would cast the input to an integer or float.

How could I use `isdigit()` in my code?

Look at this example:

```
1 def getMobileNo():
2     """gets mob phone number"""
3     invalidNo = True
4     while invalidNo:
5         telNumber = input("Enter phone number: ")
6         if len(telNumber) in range(9,12) and telNumber.isdigit():
7             print("Valid entry")
8             invalidNo = False
9             break
10        else:
11            print("The number must be between 9 and 11 digits")
12    return telNumber
13
14 def main():
15     """runs all functions"""
16     telNumber = getMobileNo()
17     print("You entered {}".format(telNumber))
18
19 main()
```

When we want to store telephone numbers or mobile phone numbers these commonly start with a 0; if I use a number data type then the leading 0 will be lost. As you can see on Line 6 I have combined my checks so that the number entered must be **between 9 and 11 characters in length AND all numbers**.

Another example:

Using a list to hold valid options and comparing data input with the list.

```
def display_menu():
    """displays the menu and asks for play choice"""
    # use print to display the menu

    print("\t\tGame Menu\n")
    print("\t\tA - Enter Name\n\t\tB - Play Game\n\t\tC - Quit")

    valid_option = ['A', 'B', 'C']

    while True:
        selection = input("Please enter your choice: ").upper()
        if selection in valid_option:
            break
        else:
            print("That is not a valid choice")
    return selection
```

In this example, the input is forced to upper case so that we are comparing the same character. This shows a slightly different use of the WHILE loop to control the validation of the input character.

The section on *Handling Errors*, below, also provides a range of options that allow you to write robust code to deal with errors through what is known as **defensive programming**.

10. Validation Exercise

In a simple grading system, a student gets a 9 with over 90%, 8 with over 80%, 7 with over 70%, etc.

Write a program which asks for a grade, and tells the user what grade they got.

Add validation so that the program only accepts **digits** (doesn't crash with stings) and **valid percentages**.

1. Use a function to "get_test_score()" which **returns** a value but does not print the grade.
2. **Declare** a variable e.g. "test_score" as you call the function and output the grade.

Ensure that the user cannot continue until a valid value has been entered.

Output a suitable message if a value does not meet format or range criteria.

Chapter 11. Dealing with Errors

There are three main types of error that you may come across when programming:

Error Type	Explanation and examples
<i>Syntax</i>	<p>This means that your code does not follow the rules of the programming language. You may have missed a bracket, forgotten to add speech marks or have a spelling error in a key word.</p> <pre>>>> Print("Hello World!") Traceback (most recent call last): File "<pyshell#1>", line 1, in <module> Print("Hello World!") NameError: name 'Print' is not defined</pre>
<i>Runtime</i>	<p>This means that when the program runs an error occurs. Examples could be trying to divide a number by 0 or trying to open a file that does not exist.</p> <pre>>>> Traceback (most recent call last): File "C:\Users\ExampleFiles\readFileExample3.py", line 9, in <module> readFile3() File "C:\Users\ExampleFiles\readFileExample3.py", line 5, in readFile3 with open('shopping.txt','r') as myFile: FileNotFoundError: [Errno 2] No such file or directory: 'shopping.txt'</pre>
<i>Logic</i>	<p>This means that your program will run but will give unexpected results. Examples are using mathematical operators incorrectly, e.g. > instead of < or forgetting to put brackets around a calculation (BIDMAS/BODMAS).</p> <pre>>>> x = 3 >>> y = 4 >>> average = x+y/2 >>> print(average) 5.0 >>> average = (x+y)/2 >>> print(average) 3.5</pre> <p>The answer should be 3.5; we need to ensure that the calculation happens in the right order to correct this logical error.</p>

How to Reduce Logical Errors

These can be harder to spot than syntax or runtime errors as the interpreter will not give you any error messages or warnings.

- Use **print statements** to see how the variable values change at different points in your code.
- **Test** each part of your program separately to pinpoint where the error could be.
- Manually trace the execution of your program using a **trace table**, which can be used to predict, step by step, how the algorithm run.

Handling runtime errors with *robust code*



Some runtime errors have specific names, e.g. `NameError`, `FileNotFoundError`. We can write **robust code** to deal with these named errors using '**exception handling**' to ensure that our program does not just crash but prints a meaningful error message so we can control what happens next.

“ValueError”

In this example the code is checking that the input value is an integer; this is called a TRY statement.

```
def getInput():
    """ get integer input"""
    while True:
        try:
            num = int(input("Please enter a whole number: "))
            print("The number you entered was {}".format(num))
            break
        except ValueError:
            print("That was not an integer, please try again.")

    return num

num= getInput()
```

```
Please enter a whole number: f
That was not an integer, please try again.
Please enter a whole number: 3.25
That was not an integer, please try again.
Please enter a whole number: 6
The number you entered was 6
>>>
```

“ZeroDivisionError”

```
def divideNums():
    """ divide x by y and print result"""
    try:
        x = int(input("Please enter a number to divide: "))
        y = int(input("Please enter a number to divide by: "))
        print("The result of {0}/{1} is {2}".format(x,y,x/y))
    except ZeroDivisionError:
        print("Cannot divide by 0!")
        y = 1
```

```
>>>
Please enter a number to divide: 15
Please enter a number to divide by: 3
The result of 15/3 is 5.0
```

If I try to enter a 0, the interpreter will 'catch' the exception and handle it by printing the error message and changing the value of the variable `y` to 1.

```
>>>
Please enter a number to divide: 15
Please enter a number to divide by: 0
Cannot divide by 0!
```

Input/ Output Error

I/O errors occur when we try to create a file object that does not exist or try to write the file to a storage area which is full. At GCSE level you should be able to write code to deal with the first error, i.e. the file is not there OR the filename supplied is incorrect.

```
def readFile():
    """reads in file line by line and strip \n"""
    try:
        with open('shopping.txt','r') as myFile:
            line = myFile.readline()
            while len(line)!=0: # while there is data in file
                print(line,end = '')
                line = myFile.readline()# read next line
    except:
        print("File does not exist...")
```

```
>>>
File does not exist...
>>>
```

In this example the name of the file is incorrect, the exception is handled and the error message is printed.

11. Exercise: Dealing with errors

1. Insert a **logic error** into a program you have written previously and get a partner to spot it!

2.

- a. What type of error is shown in this code snippet?

```
number = 1
while number < 13:
    print("{0} squared = {1}".format(number,number*number))
```

- b. Correct the code so that it will print the square numbers from 1 to 12.

3. Write some code which tries to read from a file that does not exist, but “catch” the error and display an appropriate message to the user.

Chapter 12. Test Plans and Test Data

Test plans are an essential part of your programming project; this is where you show that the code you have written works as expected and meets the requirements of the task.

Test plans usually follow a common table format such as this:

Test No.	Test Description	Data Input	Expected Outcome	Actual Outcome	Improvements

It is important to test all your code as you complete each function or section of your program. Some of your tests may not require any data input; for example, ensuring that a menu for a game displays the correct options for the game.

You should endeavour to test these categories where possible:

- **Normal** test data: data which should be **accepted** by a program without causing errors.
- **Boundary** test data: data of the correct type which is on the very edge of being valid but should be **accepted**. Sometimes called “*extreme*” values.
- **Invalid** test data: of the **correct data type** which should be **rejected** by a computer system
- **Erroneous** test data: data of the **incorrect data type** which should be **rejected** by a computer system

Each time you create a new test you need to:

- Give a clear **description** of what is being tested
- Specify what the **data input** is and the type of data being entered
- Give a clear description of **what you expect** will happen when the test is performed
- Run the test and take a **print-screen** of the results
- Give a clear **description of what actually happened** and reference the print-screen evidence
- If the test did not work as expected, you need to **explain what improvements** are needed and run the test again to prove that the code now works as expected

Testing at the end of a programming project is known as **final testing** or **terminal testing**.

Testing as the project goes along, or testing part of it, e.g. just one sub-program/ function, is called **iterative testing**, because it tests each iteration or version of the program.

The following pages have some example test plans and testing evidence for a “Magic Square” Game.

Testing a Magic Square Game

Test No.	Test description	Data Input	Expected Outcome	Actual Outcome	Improvements
1	Test the game board displays correctly when the game opens	N/A	The game board will display an 8 × 8 grid with numbers across the top of each column and letters at the start of each row.	The board displays correctly as expected. <i>Shown below.</i>	The instructions for the player should be below the game board so it is clearer.

Results of Test 1:

```

  1  2  3  4  5  6  7  8
A 0  0  0  0  0  0  0  0
B 0  0  0  0  0  0  0  0
C 0  0  0  0  0  0  0  0
D 0  0  0  0  0  0  0  0
E 0  0  0  0  0  0  0  0
F 0  0  0  0  0  0  0  0
G 0  0  0  0  0  0  0  0
H X  0  0  0  0  0  0  0 You can move your player up, down or stay on the same row
.Please enter U,D or S:

```

Test No.	Test description	Data Input	Expected Outcome	Actual Outcome	Improvements
2	Test the game board displays correctly when the game opens.	N/A	The game board will display an 8 × 8 grid with numbers across the top of each column and letters at the start of each row. The player instructions display below the grid.	The board displays correctly as expected. <i>Shown below.</i>	None

Results of Test 2:

```

  1  2  3  4  5  6  7  8
A 0  0  0  0  0  0  0  0
B 0  0  0  0  0  0  0  0
C 0  0  0  0  0  0  0  0
D 0  0  0  0  0  0  0  0
E 0  0  0  0  0  0  0  0
F 0  0  0  0  0  0  0  0
G 0  0  0  0  0  0  0  0
H X  0  0  0  0  0  0  0

You can move your player up, down or stay on the same row.
Please enter U,D or S: |

```

Test No.	Test description	Data Input	Expected Outcome	Actual Outcome	Improvements
3	Test the data input for player vertical movement .	Erroneous Data 'x'	The program will print an error message and ask for input again.	The results of the test are as expected. <i>Shown below.</i>	None

Results of Test 3:

```
You can move your player up, down or stay on the same row.
Please enter U,D or S: x
That is not a valid option

You can move your player up, down or stay on the same row.
Please enter U,D or S: |
```

Test No.	Test description	Data Input	Expected Outcome	Actual Outcome	Improvements
4	Test the data input for player vertical movement direction.	Normal Data 'U'	The program will accept the input and ask how many squares to move.	The results of the test are as expected. <i>Shown below.</i>	None

Results of Test 4:

```
You can move your player up, down or stay on the same row.
Please enter U,D or S: U
Please enter the number of squares to move: |
```

Test No.	Test description	Data Input	Expected Outcome	Actual Outcome	Improvements
5	Test the data input numbers of squares to move.	Extreme Data 7	The program will accept the input and ask whether there is any horizontal movement.	The results of the test are as expected. <i>Shown below.</i>	None

Results of Test 5:

```
Please enter the number of squares to move: 7
You can move your player left, right or stay on the same row.
Please enter L,R or S: s
```


12. Exercise: Testing

1. Insert a **testing table** into your notes. Test a program you have written previously (e.g. Inventory Management program) with **normal**, **boundary**, **invalid** and **erroneous** data.