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

Contents

Introduction

The Edexcel Programming Language Subset (PLS) is based on a restricted set of functionalities available in the programming language named Python 3.

All problems set during the practical exercises and on the examinations can be solved with only the functionalities presented in the PLS.

This means that more class time can be used for the teaching of computational thinking and problem solving, rather than the syntax and libraries provided by a programming language.

The supported functionalities are those that transfer easily to all programming languages, thereby allowing learners to move to other languages without the additional challenge of learning different constructs.

Teachers and students should practice solving problems using this restricted set of functionalities. There is no need to go beyond these. Full marks in the exams can be achieved by using this small set of functionalities well.

Notes

The pair of < > symbols indicates where expressions or values need to be supplied. The pair of < > symbols are not part of the PLS.

Additional spaces and brackets have been used in some lines of code. This is entirely to aid readability. Blank space, including inside lines of code, is ignored by the language translator, unless it is part of the indentation. Additional brackets, as long as they do not change order of execution, may be useful and aid readability. For example, there is no real difference between the lines shown in each row.

`print ("Hello World")`	`print("Hello World")`
`choice = int (input ("Choose a number"))`	`choice=int(input("Choose a number"))`
`if (num < 12) or (num > 100):`	`if num < 12 or num > 100:`

Individual programmers develop their own style. For example, some use underscores in identifiers. It is not essential to follow the suggestions in this document strictly. So long as students are consistent in use of their own preferred style, the reader should be able to understand the code.

This document should be read in conjunction with the 'Programming Language Subset (PLS)' document provided by Pearson Edexcel.

Comments

Anything on a line after the character # is considered a comment.

Line continuation

Sometimes it is helpful to control the width of a line of code. This helps readability. One way to do this is by using '\' (backslash) character. The interpreter will join the separate lines into one, when translating the code.

Both code blocks below will behave in the same way when run.

```
if (struct[j] == struct[j+1]) and \
    (struct[j] == struct[j+2]):
    print ("Game over")

if (struct[j] == struct[j+1]) and (struct[j] == struct[j+2]):
    print ("Game over")
```

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, preceding each group with a comment indicating what it holds. Here is one layout that works with the PLS.

```
1  # ------------------------------------------------------------------
2  # Import libraries
3  # ------------------------------------------------------------------
4
5  # ------------------------------------------------------------------
6  # Constants
7  # ------------------------------------------------------------------
8
9  # ------------------------------------------------------------------
10 # Global variables
11 # ------------------------------------------------------------------
12
13 # ------------------------------------------------------------------
14 # Subprograms
15 # ------------------------------------------------------------------
16
17 # ------------------------------------------------------------------
18 # Main program
19 # ------------------------------------------------------------------
20
```

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 can help students avoid unintended use of nested subprogram definitions and excessive numbers of global variables.

Identifiers

In the PLS, camel case is used to name identifiers, such as variables, constants, and subprograms. Identifiers will be started with a lowercase letter. Each change of word or abbreviation will begin with an uppercase letter.

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

count	`bigList`	`aCounter`	`countAnimalTypes`	`isValidCounter()`
name	`primaryKey`	`lastName`	`studentFirstName`	`findStudentName()`
key	`discountCode`	`blue`	`dbForeignKey`	`showKeys()`

Data types and coercion

Primitive data types

The PLS is typed. It is important for students to understand the difference between a declaration and initialisation. Declarations allocate 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 done in Python, may be implicitly typed via the data type of the first value assigned to them (see the section 'Combining declaration and initialisation').

Variable declaration

Data type	Explanation	PLS	Example
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 one byte in length	str	`myInitial = str()`

Combining declaration and initialisation

The PLS supports implicit data typing by associating an implied data type with the variable when it is assigned its initial value. Good programming practice indicates that once a variable has been associated with a defined data type, its type should not be changed to a different data type.

The table shows examples of implicit data typing, achieved by assigning a 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
`myName = "Joseph"`	A string variable initialised to "Joseph"

Constants

Constants are identifiers that are set only once in the lifetime of the program. Constants are conventionally named in all uppercase characters. This identifier name is then used to represent the value throughout the program code.

The use of constants aids readability and maintainability. The value only needs changing at a single place in the whole program code.

The table shows examples of declaring and initialising constants.

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

Coercion

Coercion is used to transform the type of a variable before processing it. It does not change the type of the original data.

Examples of coercion are shown here. This list is not exhaustive. Coercion between any types is allowed.

Example	Description
`count = int (input ("Enter a number"))`	Keyboard input is always in characters. This coerces the character into an integer. The variable 'count' is an integer.
`balance = float (count)`	The variable 'balance' is float and the variable 'count' is integer. The contents of 'count' is coerced to float before storing in 'balance'. The original value of 'count' is unchanged.

Structured data types

A structured data type is a sequence of items, which themselves are typed. Sequences start with an index of zero.

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]`

Note, that the data type 'array' is not the same as the data type 'list'. In the PLS, they are both created using a list. 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 like the record of a database where fields are of different types.

Dimensions

The number of dimensions supported by the PLS is two. Any structured data type will either be one dimensional or two dimensional.

Data type	Explanation	PLS Equivalent	Example
Two-dimensional array	**A sequence of the same type item, with a fixed length. In this case, each individual item is itself an array.**	[[ ], [ ], … [ ]]	`sensorReadings = [[80, 75, 90], [15, 25, 35], [82, 72, 62]]`
Two-dimensional structure of records	**A sequence of items, usually of mixed data types, with a fixed number of fields.**	[[ ], [ ], … [ ]]	`recordTable = [[1524, "Jones", "Rebecca", True, 78.45], [5821, "Lawson", "Martin", False, 23.98]]`

Note: The PLS does not support ragged data structures. Therefore, all records in a two-dimensional structure will have the same number of elements.

Operators

Arithmetic operators

Operator	Operation
/	division
*	multiplication
**	exponentiation
+	addition
-	subtraction
//	integer division (integer part of result)
%	modulus (remainder after division)

Relational operators

Operator	Operator meaning	Sample condition	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

Boolean operators

Operator	Description	Sample condition	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 False
not	Reverses the outcome of the expression; true becomes false, false becomes true	rain = True not rain	 False

Programming constructs

Assignment

Definition

Assignment is used to set or change the value of a variable.

`<variable identifier> = <value>`
`<variable identifier> = <expression>`

Examples

`count = 0`	`count = count + 1`
`myName = "Fred"`	`check = isalpha (myName)`

Sequence

Definition

Every instruction comes one after the other, from the top of the file to the bottom of the file.

Examples

`count = 0`
`myName = "Fred"`
`count = count + 1`
	Firstly, sets the value of 'count'
	Secondly, sets the value of '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

Definition

Selection is the way to make decisions. Nesting of selection statements is permitted. However, the programmer should consider the use of 'else' and 'elif' to make the logic clearer.

`if <expression>:` `  <command>`	If <expression> is true, then command is executed.

`if <expression>:`
`  <command>`
`else:`
`  <command>`
	If <expression> is true, then first <command> is executed, otherwise second <command> is executed.

`if <expression>:` `  <command>`	If <expression> is true, then first

elif <expression>:
    <command>
else:
    <command>
	<command> is executed, otherwise the second <expression> test is checked and if true, then second
	<command> is executed, otherwise third <command> is executed.
	
	Supports multiple instances of 'elif'.
	
	The 'else' is not required with the 'elif'.

Examples

```
if (count == 1):
    print ("In first block")
```

```
if (count == 1):
    print ("In first block")
else:
    print ("In second block")
```

```
if (count == 1):
    print ("In first block")
elif (count == 2):
    print ("In second block")
```

```
if (count == 1):
    print ("In first block")
elif (count == 2):
    print ("In second block")
else:
    print ("In third block")
```

```
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")
```

Repetition

Definition

Repetition is when programmers design their code to go around in a loop, executing the same lines of instructions several times. These types of loops keep executing while a condition is true. They are very useful when the programmer does not know, in advance, how many times the loop must go around.

`while <condition>:` `    <command>`	**Pre-conditioned loop. This executes <command> while <condition> is true.**

Examples

```
while (count > 0):
    print ("Count is", count)
    count = count - 1
```

```
while (index < len (studentRecord)):
    print (studentRecord[index])
    index = index + 1
```

```
key = input ("Enter Y or N")
while (key != "N"):
    print ("Going around again")
    key = input ("Enter Y or N")
```

```
key = "X"
while (key != "N"):
    key = input ("Enter Y or N")
    if (key != "N"):
        print ("Going around again")
    else:
        print ("Processing user request")
```

Iteration

Definition

Iteration is another way in which programmers make their code go around in a loop, executing the same lines over and over several times. It is slightly different in that it is not a test that drives the loop, but the length of the data structure being manipulated. Iteration is used to process every item in a data structure along one of its dimensions.

`for <id> in <array>:` `    <command>`	Executes for each element of a structure, in one dimension. This is a very useful construct allowing the programmer to process every item as it is encountered, starting at the front and finishing at the end of the data structure
`for <id> in range (<start>, <stop>):` `    <command>`	Count controlled loop. Executes <command> a fixed number of times, based on the numbers generated by the range function.
`for <id> in range (<start>, <stop>, <step>):` `    <command>`	Same as above, except that <step> influences the numbers generated by the range function.

Examples

```
numbers = [10, 20, 30, 40, 50]
studentTable = [[1524, "Jones", "Rebecca", True, 78.45], \
                [9384, "Collins", "Joseph", False, 24.12], \
                [4543, "Green", "Helen", True, 32.00]]
for num in numbers:
    print (num * 2)
```

```
for student in studentTable:
    print ("Last name:", student[1], "Account balance:",
student[4])
```

```
for index in range (0, len(studentTable)):
    print ("Student ID:", studentTable[index][0])
```

Subprograms

Definition

Subprograms are distinct blocks of code, incorporating their own scope, that may be called into action from other blocks of code. Specifically, the PLS makes a conceptual distinction between 'procedure' and 'function'.

- Functions may or may not take parameters and must always return a value to the calling block.
- Procedures may or may not take parameters and do not return a value to the calling block.
- The code inside a subprogram can change the elements of a data structure when it is passed as a parameter. This may have unintended side effects.

`def <procname>():` `  <command>`	A procedure with no parameters
`def <procname>(<paramA>, <paramB>):` `  <command>`	A procedure with parameters

`def <funcname>():`
`  <command>`
`  return (<value>)`
	A function with no parameters

`def <funcname>(<paramA>, <paramB>):`
`  <command>`
`  return (<value>)`
	A function with parameters

Examples

```
def displayStudents ():
    for student in studentTable:
        print (student[2], student[1])
```

```
def displayOneStudent (pIndex):
    print (studentTable[pIndex][2], studentTable[pIndex][1])
```

```
def roll():
    showing = random.randint (1, 6)
    return (showing)
```

```
def quantity (pExp):
    value = 2 ** pExp       # Calculates 2^pExp
    return (value)
```

Inputs and outputs

Screen

The PLS supports the use of 'print()' to display output to the screen. To make output more user-friendly, programmers can use the 'string.format()' method with the positional '{}' arguments. Use of this method is discussed in a later section.

Keyboard

The PLS supports the use of 'input()" to retrieve items typed in from the keyboard. Programmers need to remember that all input from the keyboard is character and must be coerced to the data type required.

Files

The PLS supports manipulation of simple comma separated value text files. No other files need to be considered.

File operations include open, close, read, write, and append. Files should be opened for reading or writing, not both at the same time.

Beginner programmers follow the simple logic of:

- Open file
- For each line in the file
    - Read the line
    - Immediately process the line or store the needed data in an internal data structure
- Close file

If a file is to be updated after a program is run, programmers follow the simple logic of:

- Open file for writing
- For each record in the internal data structure
    - Create a string, consisting of comma separated values
    - Write the string to the file
- Close file

Definition

`<fileid> = open(<filename>, "r")`	Opens file for reading
`for <line> in <fileid>:` `  <command>`	Reading every line into a structure
`<fileid> = open(<filename>, "w")`	Opens file for writing
`for <record> in <datastructure>:` `  <command>`	Writing every line from a structure
`<fileid>.close()`	Closes file

Examples

```
theFile = open("Students.txt", "r")
```

```
for line in theFile:
```

```
theFile.close()
```

```
outFile = open("Students.txt", "w")
```

```
for student in studentTable:
    outFile.writelines (<commaseparatedvaluestring>)
```

```
outFile.close()
```

Library modules

In some languages, subprograms are referred to as methods, or procedures or functions. Sometimes, these words are used interchangeably. The Programming Language Subset supports a restricted set of the programming language's library modules and an even smaller set of actual subprograms found in them.

Built-in functions

Please see the Programming Language Subset document for a full list of built-in functions supported. No import statements are necessary.

Function	Example
chr(integer)	character = chr (code)
input(prompt)	choice = input ("Make a choice")
len(object)	size = len (studentTable)
ord(char)	code = ord (character)
print(item)	prints item to the display
range(start, stop, step)	indices = range (10, 0, -1)

List methods

Please see the Programming Language Subset document for a full list of list methods supported. No import statements are necessary.

There are many functions that work on lists, but only a small subset is needed by the PLS. This table shows examples of some of the functions.

Function	Example
`list.append(item)`	`studentTable.append (line[2])`
`del <list>[<index>]`	`del studentTable[3]`
`list.insert (index, item)`	`studentTable.append (4, line[2])`
`<item> = list()` `<item> = []`	`studentTable = list()` `studentTable = []`

It is not necessary to remove items from a list. If a programmer wants to make a shorter version of an existing list, containing only some of the records, then the needed records can be copied, using indexing, from the original list to the new shorter list. In this way, none of the original data is ever lost.

Random module

Please see the Programming Language Subset document for a full list of random functions supported. To use random number generators, the 'random' module must be imported. This table shows examples of some functions.

Function	Example
`random.randint (a,b)`	`guess = random.randint (1, 100)`
`random.random()`	`percent = random.random() * 100`

String module

Manipulating strings

Please see the Programming Language Subset document for a full list of supported functions in the string module. No imports are required. Recall that individual characters are accessible in a string via indexing. This table shows examples of some supported functionality.

Function	Example
len()	`myFurniture = ["sofa", "chair", "bed", "table"]` `countFurniture = len (myFurniture)`

find()	`sentence = "Mary had a little lamb"`
	`target = "little"`
	`location = sentence.find(target)`

format()	`outString = "My name is {}.  I am {}"` `print (outString.format("Sarah",16))`

isalpha()	`name = "H311o"`
	`if (not (name.isalpha())):`
	`    print ("Invalid name")`

isdigit()	`isbn = "1234567890123"`
	`if (isbn.isdigit()):`
	`    print ("Valid isbn")`

split()	`line = "123, Bloggs, Fred, 12.48"`
	`splitLines = line.split(",")`
	`for item in splitLines:`
	`    print (item)`

lower()	`outString = "HELLO World"` `print (outString.lower())`

isupper()	`choice = "ABC"`
	`if (choice.isupper()):`
	`    print ("All Capitals")`
	`else:`
	`    print ("Error")`

Concatenation of strings is done using the + operator.

Formatting strings for output

Output can be customised to suit the problem requirements and the user by using the `string.format()` method. This method uses descriptors inside curly brackets to describe how the content of identifiers, once substituted in, should be formatted.

This example illustrates how to use the positional arguments to the

```
layout = "{:<6} {:<10} {:<10} {:^5}"
student = [1524, "Jones", "Rebecca", True, 78.45]

print (layout.format ("Num", "First", "Last", "Cost"))

layout = "{:<6} {:<10} {:<10} {:^6.2f}"
print (layout.format (student[0], student[2], student[1],
student[4]))
```

`string.format()` method.

This is the output from using the positional arguments to control columnar output.

```
Num    First      Last       Cost
1524   Rebecca    Jones      78.45
```

Math module

Please see the Programming Language Subset document for a full list of supported functions in the math module. The math module must be imported. This table shows examples of some supported functionality.

Function	Example

`math.ceil (r)`	`print (math.ceil(100.001))`
	`print (math.ceil(30.000))`
	`print (math.ceil(math.pi))`

`math.round(x, n)`	`print (round(3.14159, 2))`
	`print (round(123.456789, 3))`
	`print (round(-100.5602, 3))`

`math.pow(x, y)`	`print (math.pow (100, 2))`
	`print (math.pow (2, 3))`
	`print (math.pow (2, -3))`

Time module

Please see the Programming Language Subset document for a full list of supported functions in the time module. The time module must be imported. This table shows examples of some supported functionality.

Function	Example
`time.sleep(sec)`	`time.sleep (5)` `time.sleep (0.5)`

Turtle graphics module

Please see the Programming Language Subset document for a full list of supported functions in the turtle module. The turtle module must be imported. This table shows examples of some supported functionality.

```
myTurtle = turtle.Turtle()
myTurtle.home ()
myTurtle.pencolor ("red")
myTurtle.pensize(10)
myTurtle.forward (50)
myTurtle.right (90)
myTurtle.pencolor ("blue")
myTurtle.forward (50)
myTurtle.hideturtle ()
junk = input ("Waiting")
```

Functionalities not in the PLS

Although the Programming Language Subset is based on an existing high-level programming language, students do not need to understand or be able to use any features not expressly set out in the PLS document.

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 the break statement.

```
22  userNum = 0
23
24  # Using an infinite loop and a break statement
25  while (True):
26      userNum = int (input ("Enter a number"))
27      if (userNum == 0):
28          break
29      else:
30          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.

```
32  # Revision using a condition-controlled loop without a break statement
33  userNum = int (input ("Enter a number"))
34  while (userNum != 0):
35      print ("The number is:", userNum)
36      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 the exit statement.

```
22  userNum = 0
23
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.

January 2020