Pearson Edexcel

Pearson Edexcel Level 1/Level 2 GCSE (9–1) in Computer Science (1CP2) Good Programming Practice Guide

First teaching September 2020
First certification from 2022
Issue 1.2

Contents

Introduction

The Programming Language Subset (PLS) is a document that specifies which parts of Python 3 are required in order that the Paper 2 assessment for GCSE Computer Science can be undertaken with confidence. Students familiar with everything in that document will be able to access all parts of the Paper 2 assessment. This does not stop a teacher/student from going beyond the scope of the PLS into techniques and approaches that they may consider to be more efficient or engaging. Pearson will **not** go beyond the scope of the PLS when setting assessment tasks.

This Good Programming Practice Guide (GPPG) is a document that expands on the content of the PLS, by providing more in-depth information and a wide range of examples.

The information in this document should not prevent students or teachers from developing their own style and techniques. Teachers are encouraged to amend the examples in this document to suit the needs of their own students.

While students will have a copy of the PLS available during the Paper 2 exam, they will not be allowed a copy of the GPPG.

Please read this document together with the Programming Language Subset (PLS).

The PLS contains the definitions for programming constructs and subprograms.

This document contains explanations and examples for them. The definitions are not repeated in this document.

Readability

(Specification point: 6.1.4)

It is good programming practice to use techniques to ensure that code is easy to read, understand and maintain.

Layout

Code files should be laid out consistently to help others read the program code and increase its maintainability. One way this can be done is by grouping statements that are related to each other and preceding each group with a comment indicating what it holds. Here is one layout that works well to help organise code. It is the layout used in when setting Paper 2 assessment tasks.

```
# -------------------------------------------------------------------
# Import libraries
# -------------------------------------------------------------------

# -------------------------------------------------------------------
# Constants
# -------------------------------------------------------------------

# -------------------------------------------------------------------
# Global variables
# -------------------------------------------------------------------

# -------------------------------------------------------------------
# Subprograms
# -------------------------------------------------------------------

# -------------------------------------------------------------------
# Main program
# -------------------------------------------------------------------

```

Starting with this layout and placing statements in the correct sections will make the code more readable and easier to debug. In addition, adhering to a layout such as this can help students avoid unintended use of nested subprogram definitions and using an excessive number of global variables.

Comments

In Python, anything on a line after the '#' character is considered a comment. Comments may appear on the same line as, but after, code. They can also appear on a line all by themselves.

Students should be encouraged to use comments to explain the logic of their code, especially of code blocks, such as 'if…elif…else' and 'for…in range()', and subprograms. However, adding a comment to every line is overkill, as are comments that simply duplicate the code.

Excessive commenting simply adds visual clutter and makes code more difficult to read.

Identifiers

Students should be encouraged to select variable names and subprogram interfaces that are meaningful in the context of the problem. They should avoid using single letter identifiers.

There are many different ways to format identifier names. One way is to use camel case. This is where the names of variables and subprograms are started with a lowercase letter. Each change of word or abbreviation begins with an uppercase letter.

This table shows examples of using camel case for identifier names.

`count`	`bigList`	`up_counter`	`countAnimalTypes`	`isValidCounter()`
`name`	`primaryKey`	`last_name`	`studentFirstName`	`findStudentName()`
`key`	`discountCode`	`blue_green`	`dbForeignKey`	`showKeys()`

Camel case is the method used when setting Paper 2 assessment tasks. However, it is not obligatory for students to use this method.

White space

Good use of white space also helps make programs easier to read. Students should be encouraged to use white space to separate code blocks. (See also the Blocking section.)

Double spacing code is not helpful and should be avoided.

The logic of a line of code can sometimes be more easily understood if a few extra spaces are introduced. This is especially helpful if a long line of nested subprogram calls is involved. It can be difficult to see where one ends and another begins. The syntax of Python is not affected, but it can make understanding the code much easier.

This table shows an example of a short program that is difficult to read and how extra spaces can be used to aid readability.

Difficult to read

```
word=input("Enter a long word: ")
substring=input "Enter a substring: ")
if((word.upper().find(substring.upper(),0,len(word)-5))!=-1):
    print("Found")
else:
    print("Not found")
```

Addition of extra spaces and blank line

```
word = input ("Enter a long word: ")
substring = input ("Enter a substring: ")

if ((word.upper().find (substring.upper(), 0, len (word) - 5)) != -1):
    print ("Found")
else:
    print ("Not found")
```

Line continuation

Long code lines may be difficult to read, especially if they scroll off the edge of the display window. It is always better to limit the amount of scrolling.

Python syntax allows long lines to be broken inside brackets (), square brackets [], and braces {}. This works very well, but care should be taken to ensure that the next line is indented to a level that aids readability. It is even possible and recommended to add an extra set of brackets () to expressions to break long lines.

Here is an example of a line that has been broken around logical operators and between the outermost brackets.

```
if ((userChoice != "Q") and (userChoice != "A") and
        (userChoice != "B") and (userChoice != "C") and
        (userChoice != "D") and (userChoice != "E")):
```

Here is an example of a line that has been broken between square brackets.

```
newRecord = [idNum,
             lastName, firstName,
             birthMonth, birthYear]
```

Python also has a line continuation character, the backslash '\'. It can be inserted, following strict rules, into some expressions to cause a continuation. Some editors will automatically insert the line continuation character if the enter key is pressed.

Here is an example of a line that has been broken using the continuation character.

```
nextRecord = treeNames[index] + "," + \
             str (treeCounts[index]) + "," + \
             str (treeScore[index])
```

Data types and conversion

(Specification points: 6.2.1, 6.3.1, 6.3.2)

Primitive data types

The four primitive data types are integer, real, Boolean and character. In Python these are designated as ‘int’, ‘float’, ‘bool’ and ‘str.’ (Note, Python does not have a dedicated character data type so uses ‘str ‘instead.)

Variable declaration

It is important that students understand the difference between declaring and initialising variables. Declaration allocates memory based on the size of the indicated data type. Initialisation sets the contents of the allocated memory.

Variables may be explicitly assigned a data type during declaration or, as is most commonly done in Python, may be implicitly typed via the data type of the first value assigned to them (see below).

It is good practice to set the data type of a variable during the initial creation phase to help document the logic and understanding of the solution.

Once a variable has been associated with a data type, its type should not be changed to a different one during the life of the program.

This table shows examples of variable declarations that allocate memory, based on the size of the data type indicated.

Data type	Explanation	PLS	Example declaration
integer	A whole number (negative or positive)	int	`count = int ()`
real	A number with a fractional part, also known as decimal (negative or positive)	float	`thePrice = float ()`
Boolean	Data that can only have one of two values, either true or false	bool	`lights = bool ()`
character	A single letter, number, symbol, etc., usually available from the keyboard	str	`myInitial = str ()`

Combining declaration and initialisation

Implicit data typing is done by associating an implied data type with the variable when it is assigned its initial value.

This table shows examples of implicit data typing, achieved by assigning an initial value.

Example	Description
`TAX = 0.175`	A real constant initialised to 0.175
`count = 0`	An integer variable initialised to 0
`windowOpen = True`	A Boolean variable initialised to True
`myInitial = "J"`	A character variable initialised to "J"

Conversion

Conversion is used to transform the type of an expression before processing it.

Here is an example of conversion:

```
count = int (input ("Enter a number"))
```

Technically the 'input()' function returns the string value that the user typed in. However, the string value is immediately converted to an integer by the 'int()' function. It can then be used in calculations.

Here is another example of conversion:

```
balance = float (count)
```

In this example, a copy of the contents of the variable 'count' is taken. The copy is converted to float, a decimal number. The original contents of 'count' has not changed, nor has the original data type of 'count' changed from an integer to a float. The variable 'balance' is the only variable with a data type of float.

Constants

Constants are identifiers that are set only once in the lifetime of the program. The use of constants aids readability and maintainability.

Constants are conventionally named in all uppercase characters. This identifier name is then used to represent the constant value throughout the program code. Should at any stage this value need to be altered, the change only has to be made at one location in the whole program code.

This table shows examples of declaring and initialising constants.

`TAX = 0.175`	`CORNERS = 4`	`TITLE = str ()`
		`TITLE = "Accounts"`
			`MAX_COUNT = 34`

Although Python does not support constants in the same way as other high-level languages, students are expected to use constants in their program code. This means that they should:

- write constant names in capital letters to distinguish them from variables
- assign values to constants at the start of the program and ensure that these values are not changed during execution of the entire program.

Structured data types

(Specification points 6.2.1, 6.3.1, 6.3.3)

A structured data type is a collection of items, which themselves are typed. Each item in the collection is accessible using an index. Indices start with the value 0.

Arrays and records

This table shows examples of the different structured data types.

Data type	Explanation	PLS	Example

string	A sequence of characters	str	`myName = str ()`
			`myName = "Joseph"`

array	A sequence of the same type item, with a fixed length	[ ]	`myGrades = [80, 75, 90]`
record	A sequence of items, usually of mixed data types, with a fixed length	[ ]	`studentRecord = [1524, "Jones", "Rebecca", True, 78.45]`

In the above table:

- `studentRecord[2]` holds the value “Rebecca”,
- `myGrades[0]` holds the value 80,
- `myName[5]` holds the value “h”.

Note, that the data structure array is not the same as the Python data structure list. However, in the PLS, they are both created using the ‘list’ construct.

If a list holds homogenous data (meaning that the data type is the same for all elements), it can be thought of conceptually as an array. If a list holds heterogenous data (meaning a mixture of data types), it can be thought of conceptually as a ‘record’. The record, in this case, is similar to a record in a database where fields may have different types.

Dimensions

Students are only expected to work with one- and two-dimensional data structures, no more.

A one-dimensional data structure stores elements, each of which can be accessed using a single index value. Whereas, each element in a two-dimensional data structure is itself a one-dimensional data structure. Two indices are needed to access a single item within a two-dimensional data structure. Indices start with the value 0.

Note that students are not expected to work with ragged two-dimensional data structures (meaning ones containing records that do not necessarily have the same number of fields).

This table shows examples of two-dimensional data structures.

Data type	Explanation	PLS equivalent	Example
Two-dimensional array	A collection of one-dimensional arrays. Each one-dimensional array has the same number of elements. Each element is the same data type.	[[ ], [ ], … [ ]]	`sensorReadings = [ [80, 75, 90], [15, 25, 35], [82, 72, 62]]`
Two-dimensional array of records	A collection of one-dimensional records. Each one-dimensional record has the same number of fields. Fields are of mixed data types.	[[ ], [ ], … [ ]]	`recordTable = [ [1524, "Jones", "Rebecca", True, 78.45], [5821, "Lawson", "Martin", False, 23.98]]`

In the above table:

- `recordTable[0]` holds an entire record [1524, "Jones", "Rebecca", True, 78.45], which itself is a one-dimensional structure with mixed data types.
- `sensorReadings[0][1]` holds the value 75.
- `recordTable[1][2]` holds the value "Martin".

Multiple parallel arrays

It is possible to use a number of separate one-dimensional arrays together. The items at the same position could be associated with each other. For example, one array could hold all the items in a kitchen and the second array could hold the count of each item. A single index could then be used to move across both arrays in parallel, processing an item from each of them.

This code shows an example of using two arrays in parallel.

Multiple parallel arrays

```
things = ["cup", "plate", "fork", "spoon"]
counts = [5, 8, 4, 3]

for index in range (len (things)):
    print (things[index], counts[index])
```

Output

```
cup 5
plate 8
fork 4
spoon 3
```

While this approach is acceptable, storing the associated data as a two-dimensional array of records is good practice. When using multiple arrays, there is a greater opportunity for errors to be introduced in the process of adding, deleting, or replacing items. There is also the possibility for the associations to become mismatched, by updating one and not the other. For every amendment, there are multiple operations, any of which could go wrong.

String manipulation

Students should be familiar with common string manipulation techniques, including slicing and concatenation. (See also the String subprograms section.)

Slicing

Python supports string slicing which returns a specific substring of a given list or string. For example, this code outputs the first two characters, i.e. “Jo”.

```
myName = "Joseph"
slice = myName[0:2]
print (slice)
```

Concatenation

Concatenation of strings is done using the ‘+’ operator. (See also the Screen output section.)

Multiplying a string is done using the ‘*’ operator. Multiplying a string by an integer *n* concatenates the string with itself *n* times.

Operators

(Specification points: 6.5.1, 6.5.2, 6.5.3)

Arithmetic operators

This table shows examples of the arithmetic operators.

Operator	Operation	Example
/	division	`total / number` Always returns a real number
*	multiplication	`count * 7`
**	exponentiation (raising to the power)	`radius ** 2` The same as radius^2
+	addition	`total = total + 1`
-	subtraction	`difference = total - count`
//	integer division (integer part of result) 5 // 3 is 1	`number = total // count`
%	modulus (remainder after division) 5 % 3 is 2	`number = total % count`

BIDMAS

Students need to know the order of precedence rules that determine the order in which a calculation is executed. BIDMAS stands for Brackets, Indices (exponentiation), Division, Multiplication, Addition, Subtraction.

Relational operators

This table shows examples of the relational operators.

Operator	Operator meaning	Example	Evaluates to
==	Equal to	`“fred” == “sid”`	False
!=	Not equal to	`8 != 8`	False

>	Greater than	`10 > 2`
		`“Fred” > “Bob”`
			True
			True

>=	Greater than or equal to	`5 >= 5`	True

<	Less than	`4 < 34`
		`“Wilma” < “Fred”`
			True
			False

<=	Less than or equal to	`2 <= 109`	True

Logical/Boolean operators

This table shows examples of the Boolean/logical operators.

Operator	Description	Example	Evaluates to

and	Returns True if both conditions are true	`count = 0`
		`index = 44`
		`(count == 0) and (index > 2)`
		`(count > 4) and (index > 2)`
			
			True
			False

or	Returns True if one of the conditions is true	`name = “Fred”`
		`age = 13`
		`(name == “Alan”) or (age > 20)`
		`(name < “Alan”) or (age > 5)`
			
			False
			True

not	Reverses the outcome of the expression; True becomes False, False becomes True	`rain = True`
		`not rain`
			
			False

Brackets

Although Python does not require all arithmetic and logical/Boolean expressions to be fully bracketed, it often improves readability to bracket them. The addition of brackets can often help make understanding the logic and debugging easier.

Programming constructs

(Specification points: 6.2.1, 6.2.2)

Assignment

The assignment operator '=' is used to set or change the value of a variable. The expression on the right is evaluated and the result is stored into the location on the left.

This table shows examples of using assignment.

Example	Description
`count = 0`	Puts the integer value 0 into the variable `count`.
`myName = "Fred"`	Puts the string value 'Fred' into the variable `myName`.
`count = count + 1`	Gets the current value of `count` adds one to it and stores it back into the variable `count`
`check = myName.isalpha ()`	Calls the function <string>.isalpha() using the contents of the variable `myName`. The result is stored into the variable `check`.

Sequence

A sequence is a set of instructions that are executed one after another in order.

This table shows examples of using sequence.

Example	Description

`count = 0`
 	
`myName = "Fred"`
 	
`count = count + 1`
	Firstly, sets the value of the variable `count`.
	
	Secondly, sets the value of the variable `myName`.
	
	Thirdly, increments the value of `count`.

Blocking

Blocking of code segments is indicated by indentation and subprogram calls. These determine the scope and extent of variables they create. Examples of blocking can be seen in the following sections that introduce the other programming constructs.

Selection

In programming, selection is the way to make decisions based on whether a condition evaluates to True or False. Selection is implemented using an ‘if’ statement.

- Where there is no specific action to take if the condition evaluates to False an ‘if’ statement is used on its own.
- Where there is one action to take if the condition evaluates to True and a different action to take if it evaluates to False, an ‘if…else’ is used.
- Where there are multiple conditions to be checked, an ‘if…elif…else’ is used. It is good practice to include the ‘else’ as a catch-all when none of the previous conditions evaluate to True.

Nesting of selection statements is permitted. However, the use of ‘else’ and ‘elif’ is preferable since it makes the logic clearer.

Using two or more separate ‘if’ statements, rather than combining them into one, makes the code less efficient and should be avoided.

This table shows examples of a selection statement and how to format it.

Example	**Description**

`if (count == 1):`
`    print ("In first block")`
	It is possible to use only an ‘if’. Execution will always continue to the line following the print, whether or not the print is executed.

`if (count == 1):`
`    print ("In first block")`
`else:`
`    print ("In second block")`
	When there are only two options, then use an ‘if…else’.

`if (count == 1):`
`    print ("In first block")`
`elif (count == 2):`
`    print ("In second block")`
	In this example, nothing will be printed for any counts above 2 or below 1.

`if (count == 1):`
`    print ("In first block")`
`elif (count == 2):`
`    print ("In second block")`
`else:`
`    print ("In third block")`
	Here, the ‘else’ block will be executed for any numbers other than 1 and 2.

`if (count == 1):`
`    print ("In first block")`
`elif (count == 2):`
`    print ("In second block")`
`elif (count == 3):`
`    print ("In third block")`
`elif (count == 4):`
`    print ("In fourth block")`
	This statement only deals with the numbers from 1 to 4, inclusive. Notice that in this case the ‘else’ is not required.

Ordering of test conditions

In some cases, the order of the test conditions is important. This occurs when making decisions based on ranges, such as grades or ranks. For grades, the tests should start at the top and use greater than, or start at the bottom and use less than. Otherwise, a grade could fall into the wrong range.

In this example, if the tests for ‘Bronze’ and ‘Silver’ were reversed, then a score of 55, which should be ‘Bronze’ would be awarded ‘Silver’.

Example	Output

`score = 55`
`if (score < 40):`
`    print ("No award")`
`elif (score < 60):`
`    print ("Bronze")`
`elif (score < 80):`
`    print ("Silver")`
`else:`
`    print ("Gold")`
	`Bronze`

`score = 55`
`if (score < 40):`
`    print ("No award")`
`elif (score < 80):`
`    print ("Silver")`
`elif (score < 60):`
`    print ("Bronze")`
`else:`
`    print ("Gold")`
	`Silver`

Repetition and iteration

In programming, a loop is used when a sequence of instructions needs to be repeated.

The terms repetition and iteration are often used interchangeably to describe the action of loops. However, differentiating between them can benefit students.

This table how repetition and iteration differ.

Term	Definition
Repetition	To go around a loop executing the same sequence of instructions whilst a condition is True.
Iteration	To go around a loop executing the same sequence of instructions until all the items in a data structure have been processed.

Repetition

Repetitive loops keep repeating as long as a condition remains True.

This table shows examples of using repetition.

Example	**Description**

`while (count > 0):`
`    print ("Count is", count)`
`    count = count - 1`
	The number of times through this loop depends on the initial value of the variable `count`.

`key = input ("Enter Y or N")`
`while (key != "N"):`
`    print ("Going around again")`
`    key = input ("Enter Y or N")`
	The user is in control of this loop. The loop will keep executing until the user types in a ‘N’ when asked to enter ‘Y’ or ‘N’. Notice that any value besides ‘N’ is interpreted as ‘Y’.

`key = “X”`
`while (key != “N”):`
`    key = input (“Enter Y or N”)`
`    if (key != “N”):`
`        print (“Going again”)`
`    else:`
`        print (“Working”)`
	This example shows nesting a selection (‘if…else’) inside the repetition (‘while’).

Iteration

Iterative loops keep repeating until all the items in a data structure have been processed.

The ‘for…in’ construct allows every item in a one-dimensional data structure to be processed as it is encountered.

Nested ‘for… in’ loops are used to iterate through two-dimensional data structures. The outer loop iterates through the records and the inner through the fields in each record.

Unfortunately, the terms ‘iteration’ and ‘repetition’ are not synonymous with the Python language constructs for looping.

This table shows examples of using iteration.

Example	Description

`numbers = [10, 20, 30, 40, 50]`
`for num in numbers:`
`    print (num * 2)`
	This example outputs each number in the array `numbers`, multiplied by 2. Each output is on a separate line.

`theTable =[`
`           [152,"Jones",78.45],`
`           [938,"Black",24.12],`
`           [454,"Green",32.00]]`
 	
`for student in theTable:`
`    print ("Name:",`
`           student[1],`
`           "Balance:",`
`           student[2])`
	This 'for' loop processes every student in `theTable`. In each pass of the loop, the variable `student` holds a record, which is a one-dimensional data structure. Individual fields in `student` are accessed using indexing.
	
	This is the output from this loop:
	
	`Name: Jones Balance: 78.45`
	`Name: Black Balance: 24.12`
	`Name: Green Balance: 32.0`

`for ndx in range (0, len (theTable)):`
`    print ("ID:", theTable[ndx][0])`
	This example uses two indices to access a field in each record of `theTable`.
	
	This is the output from this loop:
	
	`ID: 152`
	`ID: 938`
	`ID: 454`

`weeklySales = [[120.00, 211.09, 99.00, 58.12, 119.45, 167.34, 308.01], [78.24, 165.59, 101.12, 96.42, 106.05, 178.24, 297.15], [132.70, 203.19, 123.57, 86.90, 138.11, 177.91, 402.64]]`
 	
`totalSales = 0.0`
 	
`for week in weeklySales:`
`    for day in week:`
`        totalSales = totalSales + day`
`print (totalSales)`
	This example uses nested loops to process all the fields in each record of `weeklySales`.

Count-controlled and condition-controlled loops

The terms associated with the looping constructs and how they are used to describe actions in a program are sometimes confusing.

This table distinguishes between count-controlled and condition-controlled loops.

Term	Definition
Count-controlled loop	The number of passes around the loop is already known or can be determined before the loop starts.
Condition-controlled loop	The number of passes around the loop is not known before the loop starts. The loop keeps going around as long as a condition is True.

Condition-controlled loops are particularly useful when a programmer does not know in advance how many times a loop must go around.

This table shows examples of each of the Python looping constructs specified in the Programming Language Subset, and how each can be categorised using the terms defined above.

Python example	Is a type of	Is an example of

`choice = "Y"`
`while (choice != "N"):`
`    choice = input ("You choose: ")`
	Repetition	Condition-controlled

`import random`
`count = 0`
`while (count < 5):`
`    count = random.randint (0, 7)`
`    print (count)`
	Repetition	Condition-controlled

`count = 0`
`while (count < 5):`
`    count = count + 1`
`    print (count)`
	Repetition	Count-controlled

`for num in range (0, 5):`
`    print (num)`
	Repetition	Count-controlled

`table = [4, 9, 2, 3, 7]`
`index = 0`
`while (index < len (table)):`
`    print (table[index])`
`    index = index + 1`
	Iteration	Count-controlled

`for num in range (5):`
`    print (num * 2)`
	Repetition	Count-controlled

`table = ["cat", "dog", "fox"]`
`for word in table:`
`    print (word)`
	Iteration	Count-controlled

`table = ["cat", "dog", "fox"]`
`for index in range (0, len (table)):`
`    print (table[index])`
	Iteration	Count-controlled

`table = [1, 2 ,3, 4, 5, 6]`
`for index in range (len (table) - 1, -1, -2):`
`    print (table[index])`
	Iteration	Count-controlled

Python example	Is a type of	Is an example of

`import random`
`table = [1, 2 ,3, 4, 5, 6]`
`stop = random.randint (1, len (table))`
`for index in range (0, stop):`
`    print (table[index])`
	Iteration	Count-controlled

From these examples, it is possible to say that when the Python 'range()' function is used to generate a sequence of numbers, the loop is count-controlled. When that sequence is used to access a data structure it is iteration. When the sequence is used to count the times around the loop, it is repetition. This is a distinction of convenience, as from another perspective, the sequence generated by 'range()' is iterated over.

Students should not worry about the subtleties of the distinctions between these terms. If the word 'iteration' is stated in an instruction, then students can use a 'for' loop to process a data structure. If the word 'repetition' is stated in an instruction, then students may choose whichever construct ('while', 'for') is suitable for the problem.

Subprograms

(Specification points: 6.2.1, 6.6.1, 6.6.2, 6.6.3)

A subprogram is a distinct, named block of code, incorporating its own scope, that performs a specific task, and is called into action from other blocks of code.

The use of subprograms is good programming practice because it breaks program code into smaller sections, making it easier to read, understand and debug.

Procedures and functions

Subprograms can be either procedures or functions. Although Python does not differentiate between the two, using the key word 'def' to define both, there is a conceptual difference between them.

This table sets out the difference between a procedure and a function.

Term	Definition
Procedure	Procedures may or may not take parameters. They do not return a value to the calling block.
Function	Functions may or may not take parameters. They always return a value to the calling block and are called as part of an expression so that the return value is assigned to a variable.

Parameters and arguments

In the subprogram definition line, the variables inside the brackets are called parameters. When the subprogram is called in another block of code, the variables passed in are referred to as arguments.

A useful technique is to name the parameters in a way that identifies them as belonging to the subprogram. This reduces the chance of confusing them with variables in other parts of the program. One easy way of doing this is to begin parameter names with the letter 'p'.

Here is an example of a subprogram definition that takes a parameter beginning with the letter 'p'.

```
def processMenuChoice (pChoice):
```

Students must take care to list the arguments in a call to a subprogram in the same order as the parameters in its definition.

Using subprograms

Python comes with a library of built-in subprograms that perform common functions. (See the Built-in subprograms section), but also allows programmers to write their own user-devised subprograms and import libraries of subprograms (See the Library modules section).

This table shows examples of using user-devised subprograms.

Example	Description

`studentTable =[`
`           [152,"Jones",78.45],`
`           [938,"Black",24.12],`
`           [454,"Green",32.00]]`
 	
`def displayStudents ():`
`    for student in studentTable:`
`        print (student[2], student[1])`
	This example is a procedure because it does not return a result. It also takes no parameters inside the brackets on the definition line.
	
	The output from calling this procedure is:
	
	`78.45 Jones`
	`24.12 Black`
	`32.0 Green`

`def displayOneStudent (pIndex):`
`    print (studentTable[pIndex][2],`
`           studentTable[pIndex][1])`
	This example is a procedure because it does not return a result. However, it does take a single parameter. The parameter is used to index a data structure.
	
	The output from calling this procedure, with an argument of 2, is:
	
	`32.0 Green`

`def roll ():`
`    showing = random.randint (1, 6)`
`    return (showing)`
	This example is a function because it does return a value, in its last line.
	
	An example of a returned value from calling this function is:
	
	`3`

Example	**Description**

`def raisePower (pPower):`
`    value = 2 ** pPower    # Calc 2^pPower`
`    return (value)`
	This example is a function as it returns a value, in its last line. It also takes a single parameter, inside the brackets on the definition line. The parameter is used in a calculation.
	
	The returned value from calling this function, with an argument of 3 is:
	
	`8`

`displayStudents ()`	This is an example of how to call a procedure with no arguments. The output is listed above.

`myStudent = 2`
`displayOneStudent (myStudent)`
	This is an example of a call to a procedure that takes an argument. The argument, in this case, is an integer variable.

`print (roll ())`	This is an example of a call to a function that takes no arguments. The returned value from the function is immediately passed into the 'print()' function.

`thePower = raisePower (3)`
`print (thePower)`
	This is an example of a call to a function that takes a single argument.

Global and local variables

The scope of a variable refers to the part of a program in which it exists and can be used. Global variables are defined at the level of the main program and are accessible from anywhere in the program.

Local variables are defined inside subprograms. They are only accessible within the subprogram in which they are defined and cease to exist once that subprogram finishes executing. A local variable with the same name as a variable declared in the main program or in a different subprogram is a different variable.

It is good programming practice to minimise the use of global variables and include only those that are needed at the main program level. Having lots of global variables can lead to conflicts with naming and make debugging much more difficult.

Variables that are only used to do work inside a subprogram should be defined as local variables inside that subprogram. Parameters, the placeholders on the definition line for values passed into a subprogram, are automatically local variables inside that subprogram.

Code example

This example minimises the number of global variables. All variables used by the subprograms are defined inside the subprogram. It also illustrates how to pass a global variable into a subprogram as an argument.

```python
# ------------------------------------------------------------
# Global variables
# ------------------------------------------------------------
# A global data structure
studentTable = [["Student A", 11],
                 ["Student B", 22],
                 ["Student C", 33]]
userName = ""                    # Only used in main program

# ------------------------------------------------------------
# Subprograms
# ------------------------------------------------------------
def displayWelcome (pName):
    # pName is a parameter so is a local variable
    theMessage = "Welcome "      # Local variable

    theMessage = theMessage + pName
    print (theMessage)

def displayStudents ():
    print ("Using a global variable")
    for student in studentTable:
        print (student)

def displayTable (pTable):
    print ("Using a parameter")
    for item in pTable:
        print (item)
# ------------------------------------------------------------
# Main program
# ------------------------------------------------------------
userName = input ("What is your name? ")

# Global variable passed into a subprogram as an argument,
#    which takes the place of the parameters
displayWelcome (userName)

displayStudents ()       # No arguments

# Global data structure passed into a subprogram as an
#    argument, which takes the place of the parameters
displayTable (studentTable)
```

A variable or data structure held in global scope can be accessed and changed from inside a subprogram.

There is no strict rule about what can be global and what should be passed into a subprogram as an argument. However, it is good programming practice is to avoid accessing global variables directly from inside subprograms and – instead – to pass them into the subprogram as arguments.

Global keyword

The Python keyword ‘global’ is used to create global variables from within subprograms. This allows code, outside a subprogram, to modify a variable created inside the subprogram. This is not good programming practice as it can cause naming conflicts and make debugging challenging. Students are advised not to use this facility.

Inputs and outputs

(Specification points: 6.4.1, 6.4.2)

Screen output

The ‘print()’ function is used to display output on the screen.

This table shows examples of using screen output.

Example	Description
`print (“Hello world”)`	The output on the screen is: `Hello world`

`myScore = 83`
`print (myScore)`
	The output on the screen is:
	`83`

`print (“My score is”, myScore)`	The output on the screen is: `My score is 83`

Output can be formatted to suit the problem requirements and the user's needs using string formatting with positional placeholders and format requirements shown in '{}'. This is particularly useful for presenting columnar and tabular information.

Here are two examples.

Example	**Description**

`layout = "{:<6} {:^4.3f}"`
`print (layout.format ("Hello", 3.14159))`
	This example uses placeholders to control the spacing of output displayed to the screen.
	
	There is another example in the PLS.
	
	The output is:
	`Hello  3.142`

`print ("{:.2f}".format (price))`	This example uses the string formatting operator to display a currency value to two decimal places.

The use of formatting strings using literal string interpolation (f-strings) is not set out in the PLS. However, some students may want to learn this approach to formatting strings.

Commas versus concatenation

Commas delimit the parameter values passed to the print() function, which adds a space between each parameter value on output. The use of commas to join strings produces a tuple of strings, rather than a single string.

Concatenation joins the values together into one string without spaces, before being passed to the print() function. Python can only concatenate string objects. If the programmer wants to merge a string with a non-string (an integer or a float), they must use the function 'str()' to convert the non-string to a string.

Non-printable control characters

Non-printable control characters are used to specify how output appears on the screen and in a text file.

The new line character '\n' moves the cursor down to the beginning of the next line of text. Each line of a text file is terminated with the '\n'.

The tab character '\t' moves the cursor one tab space to the right.

Keyboard input

The 'input()' function is used to take input from the user via the keyboard. Remember, that all input from the keyboard is in strings, so may need to be converted to the data type required by the program.

This table shows examples of using keyboard input.

Example	Description
`theName = input ("Enter your name: ")`	This example accepts a string input and stores it in the variable `theName`.
`theGuess = int (input ("Guess: ")`	This example accepts a string input, converts it to an integer, then stores it in the variable `theGuess`.
`height = float (input ("Metres: ")`	This example accepts a string input, converts it to a real (decimal) number, then stores it in the variable `height`.

Files

All data stored on disk for this qualification will be stored as comma separated value text files. No other file formats need to be considered.

File operations include open, close, read, write, and append. It is recommended that files be opened for reading or writing, not both at the same time. Although it is possible to read from and write to the same file, it is beyond the scope of the specification.

Students need to be aware of how the write and append operations differ. The former overwrites any existing content in a file; the latter adds to the end of any existing content.

It is good programming practice to always close files before a program finishes. Sometimes, files that are left open can be corrupted.

This table shows examples of using files.

Example	Description
`theFile = open ("Students.txt", "r")`	This example opens a file for reading only.

`for line in theFile:`
`    <process the line>`
	This example shows how to read each line from the file, using the variable returned from the 'open()' function. The commands indented under the 'for…in' loop will execute for each line.

Example	**Description**
`theFile.close()`	This example closes a file, using the variable returned from the ‘open()’ function.
`outFile = open ("Students.txt", "w")`	This example opens a file for writing only. When a file is opened for writing, all of its existing content is destroyed.
`for student in studentTable:` `<build an output string>` `outFile.write (<output string>)`	This example processes each record in a data structure, by creating a string from all the fields, and then writes the resulting string to the file, using the variable returned from the ‘open()’ function.
`outFile.close()`	This example closes a file, using the variable returned from the ‘open()’ function.

Note that reading and writing files using Python’s ‘with open (…) as …’ does not require an explicit ‘<file>.close()’ function call.

Reading from a file

The general approach for reading data from a file is to

- open the file for reading
- read each line in the file
    - process the line by removing the line feed and converting field types
    - build a record from the individual fields
    - append the new record to a two-dimensional data structure
- close the file.

This approach will result in a two-dimensional table of records, i.e. a nested list of lists in Python. The internal data structure can then be processed in any way required.

Writing to a file

The general approach for writing data to a file is to

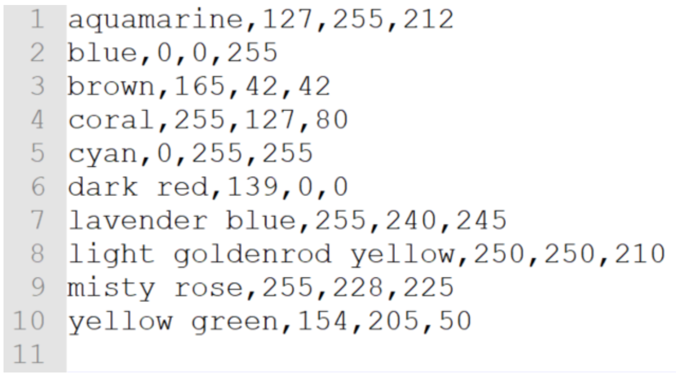
- open the file for writing
- process each record in the two-dimensional data structure
    - convert each field to a string
    - join the field strings with commas, without any white space before or after them, except the last field, to create an output line
    - add a line feed to the output line
    - write the output line to the file
- close the file.

This approach will result in a file where each record in the original data structure is on a single line, with each field separated by a comma. The content of the file is viewable in any text editor.

Code example

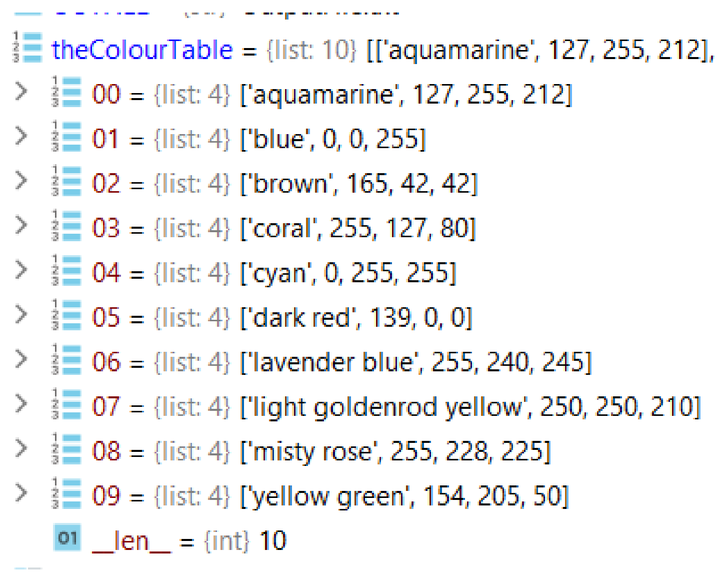
This example shows one way that beginner programmers can work with files.

The original file has this content:



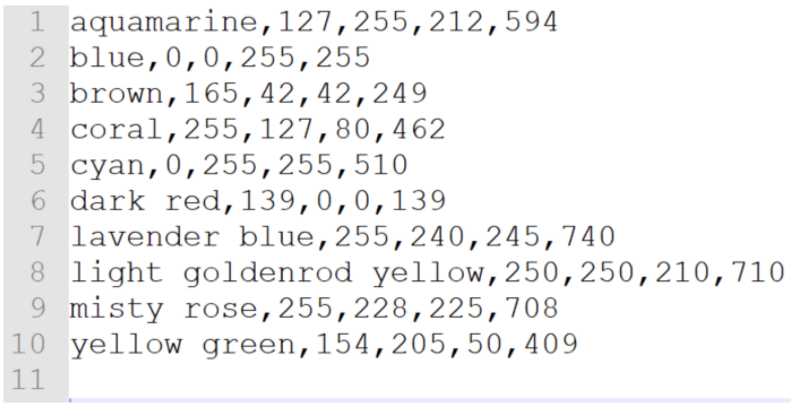
```
aquamarine,127,255,212
blue,0,0,255
brown,165,42,42
coral,255,127,80
cyan,0,255,255
dark red,139,0,0
lavender blue,255,240,245
light goldenrod yellow,250,250,210
misty rose,255,228,225
yellow green,154,205,50

```

Once the file content is read into the program, the content of the memory holding the two-dimensional data structure looks like this:



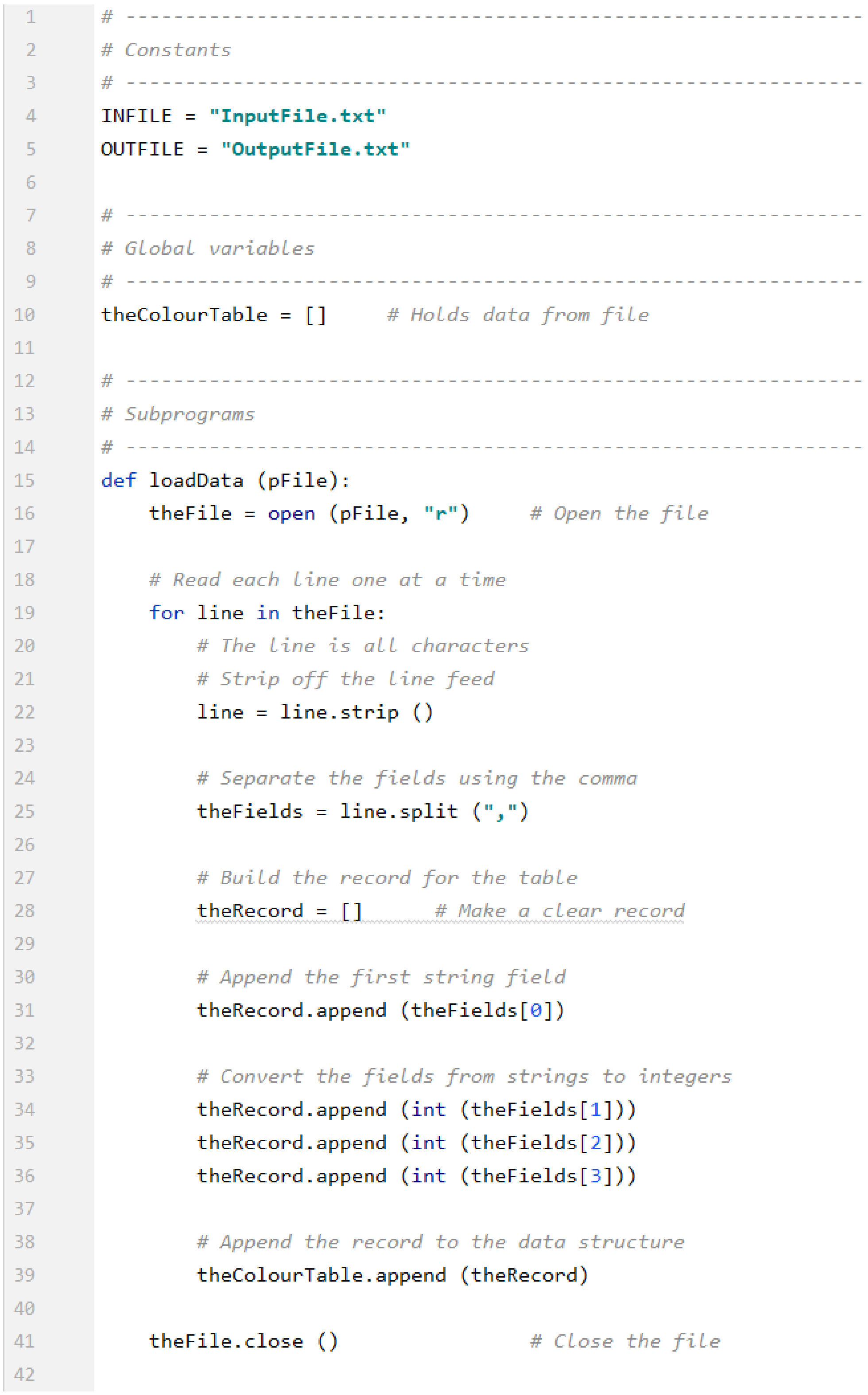
```
theColourTable = {list: 10} [['aquamarine', 127, 255, 212],
>   00 = {list: 4} ['aquamarine', 127, 255, 212]
>   01 = {list: 4} ['blue', 0, 0, 255]
>   02 = {list: 4} ['brown', 165, 42, 42]
>   03 = {list: 4} ['coral', 255, 127, 80]
>   04 = {list: 4} ['cyan', 0, 255, 255]
>   05 = {list: 4} ['dark red', 139, 0, 0]
>   06 = {list: 4} ['lavender blue', 255, 240, 245]
>   07 = {list: 4} ['light goldenrod yellow', 250, 250, 210]
>   08 = {list: 4} ['misty rose', 255, 228, 225]
>   09 = {list: 4} ['yellow green', 154, 205, 50]
    __len__ = {int} 10
```

After a new field is added to each record in the data structure, the content of the output file looks like this:



```
aquamarine,127,255,212,594
blue,0,0,255,255
brown,165,42,42,249
coral,255,127,80,462
cyan,0,255,255,510
dark red,139,0,0,139
lavender blue,255,240,245,740
light goldenrod yellow,250,250,210,710
misty rose,255,228,225,708
yellow green,154,205,50,409

```

The full program is shown in these two images.



```python
# -------------------------------------------------------------
# Constants
# -------------------------------------------------------------
INFILE = "InputFile.txt"
OUTFILE = "OutputFile.txt"

# -------------------------------------------------------------
# Global variables
# -------------------------------------------------------------
theColourTable = []     # Holds data from file

# -------------------------------------------------------------
# Subprograms
# -------------------------------------------------------------
def loadData (pFile):
    theFile = open (pFile, "r")     # Open the file

    # Read each line one at a time
    for line in theFile:
        # The line is all characters
        # Strip off the line feed
        line = line.strip ()

        # Separate the fields using the comma
        theFields = line.split (",")

        # Build the record for the table
        theRecord = []        # Make a clear record

        # Append the first string field
        theRecord.append (theFields[0])

        # Convert the fields from strings to integers
        theRecord.append (int (theFields[1]))
        theRecord.append (int (theFields[2]))
        theRecord.append (int (theFields[3]))

        # Append the record to the data structure
        theColourTable.append (theRecord)

    theFile.close ()                # Close the file

```

```
def addColumn (pTable):
    total = 0

    for item in pTable:
        total = 0
        total = total + item[1] + item[2] + item[3]
        item.append (total)

def saveData (pFile):
    outLine = ""            # To write to the file
    theFile = open (pFile, "w")

    for colour in theColourTable:
        outLine = colour[0]     # String field
        # Add the comma separator
        outLine = outLine + ","

        # Any field not a string must be converted to string
        outLine = outLine + str (colour[1]) + ","
        outLine = outLine + str(colour[2]) + ","
        outLine = outLine + str (colour[3]) + ","
        # No comma on the last field
        outLine = outLine + str(colour[4])

        # Add the line feed character
        outLine = outLine + "\n"

        # Write the line to the file
        theFile.write (outLine)

    theFile.close ()        # Close the file

# -------------------------------------------------------------
# Main program
# -------------------------------------------------------------
loadData (INFILE)       # Load the data from a file

# Process the 2D data structure in some way
addColumn (theColourTable)

saveData (OUTFILE)      # Save the data to a file

print ("Goodbye")
```

Supported subprograms

(Specification point: 6.6.1)

Built-in subprograms

A built-in subprogram is a procedure or function that is always available and does not need to be written or imported before it can be used.

This table shows examples of using the built-in subprograms.

Example	Description

`character = chr (65)`
`print (character)`
	This example returns the string which matches the Unicode value of 65.
	
	The output is:
	`A`

`firstNumber = 7`
`secondNumber = 21`
`print(bool(firstNumber ==`
`secondNumber))`
	This example returns False as `firstNumber` is not equal to `secondNumber`.

`name = “Manjit”`
`print(bool(name))`
	This example returns True as `name` is a non-empty string.

`newList = []`
`print(bool(newList))`
	This example returns False as `newList` is an empty list.

`decimalNumber = float (input (`
`print (decimalNumber)`
	This example converts the string input `decimalNumber` into a real.

`name = input ("Name: ")`
`print (name)`
	This example displays the prompt to the screen and waits for the user to type in characters followed by a new line.
	
	The console session looks like this:
	`Name: `*`Shaun`*
	`Shaun`

`integerNumber = int (input(`
`print (integerNumber)`
	This example converts the string input into an integer.

`word = "Garage"`
`length = len (word)`
`print (length)`
	This example returns the length of the word ‘Garage’.
	
	The output is:
	`6`

`string = ord ("K")`
`print (string)`
	This example returns the integer equivalent to the Unicode single character string ‘K’.
	
	The output is:
	`75`

`print (“Hello”)`	This example prints ‘Hello’ to the display. The output is: `Hello`

Example	**Description**

`for num in range (2):`
`    print (num)`
	This example prints the numbers from 0 up to, but not including 2, incrementing 1 each time.
	
	The output is:
	`0`
	`1`

`for num in range (10, 13):`
`    print (num)`
	This example prints the numbers from 10 up to, but not including 13, incrementing 1 each time.
	
	The output is:
	`10`
	`11`
	`12`

`for num in range (10, 0, -2):`
`    print (num)`
	This example prints the numbers from 10 down to, but not including 0, decreasing by 2 each time.
	
	The output is:
	`10`
	`8`
	`6`
	`4`
	`2`

`total = 62.7259`
`cost = round (total, 2)`
`print (cost)`
	This example rounds the variable ‘total’ to two decimal places and stores the result in the variable ‘cost’.
	
	The output is:
	`62.73`

`number = 66`
`stringNumber = str (number)`
`print (stringNumber + " is stored as a string.")`
	Python can only concatenate string objects. So, if the programmer wants to merge a string with a non-string (an integer or a float), they must use the function ‘str()’ to convert the non-string to a string.

List subprograms

This table shows examples of using the list subprograms.

Example	Description

`table = [1, 2, 3, 4]`
`table.append (5)`
`print (table)`
	This example appends an item to an existing list.
	
	The output is:
	`[1, 2, 3, 4, 5]`

`table = [10, 9, 8, 7, 6]`
`del table[1]`
`print (table)`
	This example deletes the item at index 1.
	
	The output is:
	`[10, 8, 7, 6]`

`table = [1, 2, 4, 5]`
`table.insert (2, 3)`
`print (table)`
	This example inserts a new item, the number 3, at index position 2.
	
	The output is:
	`[1, 2, 3, 4, 5]`

`table_one = list ()`
`print (table_one)`
 	
`table_two = []`
`print (table_two)`
	This example shows two ways to create an empty list. Once created the append subprogram can be used to put items into the list.
	
	The output is:
	`[]`
	`[]`

String subprograms

This table shows examples of using the string subprograms.

Example	**Description**

`str_one = "hello"`
`length = len (str_one)`
`print (length)`
	This example finds the length of a string.
	
	The output is:
	`5`

`str_one = "hello"`
`where = str_one.find ("ell")`
`print (where)`
	This example finds the start of the substring, inside the original string. It will return -1, if the substring is not there.
	
	The output is:
	`1`

`str_one = "hello"`
`where = str_one.index ("o")`
`print (where)`
	This example finds the starting index of the substring, inside the original string. It will report an error if the substring is not there.
	
	The output is:
	`4`

`status = str_one.isalpha ()`
`print (status)`
	This example checks to see if the string is all alphabetic characters.
	
	The output is:
	`True`

`str_two = "123XYZ"`
`status = str_two.isalnum ()`
`print (status)`
	This example checks to see if the string is all alphabetic and numeric characters.
	
	The output is:
	`True`

`str_three = "789"`
`status = str_three.isdigit ()`
`print (status)`
	This example check to see if the string is all digits.
	
	The output is:
	`True`

`str_one = "Hello world"`
`str_two = "Sun"`
`str_three = str_one.replace (`
`              "world", str_two)`
`print (str_three)`
	This example replaces a substring in the original string, with a new substring.
	
	The output is:
	`Hello Sun`

`str_one = "cat,dog,fox"`
`str_list = str_one.split (",")`
`print (str_list)`
	This example splits a string into a list, based on the character supplied, in this case a ‘,’.
	
	The output is:
	`['cat', 'dog', 'fox']`

`str_one = "*ABC**"`
`str_two = str_one.strip ("*")`
`print (str_two)`
	This example removes all occurrences of a character from the front and back of the original string.
	
	The output is:
	`ABC`

Example	Description

`str_one = "abc"`
`str_two = str_one.upper ()`
`print (str_two)`
	This example converts the original string to all uppercase.
	
	The output is:
	`ABC`

`str_one = "XYZ"`
`str_two = str_one.lower ()`
`print (str_two)`
	This example converts the original string to all lowercase.
	
	The output is:
	`xyz`

`str_one = "ABCxyz"`
`status = str_one.isupper ()`
`print (status)`
	This example checks to see if a string is all uppercase characters.
	
	The output is:
	`False`

`str_one = "abcxyz"`
`status = str_one.islower ()`
`print (status)`
	This example checks to see if a string is all lowercase characters.
	
	The output is:
	`True`

Library modules

The PLS specifies a limited set of library modules and an even smaller set of actual subprograms found in them that students are expected to be familiar with.

This table shows how to import a library module.

Statement	Example	Description
import	`import turtle`	This example imports the turtle graphics library module.
import	`from math import pi`	This example imports the constant Pi from the math library.

When referring to an item from an imported library, the name of the library must be included, i.e. <libraryName>.<itemName>.

Random library module

This table shows examples of using the random library module subprograms.

Example	Description

`import random`
`num = random.randint (1, 10)`
`print (num)`
	This example shows how to generate a random whole number between two bounds, inclusive.
	
	The output is:
	`8`

`import random`
`decimal = random.random ()`
`print (decimal)`
	This example shows how to generate a random decimal number, between 0 and 1.
	
	The output is:
	`0.9754469153477409`

Math library module

This table shows examples of using the math library module subprograms and constant.

Example	Description

`import math`
`num = 8.752`
`result = math.ceil (num)`
`print (result)`
	This example shows that the 'math.ceil()' subprogram returns the nearest integer not less than the original.
	
	The output is:
	`9`

`import math`
`num = 8.752`
`result = math.floor (num)`
`print (result)`
	The 'math.floor()' subprogram returns nearest integer not greater than the original.
	
	The output is:
	`8`

`import math`
`num = 25`
`result = math.sqrt (num)`
`print (result)`
	This is an example of the square root subprogram.
	
	The output is:
	`5.0`

`import math`
`print (math.pi)`
	Pi is a constant, not a subprogram.
	
	The pi value is given to fifteen decimal places.
	The output is:
	`3.141592653589793`
	
	When performing a mathematical calculation using pi, it's best practice to use the pi value given by the math module rather than hard coding the value.

Time library module

The 'time.sleep()' function suspends program execution for a given number of seconds, after which executions resumes at the next line of the program.

This table shows an example of using the time library module subprogram.

Example	Description

`import time`
`print ("Sleeping ...")`
`time.sleep (5)`
`print ("Awake")`
	This example prints the sleeping message, waits 5 seconds, then prints the awake message.
	
	The output is:
	`Sleeping ...`
	`Awake`

Turtle graphics library module

Examples of all the turtle graphics library module subprograms can be found online in the online Python documentation. In addition, there are many online tutorials that will introduce the functionality of turtle graphics.

Console Session

A console session is the window or command line where the user interacts with a program. It is the default window that displays the output from `print ()` and echoes the keys typed from the keyboard. It will appear differently in different development tools.

Functionalities not in the PLS

Although the Programming Language Subset is based on an existing high-level programming language (Python 3), students do not need to understand or be able to use any features not expressly set out in the PLS document. Some commonly used features of Python are not included in the PLS. Therefore, students are encouraged to design and implement code without them.

Flow control statements

Although some languages use flow control statements such as goto, break, continue, pass, or exit, these are not supported in the PLS. Using these statements can make code difficult to read and significantly more difficult to debug. Introducing one of these statements, which may appear to fix one bug, can introduce incorrect behaviours in other areas. If beginner programmers believe they need to use them, then they are advised to reconsider the design of their repetition, iteration, or selection blocks.

Break

Consider this very simple program that uses an infinite loop and the break statement.

```
userNum = 0
# Using an infinite loop and a break statement
while (True):
    userNum = int (input ("Enter a number: "))
    if (userNum == 0):
        break
    else:
        print ("The number is: ", userNum)
```

It can be rewritten, providing the same logic, much more clearly. In the revised version, the test at the top of the loop clearly identifies under what condition the loop should terminate.

```
userNum = 0
# Using a condition-controlled loop
userNum = int (input ("Enter a number: "))
while (userNum != 0):
    print ("The number is: ", userNum)
    userNum = int (input ("Enter a number: "))
```

Another advantage of restricting the use of the break statement is transferability to other programming languages. Using the later logic works in most programming languages. Therefore, no additional learning of constructs is required to use another language to implement the original logic.

Exit

Consider this very simple program that uses an exit statement.

```
24  # Logic using exit statement
25  userNum = int (input ("Enter a number"))
26  while (userNum >= 0) and (userNum <= 10):
27      if (userNum > 5):
28          print ("Upper Half")
29      elif (userNum == 0):
30          exit()
31      else:
32          print ("Lower Half")
33      userNum = int (input ("Enter a number"))
34  print ("Bye")
```

The user types in numbers with 0 indicating termination. Using the exit statement means the code on line 34 is never executed. In this case, it is only a print statement, but it could be code that should have written data to a file.

Programs should have only a single termination point and that should be when the last instruction in the sequence of instructions is executed. Usually, this means the last instruction in the file.

This is a revision of the logic that behaves in a more predictable way.

```
37  # Revision of logic without exit statement
38  userNum = int (input ("Enter a number"))
39  while (userNum > 0) and (userNum <= 10):
40      if (userNum > 5):
41          print ("Upper Half")
42      else:
43          print ("Lower Half")
44      userNum = int (input ("Enter a number"))
45  print ("Bye")
```

Exception handling

The use of try/except for exception handling is not set out in the PLS. However, some students may want to learn this approach to handling some types of errors.