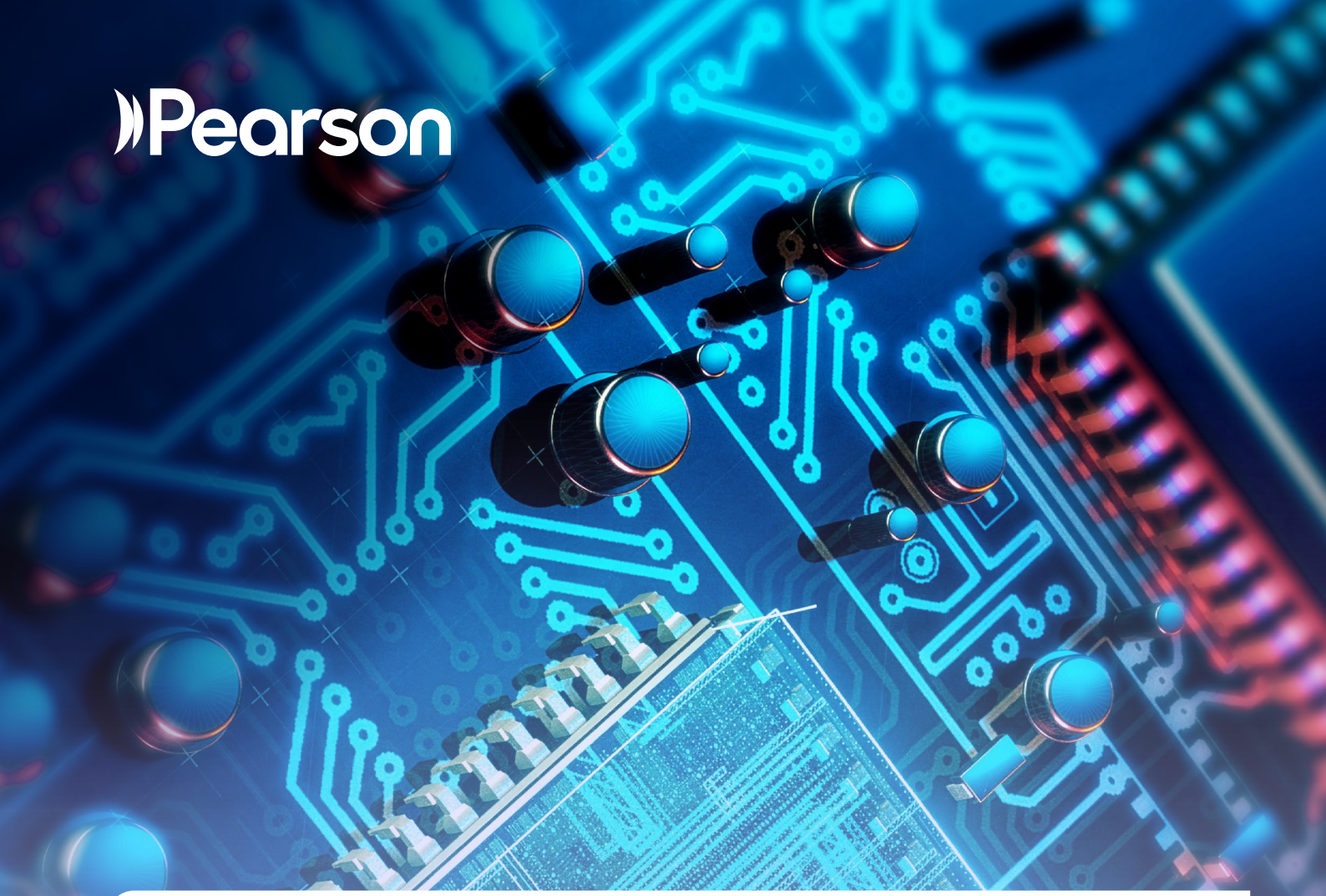


International Advanced Level

Computer Science

Good Programming Practice Guide (GPPG)

Pearson Edexcel International Advanced Subsidiary in Computer Science (XCP01)
Pearson Edexcel International Advanced Level in Computer Science (YCP01)

First teaching September 2026 | First examination June 2027

First certification from June 2027 (International Advanced Subsidiary)
and August 2028 (International Advanced Level)

Issue 1

Contents

Introduction

The Programming Language Subsets (PLS) are documents that specify which parts of Python 3 are required in order that the Unit 2 and Unit 4 assessments for IAL Computer Science can be undertaken with confidence. Learners familiar with everything in those documents will be able to access all parts of the Unit 2 and Unit 4 assessments. This does not stop a teacher/learner 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.

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

The information in this document should not prevent learners 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 learners.

While learners will have a copy of the relevant PLS available during the Unit 2 and Unit 4 examinations, the GPPG must not be used during the examination.

The information provided in this document gives opportunities for learners to engage with transferable practices. Python has some behaviours that do not align with other industry-standard programming languages. Experienced programmers may choose to take advantage of them. However, restricting learning to Pythonic behaviours means that relearning is required when changing programming language.

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

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.

Programming

Flow control constructs

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`
`name = "Fred"`
`count = count + 1`
	Sets the value of count to zero.
	Sets the value of name to Fred.
	Increments the value of count.

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 statement is used.

Where there are multiple conditions to be checked, an if…elif…else statement 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 elif…else 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 statement. 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 statement.

`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

Match-case statement

Selection may also be implemented using the match-case statement. This construct matches an expression against multiple cases. It has the same behaviour as multiple if…elif…elif…else statements.

The case statement does not permit logical operators. Where the code block is short and simple, the code block could be repeated. Alternatively, the case statement does permit the use of bitwise operators, so the or operator | could be used. A third option to prevent long duplicated blocks or hard to read case statements is to create a subprogram and invoke it.

The last case in the sequence is the default. If no others are matched, then the code for the default is executed. It is always best practice to include a default case for unanticipated inputs.

This table shows examples of a match-case statement and how to format it.

Example	Description

`day = 15`
 	
`def display (p_string):`
`  print(p_string)`
 	
`match day:`
`  case 1:`
`  print("Monday")`
`  case 2:`
`  print("Tuesday")`
`  # . . . . . .`
`  case 6 | 7:`
`  display("Weekend")`
`  case _:`
`  print("Invalid")`
	The variable day is matched to determine the name of the day.

Iteration

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

Count-controlled

Count-controlled loops keep repeating for a set number of passes through the loop. The number of passes through the loop is known or can be determined at the start of the loop. The Python range() function is used to generate a sequence of numbers. Loops that use range() are count-controlled.

This table shows examples of using count-controlled loops.

Example	Description

`count = 4`
`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.

`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.

`table = [[152, "Jones", 78.45],`
`  [938, "Black", 24.12],`
`  [454, "Green", 32.00]]`
`for student in table:`
`  print("Name:",`
`  student[1],`
`  "Balance:",`
`  student[2])`
	This for loop processes every student in table. 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 example:
	`Name: Jones Balance: 78.45`
	`Name: Black Balance: 24.12`
	`Name: Green Balance: 32.0`

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

`sales = [[80.00, 73.09, 99.00],`
`  [78.24, 65.59, 78.12],`
`  [77.70, 83.19, 95.57]]`
`for week in sales:`
`  total = 0.0`
	This example uses nested loops to process all the fields in each record of sales.
	
	This is the output from this example:

`    for day in week:`
`        total = total + day`
`    print(total)`
	`252.09`
	`221.95`
	`256.46`

Condition-controlled

For condition-controlled loops, the number of passes around the loop is not known before the loop starts. The loop keeps going around as long as a condition evaluates to True. This table shows examples of using condition-controlled loops.

Example	Description

`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 the letter 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 loop (while).

Exception handling

Using exception handling is good practice as it isolates the error handling code from the rest of the code. It also simplifies flow control, aids readability, and makes debugging easier. It makes program code respond gracefully or elegantly to errors by taking alternative actions. No programmer or user sets out to make the code crash.

The most important reason to implement exception handling, or any error handling for that matter, is to clean up and release any resources before termination. Consider the situation where a file is opened, half the data is written, and then there is an error. It's not enough to just exit the program. Good practice dictates that the file be destroyed as well, so that it is not inadvertently used as input to the next stage of the process.

That said, it's not always necessary to implement exception handling, particularly in simple, straight-forward programs that involve no jeopardy with resources.

There is a general exception class, Exception, and specifically named errors, which are found in the PLS.

This table shows examples of using exception handling.

Example	Description

`try:`
`    result = number / 0`
`except Exception as e:`
`    print("Error:", e)`
	In this case an arithmetic error, divide by zero, has occurred, but the code catches the general class Exception.

`try:`
`    handle = open ("File.txt", "r")`
`    for line in handle:`
`        print(line)`
`    handle.close ()`
`except FileNotFoundError as e:`
`    print ("Error:", e)`
	In this case, a named error is caught, particularly the one indicating that the file does not exist already.

Variables and constants

Variable declaration

It is important that learners 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()`
float	A number with a fractional part, also known as decimal (negative or positive)	float	`total_price = float()`
Boolean	Data that can only have one of two values, either true or false	bool	`lights = bool()`
string	A sequence of letters, numbers, symbols, etc., usually available from the keyboard	str	`name = "Rohan"`
character	A single letter, number, symbol, etc., usually available from the keyboard	str	`my_initial = 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 variable initialised to 0.175
`count = 0`	An integer variable initialised to 0.
`window_state = True`	A Boolean variable initialised to True.
`my_initial = "J"`	A character variable initialised to J.

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

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, usually at the top of the code file.

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, learners are expected to use constants in their program code.

Good practice means:
writing constant names in capital letters to distinguish them from variables
assigning values to constants at the start of the program ensuring that the values of constants are not changed during execution of the entire program.

Operators

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

BIDMAS means to carry out these operations in order:

- Brackets
- Indices (exponentiation)
- Division
- Multiplication
- Addition
- Subtraction

Learners need to know the order of precedence rules that determine the order in which a calculation is executed.

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`

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`

Bitwise operators

This table shows examples of the bitwise operators, using the variables arg1 and arg2, which are each eight bits in length.

```
arg1 = 00010111
arg2 = 11001011
```

Operator	Description	Example	Evaluates to
&	AND	`arg1 & arg2`	`00000011`
|	OR	`arg1 | arg2`	`11011111`
^	XOR	`arg1 ^ arg2`	`11011100`
~	NOT	`~arg1`	`11101000`
<<	Logical shift left	`arg2 << 2`	`00101100`
>>	Arithmetic shift right	`arg2 >> 3`	`11111001`

Organising and handling data

Data types

The basic data types are integer, float, char, Boolean, and string. In Python these are designated as int, float, bool and str.

Python does not have a dedicated character data type so uses str instead.

Type 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 content 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. This method is good practice, as it reduces side effects. In this instance, the side effect would be the change in data type of the original count variable.

Data structures

The data structures, in this section, are the simplest way to organise data. They align conceptually with the way computers organise memory.

A data structure, in general terms, is a collection of items, which themselves are typed. Each item in the collection is accessible. The type of structure dictates the method of access. Where items are accessible by indexing, indexes start with the value 0.

Dimensions

Learners are only expected to work with indexable data structures that are one- or two-dimensional.

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 learners 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	Example declaration

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.	`[`
		`  [ ],`
		`  [ ], …`
		`  [ ]`
		`]`
			`sensor_readings = [`
			`  [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.	`[`
		`  [ ],`
		`  [ ], …`
		`  [ ]`
		`]`
			`record_table = [`
			`  [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".

Arrays and records

This table shows examples of the different data structures.

Data type	Explanation	PLS	Example declaration
string	A sequence of characters	`str`	`name = str()` `name = "Joseph"`
array	A sequence of the same type item, with a fixed length	`[ ]`	`grades = [80, 75, 90]`

record	A sequence of items, usually of mixed data types, with a fixed length	`[ ]`	`student_record =`
			`[1524, "Jones",`
			`"Rebecca", True,`
			`78.45]`

In the above table:

- `student_record[2]` holds the string Rebecca,
- `grades[0]` holds the integer 80,
- `name[5]` holds the character h.

Note, that the data structure array is not the same as the Python data structure list. In the PLS, they are both created using lists.

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.

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	Output

`things = ["cup", "plate", "fork", "spoon"]`
`counts = [5, 8, 4, 3]`
`for index in range(len(things)):`
`    print(things[index], counts[index])`
	`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. Therefore, where possible, store data as multi-dimensional lists of records, consisting of mixed data types.

Abstract data types

Abstract data types are different to the data structures in the previous section. An abstract data type has predictable behaviours that are not dependent upon how they are implemented inside the computer. For example, a dictionary has key:value pairs, but they are not stored like that inside the computer.

List

A list is the easiest abstract data type to understand. Its behaviours include creating, deleting, adding, inserting, removing and accessing the items. Remember, indexes start at zero.

A list is identifiable by the square brackets, [ ], surrounding the data items.

This table shows examples of using a list.

Example	Description
`kitchen = ["pot", "pan", "plate"]` `print(kitchen)`	This example shows a way declare and initialise a list of strings. The output is: `['pot', 'pan', 'plate']`

`kitchen.append("stove")`
`kitchen.insert(1, "bowl")`
`print(kitchen)`
	This example shows two ways of putting data into a list. The append adds data to the rear. The insert places data into the index location. It does not replace the data at the index, as all the data to the right conceptually moves toward the end of the list. Remember, indexes start at zero.
	
	The output is:
	`['pot', 'bowl', 'pan', 'plate', 'stove']`

`print(kitchen[0])`
`print(kitchen[2:4])`
	This example shows two ways of accessing data in a list. The first is by indexing and the second is by slicing. Remember, the start index is included in the slice, but the stop index is not included in the slice.
	
	The output is:
	`pot`
	`['pan', 'plate']`

`print(kitchen.pop(3))`
`print(kitchen)`
	This example shows a way to remove and retrieve data in a list. In this case, the data at the index is removed from the list and returned to the calling code. All data to the right of the index conceptually moves toward the front of the list.
	
	The output is:
	`plate`
	`['pot', 'bowl', 'pan', 'stove']`

`alist = [30, 50, 10, 40, 20]`
`alist.sort()`
`print(alist)`
 	
`blist = ["pot", "cup", "spoon"]`
`blist.sort(reverse = True)`
`print(blist)`
	This section Is applicable to Unit 4 only.
	
	These examples show how to sort numeric and string data stored in a list.
	
	The output is:
	`[10, 20, 30, 40, 50]`
	`['spoon', 'pot', 'cup']`

Dictionary

The behaviours of a dictionary are also easy to understand. This abstract data type behaves as you would anticipate, using a key to identify a data item. The remainder of the item is referred to as the value. Together they're called a key:value pair. Interestingly, the value part of the pair can be other data structures or abstract data types.

Its behaviours include creating, adding, removing and accessing the items. A dictionary cannot be indexed or sliced. However, the value part of an existing key:value pair can be changed.

A dictionary is identifiable by the curly braces, {}, surrounding the key:value pairs.

This table shows examples of using a dictionary.

Example	Description

`kitchen = ["pot", "pan", "plate"]`
`bedroom = ["bed", "pillow",`
`"dresser"]`
`house = {"kitchen": ["pot", "pan",`
`"plate"]}`
`house["bedroom"] = bedroom`
`print(house)`
	This example shows creating a dictionary and adding a key:value pair to it.
	
	The output is:
	`{'kitchen': ['pot', 'pan',`
	`'plate'], 'bedroom': ['bed',`
	`'pillow', 'dresser']}`

`print(house.get("bedroom"))`	This example shows accessing an item using a key. The output is: `['bed', 'pillow', 'dresser']`

`print(house.keys())`
`print(list(house.keys()))`
	This example shows one method for retrieving all the keys in a dictionary.
	
	The structure returned is a dict_keys type. It can be converted to a different type for easier manipulation.
	
	The output is:
	`dict_keys(['kitchen', 'bedroom'])`
	`['kitchen', 'bedroom']`

`del house["kitchen"]` `print(house)`	This example shows a way to remove a key:value pair from the dictionary. The output is: `{'bedroom': ['bed', 'pillow', 'dresser']}`

Tuple

The behaviours of a tuple may be new, but they are consistent. This abstract data type is useful for problems where the structure must not change.

Its behaviours include creating and accessing the items. A tuple can be indexed. However, a tuple or items in the tuple cannot be changed. Attempting to do so will cause a runtime error.

A tuple is identifiable by the round brackets, ( ), surrounding the data items.

This table shows examples of using a tuple.

Example	Description
`kitchen = ("pot", "pan", "plate")` `print(kitchen)`	This example shows creating a tuple. The output is: `('pot', 'pan', 'plate')`

`print(len(kitchen))`
`print("pot" in kitchen)`
	This example shows two ways of manipulating a tuple.
	
	The output is:
	`3`
	`True`

`for item in kitchen:`
`  print(item)`
`print(kitchen[0:2])`
	This example shows two ways of manipulating a tuple.
	
	The output is:
	`pot`
	`pan`
	`plate`
	`('pot', 'pan')`

Set

The behaviours of a set are consistent, but there are some unintuitive characteristics. This abstract data type is useful for problems where membership and non-duplication are important.

Its behaviours include creating, adding and removing. Traditional logical operations of difference, intersection, union, and exclusive or can be carried out on sets.

A set cannot be indexed or sliced. Sets are unordered. That means displaying a set at different times or during different executions of a program will give the same data, but may give a different order.

A set is identifiable by the curly braces, {}, surrounding the data items.

This table shows examples of using sets.

Example	Description

`kitchen = {"pot", "pan", "towel"}`
`bedroom = {"bed", "pillow"}`
`bedroom.add("towel")`
`print(kitchen)`
`print(bedroom)`
	This example shows creating two sets.
	
	The output is:
	`{'towel', 'pot', 'pan'}`
	`{'towel', 'bed', 'pillow'}`

`print("pot" in kitchen)`
`print("pot" in bedroom)`
	This example shows the membership operation.
	
	The output is:
	`True`
	`False`

`print(kitchen.union(bedroom))`
`print(set.intersection(kitchen,`
`bedroom))`
	This example shows the union and intersection operations on sets.
	
	The output is:
	`{'pot', 'towel', 'bed', 'pan',`
	`'pillow'}`
	`{'towel'}`

`print(kitchen - bedroom)`
`print(kitchen ^ bedroom)`
	This example shows the difference operator and the exclusive or operator.
	
	The output is:
	`{'pot', 'pan'}`
	`{'pan', 'pillow', 'pot', 'bed'}`

Numeric data

As with all data, numeric data should be stored in a way that is suitable for the processing performed on it. Numeric data is best stored and manipulated as numeric data types.

This table shows examples of manipulating numeric data.

Example	Description
`print(chr(35))`	This example shows the mapping of integers to characters. The output is: #

`print(round(3.14159, 2))`
`print(round(15.53, 0))`
`print(int(round(15.53, 0)))`
`print(f"{round(15.53,`
`0):.0f}")`
	This example shows rounding of floating-point numbers. Round uses the 0.5 rule for rounding up or down.
	
	The print() function displays floating-point numbers with one decimal place, as default. Format or convert the result the result of round() to control that.
	
	The output is:
	`3.14`
	`16.0`
	`16`
	`16`

`print(*range(5, 56, 10))`	This example shows the result of the range() function. The range() function is most often used to generate indexes for iteration. However, it actually generates a whole range of numbers. The lower bound is included. The upper bound is excluded. Note, disregard the * operator. It is used to unpack the sequence only. The output is: `5 15 25 35 45 55`

Decimal library module

This section is applicable to Unit 4 only.

It's especially important in some contexts that decimal numbers are stored and manipulated with as much precision as possible. For simple problems, two decimal places may be sufficient, but for accounting and financial contexts, more decimal places are needed. Scientific contexts require even more decimal places and precision than financial contexts. Many programming languages have specialised types, classes and library modules that provide this type of assistance for programmers. In Python, the decimal library module provides these facilities.

In using the decimal library module, some understanding of terminology is useful, so that its behaviour is clear.

Significant digits:

- Include:
    - All non-zero digits
    - Any zeros between significant digits
    - Trailing zeros to the right of the decimal
- Exclude:
    - Leading zeros to the left of any non-zero digits

Precision:

- Reflects the confidence in the accuracy of a measurement or calculation
- The more significant digits, the more precise the number

Some constraints around the decimal library module include:

- Mixed type arithmetic generates a runtime error
- The precision value is applied only after arithmetic operations

This table shows examples of manipulating numeric data with the decimal library module subprograms.

Example	Description

`import decimal`
`pi_string = "3.141592653589793"`
`phi_string = "1.618033988749895"`
`pi_decimal =`
`decimal.Decimal(pi_string)`
`phi_decimal =`
`decimal.Decimal(phi_string)`
 	
`print(pi_decimal)`
`print(phi_decimal)`
	This example demonstrates creation of decimal type variables
	
	The output is:
	`3.141592653589793`
	`1.618033988749895`

`decimal.getcontext().prec = 10`
`result = pi_decimal /`
`decimal.Decimal("1.0")`
`print(result)`
	This example demonstrates the effect of setting the precision value and performing an arithmetic operation.
	
	The output is:
	`3.141592654`

`result = pi_decimal.min(phi_decimal)` `print(result)`	This example demonstrates one of the subprograms available in the decimal library module. The output is: `1.618033989`

`decimal.getcontext().prec = 4`
 	
`result = phi_decimal + pi_decimal`
`print(result)`
 	
`result = phi_decimal - pi_decimal`
`print(result)`
	This example demonstrates the impact of changing the precision on arithmetic operations between two variables, each stored as decimal.
	
	The output is:
	`4.760`
	`-1.524`

String data

Strings can be manipulated in many different ways, including indexing and slicing. String subprograms are most often used for validation and are usually used in combination.

Concatenation of strings is done using the + operator.

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

This table shows examples of manipulating strings using the string manipulation 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 checks to see if the string is all digits.
	
	The output is:
	`True`

`in_str = "ab12CD"`
`out_str = ""`
`for index in range(len(in_str)):`
`    char = in_str[index]`
`    if (char.isalpha()):`
`        out_str = out_str + char`
`out_str = out_str.lower()`
`print(out_str)`
	This example shows one way to remove non-alphabetic characters from a string, using the string manipulation subprograms.
	
	The output is:
	`abcd`

`in_str = "ab12CD"`
`out_str = ""`
`for index in range(len(in_str)):`
`    char = in_str[index]`
`    if (char.isalpha()):`
`        out_str = out_str + char`
`out_str = out_str.lower()`
`print(out_str)`
	This example shows one way to remove non-alphabetic characters from a string, using the string manipulation subprograms.
	
	The output is:
	abcd

Formatted string literals (f-strings)

Formatting of output so that it is suitable for the user is very important. It's very important to refrain from formatting unless the result is leaving the program, e.g. display to the screen or writing to a file. The controls for formatting are extensive. More documentation is in the PLS.

This table shows examples of formatting using string literals. Output of data in tabular form is demonstrated in the **date and time** section.

Example	Description
`arg1 = 23` `print(f"{arg1:08b}")`	This example demonstrates displaying a variable as a Boolean string. The output is: `00010111`

`data = [["AB154", "Person 1",`
`45, 45.7],`
`["CE072", "Person 4",`
`312, 12.23]]`
 	
`for person in data:`
`print(f"{person[1]} has a key of`
`{person[0]}.")`
`print(f"They have {person[2]}`
`points and`
`£{person[3]:0.2f}")`
	This example demonstrates controlling spacing, integer, and floating-point variables.
	
	The output is:
	`Person 1 has a key of AB154.`
	`They have 45 points and £45.70`
	`Person 4 has a key of CE072.`
	`They have 312 points and £12.23`

Date and time

Dealing with dates and times can be challenging, even for experienced programmers. In order to make this more manageable, the PLS states that only ISO 8601 is used. Practising with the datetime library subprograms and f-strings will help. More documentation is in the PLS.

This is an example of formatting data, including dates, in a tabular form.

```
import datetime
data = [["AB154", "Person 1", 1730419200],
        ["CD815", "Person 2", 1730420400],
        ["AA034", "Person 3", 1730775000],
        ["CE072", "Person 4", 1730611200]
        ]

print (f"{'Key':<5} {'String':^10} {'Date':<10}")
print ("-" * 27)
for person in data:
    stamp = datetime.datetime.fromtimestamp(person[2])
    print(f"{person[0]:^5} {person[1]:^10} {stamp:%Y-%m-%d}")
```

The output is:

```
Key     String   Date
---------------------------
AB154  Person 1  2024-11-01
CD815  Person 2  2024-11-01
AA034  Person 3  2024-11-05
CE072  Person 4  2024-11-03
```

RegEx library module

This section Is applicable to Unit 4 only.

Regular expressions (regex) are powerful pattern matching tools used for validation, search and replace and analysing output logs from programs.

It is good practice to put the letter r in front of strings for the RegEx library module. This makes it easier to deal with the backslash, which is also the escape sequence character.

Contains a word – like 'Error' in log data
Match beginning letters
Match a full pattern (key)

Example	Description

`import re`
 	
`text = "Log Error Module Time"`
`text = text + " log Error module error"`
 	
`pattern = r"Error"`
`result = re.findall(pattern, text)`
`print(result)`
`result = re.split(pattern, text)`
`print(result)`
	This example demonstrates finding all the occurrences of a word in text. Notice that the first output is all the matches, while the second is the text that does not match.
	
	The output is:
	`['Error', 'Error']`
	`['Log ', ' Module Time log ', ' module error']`

`pattern = r"[Ee][Rr][Rr][Oo][Rr]"`
`match = re.search(pattern, text)`
`start_index = match.start()`
`print(f"Found at index: {start_index}")`
	This example demonstrates finding the index of the first occurrence of a pattern. The pattern is described in a different way.
	
	The output is:
	`Found at index: 4`

indices = []
start = 0
pattern = r"Error|error"
 	
match = re.search(pattern,
text[start:])
while (match):
  if match:
  indices.append(start +
  match.start())
  start = start +
  match.start() + 1
  match = re.search(pattern,
  text[start:])
 	
print(f"Found at indexes: {indices}")
	This example demonstrates finding all the indexes where the pattern occurs. The pattern is described in a different way.
	
	The output is:
	Found at indices: [4, 26, 39]

text = "Log error Module Time" text = text + " log Error module error" updated_text = re.sub(r'error', 'Error', text) print(updated_text)	This example demonstrates replacing all occurrences of error with Error. The output is: Log Error Module Time log Error module Error

keys = ["AB123CD", "1A123", "&",
"Ef987Gh", "123", "WX345YZ"]
 	
regex = r"^[a-zA-Z]{2}[0-9]{3}
[a-zA-Z]{2}$"
for key in keys:
  match = re.search(regex, key)
  if (match is not None):
  print(key)
	This example demonstrates identifying keys that have a valid pattern.
	
	The output is:
	AB123CD
	Ef987Gh
	WX345YZ

Text files

Having a way to save persistent data even when the program is finished is very important. Writing data to a file provides this persistence. Reading data from a file allows larger amounts of data to be processed. It also allows the same program to be run on different data, without someone having to type it in.

All data stored on disk for this qualification will be stored as 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 this specification.

Learners 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
`the_file = open("Students.txt", "r")`	This example opens a file for reading only. The file handle returned is used in all subsequent operations requested on the file.
`for line in the_file:` `  <process the line>`	This example shows how to read each line from the file, using the file handle variable returned from the open() function. The commands indented under the for…in loop will execute for each line read from the file.
`the_file.close()`	This example closes a file, using the file handle returned from the open() function.
`out_file = 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. If the file does not exist, it is created.

`for student in student_table:`
`  <build an output string>`
`  out_file.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 file handle returned from the open() function.

`out_file.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. However, this construction does not translate to all programming languages. Using the approach set out in the table above is transferable.

It is possible to read the contents of an entire file into a program with a single Python subprogram call. This operation is dependent on available memory. Reading data in this manner is not a transferable problem-solving approach. Real-life situations often generate more data than could possibly be held in memory at one time. Using the approach set out in the table above is transferable.

Reading from a file

The general approach for reading data from a file is to:

- open the file for reading
- read each line from 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

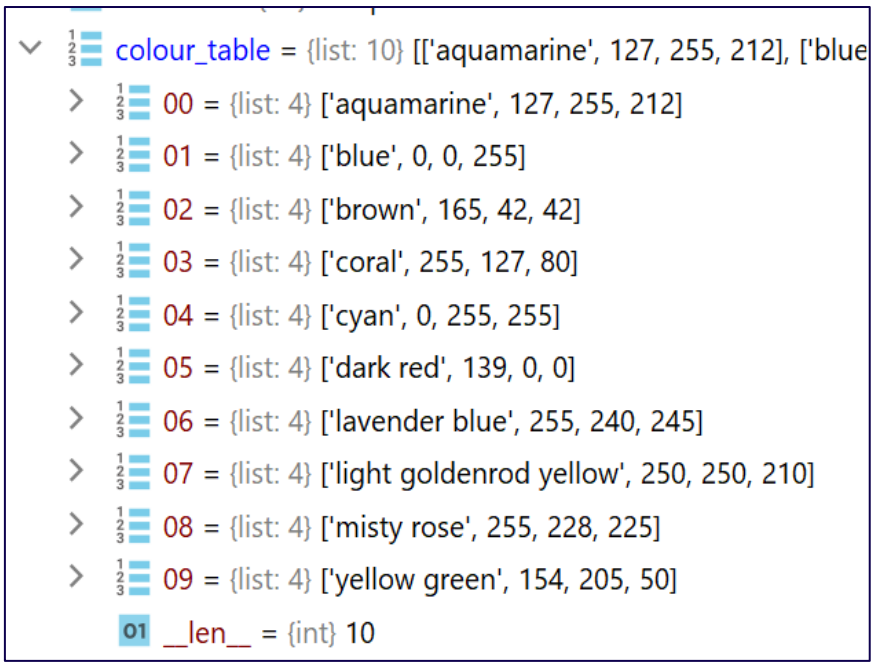
This example shows the recommended approach to working with files.

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:



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.

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

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

# ------------------------------------------------------------
# Subprograms
# ------------------------------------------------------------
def load_data (p_file):
    the_file = open(p_file, "r")     # Open the file

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

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

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

        # Append the first string field
        the_record.append(the_fields[0])

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

        # Append the record to the data structure
        colour_table.append(the_record)

    the_file.close()                # Close the file

```

```
def add_column (p_table):
    total = 0

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

def save_data (p_file):
    out_line = ""            # To write to the file
    the_file = open(p_file, "w")

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

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

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

        # Write the line to the file
        the_file.write(out_line)

    the_file.close()        # Close the file

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

# Process the 2D data structure in some way
add_column(colour_table)

save_data(OUTFILE)      # Save the data to a file

```

Control characters

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

Carriage return means to go back to the beginning of the current line without going down to the next line. Line feed means to go down to the next line. Horizontal tab means to go to the next tab alignment after the current cursor position. Each is a non-printable ASCII character that has an equivalent string in programming languages.

Name	Abbreviation	ASCII hexadecimal	String
Carriage return	CR	0x0D	"\r"
Line feed	LF	0x0A	"\n"
Horizontal tab	HT	0x09	"\t"

These characters are used in some combination to control outputs. Unfortunately, not every operating system uses the same ones. However, editors automatically convert input and output files to make sure they work properly. In Python, print () automatically adds them so that the console output appears on separate lines.

When writing code to handle files, a programmer will need to remove some of these characters when reading lines from files and add them when writing lines to files. If needed, they are added with string concatenation. If needed to be removed, they are removed using the strip() or replace() string manipulation subprograms.

Relational databases

This section is applicable to Unit 2 only.

Many functionalities are provided by third-parties, through the use of application programming interfaces (API). They support interactions between different platforms and vendors. The SQLite3 library is just one of these APIs.

The steps to set up and tear down communication with a database are standard. Any executable query submitted between these two points takes the form of an SQL statement.

This table shows examples of set up, query submission, and tear down of communications using the SQLite3 API.

Example	Description

`import sqlite3`
`db_con = sqlite3.connect("House.db")`
`db_cur = db_con.cursor()`
	This example opens a connection to the database. The returned cursor handle is used for any subsequent requests to the database, through this connection.

`db_query = "SELECT * FROM kitchen;"`
`db_cur.execute(db_query)`
`db_result = db_cur.fetchall()`
`for item in db_result:`
`  print(item)`
	This example creates a string query and requests execution by the database. The results are assigned to a local variable. This variable is accessed and each row displayed.
	
	The output is:
	`(0, 'Chair')`
	`(1, 'Spoon')`
	`(3, 'Plate')`
	
	Notice that the results are tuples, which are immutable.

`db_cur.close()`	This example closes the connection to the database.

It is best practice and conventional to use uppercase for all SQL keywords. The semi-colon at the end of each query is also good practice, although not every relational database engine requires it.

Best practice

Very often there is a desire to create a solution, any solution, for a problem. In computer science, the standard is set higher. Just any old solution is not good enough. A good solution is well-designed, robust, considers resource usage, while ensuring it is easy to debug, read, and maintain.

Modularity

Modularity is achieved by separating program code into separate files or separating functionality into subprograms.

Subprograms

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 of logic, making it easier to read, understand and debug. It also allows for separation of concerns, a term that means each subprogram should do one job and only one job. It's equivalent to organising decomposition.

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	Description
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.

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 process_menu_choice (p_choice):
```

Learners 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 libraries of built-in subprograms that perform common tasks. Python also allows programmers to write their own user-devised subprograms and import libraries of subprograms.

This table shows examples of using user-devised subprograms.

Example	Description

`student_table =[[152,"Jones",78.45],`
`[938,"Black",24.12],`
`[454,"Green",32.00]]`
`def display_students ():`
`for student in student_table:`
`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`

`student_table =[[152,"Jones",78.45],`
`[938,"Black",24.12],`
`[454,"Green",32.00]]`
`def display_one (p_index):`
`print(student_table[p_index][2],`
`student_table[p_index][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`

`def raise_power (p_power):`
`    value = 2 ** p_power`
`    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

`display_students()`	This is an example of how to call a procedure with no arguments. The output is listed above.
`my_student = 2` `display_one(my_student)`	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.
`the_power = raise_power(3)` `print(the_power)`	This is an example of a call to a function that takes a single argument.

Variable scope

The scope of a variable refers to the part of a program in which it exists and can be used.

Local variables

Local variables are defined inside blocked code or subprograms. They are only accessible within the block or subprogram in which they are defined and cease to exist once that blocked code or subprogram finishes executing. A local variable with the same name as a variable declared in the main program, a different level of scope, or in a different subprogram is in reality a different variable. The names point to two different memory locations.

Variables that are only used to do work inside blocks, such as loops, are automatically defined as local variables. The for loop is a good example of this. The iterator variable, i.e. the name following the for statement, is treated as a local variable.

Global variables

Global variables are defined at the level of the main program and are accessible from anywhere in the program.

Where variables need to be accessed in global scope, they should be defined in global scope. The layout section shows a way to group all global variables together at the highest level in a main program file. This is good practice.

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.

Global keyword

The global keyword should be used sparingly and with extreme caution. It allows a variable, existing in global scope, to be modified from inside a subprogram. Worse, it allows the creation of a variable in global scope from inside a subprogram. This hidden creation can cause complexities in debugging and conflicts with other named variables. Learners are advised not to use this facility.

Again, using global variables complicates debugging and maintenance of code. Good practice in design dictates that a programmer should think about the design of the solution before deciding to use them.

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. The iteration loop, on line 22, uses the local variable student, which does not exist outside the for loop.

```
1   # ------------------------------------------------------------
2   # Global variables
3   # ------------------------------------------------------------
4   # A global data structure
5   student_table = [["Student A", 11],
6                    ["Student B", 22],
7                    ["Student C", 33]]
8   user_name = ""                  # Only used in main program
9
10  # ------------------------------------------------------------
11  # Subprograms
12  # ------------------------------------------------------------
13  def display_welcome (p_name):
14      # p_name is a parameter so is a local variable
15      the_message = "Welcome "     # Local variable
16
17      the_message = the_message + p_name
18      print(the_message)
19
20  def display_students ():
21      print("Using a global variable")
22      for student in student_table:
23          print(student)
24
25  def display_table (p_table):
26      print("Using a parameter")
27      for item in p_table:
28          print(item)
29  # ------------------------------------------------------------
30  # Main program
31  # ------------------------------------------------------------
32  user_name = input("What is your name? ")
33
34  # Global variable passed into a subprogram as an argument,
35  #    which takes the place of the parameters
36  display_welcome(user_name)
37
38  display_students()      # No arguments
39
40  # Global data structure passed into a subprogram as an
41  #    argument, which takes the place of the parameters
42  display_table(student_table)
```

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 to avoid accessing global variables directly from inside subprograms and – instead – to pass them into the subprogram as arguments.

Blocked code

Blocking of code segments is indicated by indentation and subprogram calls. These determine the scope and extent of variable creation.

Built-in subprogram

Length of an object

The len() function can be used to determine the length of many different data structures, including strings, lists and sets.

Example	Description

`man = "Fred Flinstone"`
`family = ["Fred", "Wilma",`
`"Pebbles"]`
`print(len(man), len(family))`
	This example demonstrates using the len() function on a string and a list.
	
	The output is:
	`14 3`

`kitchen = ["pot", "pan", "plate"]`
`bedroom = ["bed", "pillow",`
`"dresser"]`
`house = {"kitchen": ["pot",`
`"pan", "plate"],`
`"bedroom": ["bed",`
`"pillow", "dresser"]}`
`print(len(kitchen), len(house))`
	This example demonstrates using the len() function on a dictionary.
	
	The output is:
	`3 2`

`garden = {"shovel", "trowel"}` `print(len(garden))`	This example demonstrates using the len() function on a set. The output is: `2`
`attic = (5, 4, 3, 2, 1)` `print(len(attic))`	This example demonstrates using the len() function on a tuple. The output is: `5`

Keyboard input

The input() function is used to take input from the user via the keyboard. Remember, 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
`name = input("Enter your name: ")`	This example accepts a string input and stores it in the variable name.
`guess = int(input("Guess: ")`	This example accepts a string input, converts it to an integer, then stores it in the variable guess.
`height = float(input("Metres: ")`	This example accepts a string input, converts it to a real number, then stores it in the variable height.

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 is: `Hello world`
`score = 83` `print(score)`	The output is: `83`
`print("My score is", score)`	The output is: `My score is 83`

Commas versus concatenation

Commas delimit the parameter values passed to the print() function, which adds a space between each 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.

Libraries of subprograms

Importing library modules

The PLS specifies a limited set of library modules and a set of actual subprograms found in them that learners 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. <library name>.<item name>.

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 floating-point 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 greater than or equal to the original.
	
	The output is:
	`9`

`import math`
`num = 8.752`
`result = math.floor(num)`
`print(result)`
	The math.floor() subprogram returns nearest integer less than or equal to 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 constant given by the math module rather than hard coding the value.

Copy library module

Making copies of data structures such as lists, dictionaries, and sets can have unintended side effects. This is made worse where the data structures are nested in some way, such as a two-dimensional table of records. The best way to avoid any potential side effects is to understand the use of shallow copy and deep copy.

Shallow copy

It creates a new object but copies pointers (or references) to the original underlying items. This means that both the original and the shallow copy reference the same memory locations for those underlying items. If you change an item in the shallow copy, it affects the original because both refer to the same underlying data in memory.

Deep copy

It creates a completely new object and copies the entire data structure, including all items, recursively. This means that the new object has its own separate memory locations for all underlying items. Changes made in the deep copy do not affect the original, as they do not share any pointer to the same memory locations.

It is good practice to use deep copy to duplicate any data structures required. Then, carry out processes on the copy. In that way, the original data is always recoverable.

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

Example	Description

`import copy`
 	
`table = [[152, "Jones"],`
`[938, "Black"]]`
 	
`shallow = copy.copy(table)`
`deep = copy.deepcopy(table)`
`print(table)`
`print(shallow)`
`print(deep)`
	This example demonstrates the use of shallow and deep copy on a two-dimensional table of records.
	
	The output is:
	`[[152, 'Jones'], [938, 'Black']]`
	`[[152, 'Jones'], [938, 'Black']]`
	`[[152, 'Jones'], [938, 'Black']]`

`table[1] = [111, "White"]`
`shallow[0][0] = 999`
`deep[0][1] = "Purple"`
`print(table)`
`print(shallow)`
`print(deep)`
	This example demonstrates amending each of the three lists.
	
	When the original is amended, as in the record at index [1], the two copies are not affected.
	
	When the shallow copy is amended, as in the number at index [0][0] is changed to 999, the corresponding content in the original is also affected.
	
	When the deep copy is amended, as in changing the name at index [0][1] to Purple, the original is not affected.
	
	The output is:
	`[[999, 'Jones'], [111, 'White']]`
	`[[999, 'Jones'], [938, 'Black']]`
	`[[152, 'Purple'], [938, 'Black']]`

NumPy library module

This section is applicable to Unit 4 only.

NumPy is used for efficient processing of number-based problems. It provides support for large, multi-dimensional arrays and matrices, which have a data type of ndarray. Learners need only work in two dimensions. It also includes a range of mathematical subprograms to operate on the arrays and matrices. These are written specifically for high performance. NumPy is used for numeric data.

Working with data in a matrix requires a slight shift in thinking. The process, e.g. the procedural programming, is not the priority. The priority is thinking about the data first, the shape of the date, the arrangement of the data, and how to apply subprograms to refine, filter, change, and reshape the data to get to a solution.

Some useful tips are:

- visualise the result, before tackling the problem
- pay careful attention to the potential for side effects, such as accidently removing rows or columns from the original data
- use NumPy subprograms with a functional approach
- always favour use of a NumPy subprogram over built-in libraries or writing the code yourself
- apply subprograms one at a time
- investigate the result of applying a subprogram, to find out its shape, dimensions, and type.

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

Note:

- The pprint library module is used for demonstration purposes only. Learners are not required to use the pprint library module.
- The use of very short identifiers is to aid presentation. Learners are advised to use more meaning identifier names to aid understanding of the logic.

Example	Description

`import numpy`
 	
`arr = numpy.array([[1,2,3,4,5],`
`[6,7,8,9,10]])`
`pprint.pprint(arr)`
`print('Index [0, 1]:',`
`arr[0, 1])`
	This example demonstrates creating a ndarray and accessing items in it by indexing.
	
	Notice that the indexing [0, 1] is formatted different than would be expected in a list [0][1].
	
	The output is:
	`array([[ 1,  2,  3,  4,  5],`
	`[ 6,  7,  8,  9, 10]])`
	`Index [0, 1]:  2`

`print("Dims:", numpy.ndim(arr))`
`print("Shape:",`
`numpy.shape(arr))`
`print("Size:", numpy.size(arr))`
`print("Dim 0:",`
`numpy.size(arr, 0),`
`"elements")`
`print("Dim 1:",`
`numpy.size(arr, 1),`
`"elements")`
	This example demonstrates ways to find out information about a ndarray structure.
	
	The output is:
	`Dims: 2`
	`Shape: (2, 5)`
	`Size: 10`
	`Dim 0: 2 elements`
	`Dim 1: 5 elements`

`pprint.pprint(`
`numpy.reshape(arr,`
`(5, 2)))`
`pprint.pprint(`
`numpy.reshape(arr,`
`-1))`
`pprint.pprint(numpy.ravel(arr))`
	This example demonstrates how to reshape and flatten a ndarray.
	
	The output is:
	`array([[ 1,  2],`
	`[ 3,  4],`
	`[ 5,  6],`
	`[ 7,  8],`
	`[ 9, 10]])`
	`array([ 1,  2,  3,  4,  5,`
	`6,  7,  8,  9, 10])`
	`array([ 1,  2,  3,  4,  5,`
	`6,  7,  8,  9, 10])`

`arr = numpy.array(`
`[10, 15, 20, 25, 30])`
`result = numpy.where(`
`arr > 20, 1, 0)`
`pprint.pprint(result)`
	This example demonstrates how to use the replace subprogram to generate a matrix showing all items greater than 20. It uses the where subprogram to perform the test.
	
	The output is:
	`array([0, 0, 0, 1, 1])`

print ("Along first axis: ") a = numpy.array([[12, 15], [10, 1]]) pprint.pprint(a) arr1 = numpy.sort(a, axis = 0) print (arr1) print ("Along second axis: ") a = numpy.array([[15, 10], [12, 1]]) pprint.pprint(a) arr2 = numpy.sort(a, axis = 1) pprint.pprint(arr2)	This example demonstrates sorting the entire ndarray in the first dimension (row) and sorting each row by the second dimension. The results may not appear intuitive, but the behaviour is consistent. The output is: Along first axis: array([[12, 15], [10, 1]]) [[10 1] [12 15]] Along last axis: array([[15, 10], [12, 1]]) array([[10, 15], [1, 12]])
a = numpy.array([1, 2, 3, 4]) pprint.pprint(a) select = a % 2 == 0 pprint.pprint(select) even = numpy.array([select]) pprint.pprint(even) print(type(a), type(even)) b = numpy.ravel(even) pprint.pprint(b) c = a[numpy.ravel(even)] pprint.pprint(c)	This example demonstrates using an array of Boolean values to select particular items. Even is a ndarray. It must be ravelled before being used as a sequence of indexes. The output is: array([1, 2, 3, 4]) array([False, True, False, True]) array([[False, True, False, True]]) <class 'numpy.ndarray'> <class 'numpy.ndarray'> array([False, True, False, True]) array([2, 4])

`arr = numpy.array([1, 2, 3, 4,`
`5, 6, 7, 8,`
`9, 10])`
`print("Mean:",`
`numpy.average(arr))`
`weight = numpy.array([5, 4, 3,`
`5, 4, 3, 5,`
`4, 3, 1])`
`print("Weighted Mean:",`
`numpy.average(arr,`
`weights = weight))`
	This example demonstrates using one of the many statistical subprograms available in NumPy.
	
	The output is:
	`Mean: 5.5`
	`Weighted Mean: 4.972972972972973`

Pandas library module

This section is applicable to Unit 4 only.

Pandas is a powerful Python library used for data manipulation and analysis. It provides data structures and functions to efficiently handle and process large datasets. Think about a dataset as data in tabular form, like spreadsheets and database tables. Pandas Is suitable for processing mixed-type data, for example, strings, integers and real numbers.

Working with data in Pandas requires a slight shift in thinking. The process, e.g. the procedural programming, is not the priority. The priority is thinking about the data first, the shape of the date, the arrangement of the data, and how to apply subprograms to refine, filter, sort and merge data to get to a solution.

In Pandas, most operations and subprograms are applied to either:

- DataFrames, which are two-dimensional labelled data structures with columns of data which may be of different data types. Think of dataframes as a two-dimensional table with rows and columns.
- Series which are one -dimensional labelled arrays of values, all of the same data type. Think of series as a one-dimensional array representing a single column.

However, there are a few standalone subprograms in Pandas, such as:

- pandas.read_csv()
- pandas.to_csv().

Some useful tips are:

- forget the concept of a unique key, as this is not a relational database
- the identifiers/labels for the columns, like headers in a table, are called columns
- the identifiers/labels for the rows, from top to bottom, are called index, just like in a table. They are not a key for a record. The default for indexes is zero.
- visualise the result, before tackling the problem
- pay careful attention to the potential for side effects, such as accidently removing rows or columns from the original data
- try using Pandas subprograms with a functional approach
- always favour use of a Pandas subprogram over built-in libraries or writing the code yourself
- apply subprograms one at a time
- investigate the result of applying a subprogram, to find out its shape, dimensions, and type.

Note:

- The pprint library module is used for demonstration purposes only. Learners are not required to use the pprint library module.
- The use of identifier names of one or two letters is to aid presentation. Learners are advised to use more meaning identifier names to aid understanding of the logic.

Dataframes and Series

This table shows examples of using the Pandas library module subprograms, the Pandas DataFrame subprograms and the Pandas Series subprograms.

Example	Description

`import pandas`
 	
`column = ["ID", "Pass", "Balance"]`
`users = [["Lae104", 8580, 10.60],`
`["Quy112", 9028, 2.98],`
`["Cho103", 7333, 45.89],`
`["Waa119", 2004, 103.00],`
`["Waa119", 2004, 103.00],`
`["Gre107", 3054, 12.34],`
`["Rya113", 1177, 56.10]`
`]`
 	
`df = pandas.DataFrame(users,`
`columns = column)`
`pprint.pprint(df)`
`print(type(df))`
	This example shows constructing a dataframe from a table of records.
	
	The output is:
	`ID Pass Balance`
	`0 Lae104 8580 10.60`
	`1 Quy112 9028 2.98`
	`2 Cho103 7333 45.89`
	`3 Waa119 2004 103.00`
	`4 Waa119 2004 103.00`
	`5 Gre107 3054 12.34`
	`6 Rya113 1177 56.10`
	`<class`
	`'pandas.core.frame.DataFrame'>`

`ids = df["ID"]`
`pprint.pprint(ids)`
`print(type(ids))`
	This example shows retrieving a column of data. Notice that the result is a series, not a dataframe.
	
	The output is:
	`0 Lae104`
	`1 Quy112`
	`2 Cho103`
	`3 Waa119`
	`4 Waa119`
	`5 Gre107`
	`6 Rya113`
	`Name: ID, dtype: object <class`
	`'pandas.core.series.Series'>`

`nodups = df.drop_duplicates(`
`  inplace = False)`
`pprint.pprint(nodups)`
	This example shows dropping the duplicate row for Waa119. Setting the inplace keyword to false preserves the original dataframe. The index values from the original dataframe are retained.
	
	The output is:
	`  ID  Pass  Balance`
	`0  Lae104  8580  10.60`
	`1  Quy112  9028  2.98`
	`2  Cho103  7333  45.89`
	`3  Waa119  2004  103.00`
	`5  Gre107  3054  12.34`
	`6  Rya113  1177  56.10`

`bal = nodups["Balance"]`
`sub = bal.iloc[1:-1]`
`pprint.pprint(sub)`
`total = sub.sum()`
`print("Total: ", total)`
	This example shows using one of the series subprograms to find the total of the balance column. The slice starting at an index of one skips the headers.
	
	The output is:
	`1  2.98`
	`2  45.89`
	`3  103.00`
	`5  12.34`
	`Name: Balance, dtype: float64`
	`Total:  164.21`

`max_bal = nodups["Balance"].max()`
`max_rows = nodups[nodups["Balance"]`
`  == max_bal]`
`pprint.pprint(max_rows)`
`print(type(max_rows))`
	This example shows finding the row or rows with the highest balance.
	
	The output is:
	`  ID  Pass  Balance`
	`3  Waa119  2004  103.0`
	`<class`
	`  'pandas.core.frame.DataFrame'>`

owing = nodups[nodups["Balance"] > 40.00] pprint.pprint(owing)	This example shows filtering the dataframe to find all the rows with a balance greater than 40.00. The output is: ID Pass Balance 2 Cho103 7333 45.89 3 Waa119 2004 103.00 6 Rya113 1177 56.10
snodups = nodups.sort_values(by = 'ID', ascending = False) pprint.pprint(snodups)	This example shows sorting the dataframe by id. The output is: ID Pass Balance 3 Waa119 2004 103.00 6 Rya113 1177 56.10 1 Quy112 9028 2.98 0 Lae104 8580 10.60 5 Gre107 3054 12.34 2 Cho103 7333 45.89
snodups.to_csv("NoDups.txt", sep = ",", float_format = "%.2f", index = False, header = False)	This example shows saving a dataframe to a file. The content of the file is: Waa119,2004,103.00 Rya113,1177,56.10 Quy112,9028,2.98 Lae104,8580,10.60 Gre107,3054,12.34 Cho103,7333,45.89

Date and time

Recall from the PLS, that time and dates are universal coordinated time (UTC), which is time zone independent.

Also, when formatting is required the ISO 8601 standard is supported. Some examples of using the datetime library and formatting of dates appear in the **Date and time** section.

Datetime library module

It is good practice to always manipulate and store datetimes as UTC, using as much precision as required by the context, which aids internationalisation. Unfortunately, most library subprograms that get the current time use the local time from the operating system. This must be expressed as the equivalent UTC. However, there are standard approaches to dealing with this.

When storing datetimes to persistent storage, e.g. files, they need to be converted to integers.

This table shows examples of using the datetime library to manipulate datetime objects, adjusting for local time, storing datetime as integers, and formatting datetime objects.

Example	Description

`from datetime import datetime, timezone`
`utc_now = datetime.now(timezone.utc)`
`print("Raw: ", utc_now)`
`print(f"Formatted: {utc_now.date()} {utc_now.time()}")`
`print(f"No part seconds: {utc_now.date()} {utc_now.strftime('%H:%M:%S')}")`
	This example shows retrieving the current time in UTC, adjusted for local time and displaying the result in different formats.
	
	The output is:
	`Raw: 2026-02-12 15:32:45.950631+00:00`
	`Formatted: 2026-02-12 15:32:45.950631`
	`No part seconds: 2026-02-12 15:32:45`

`timestamp = utc_now.timestamp()`
`print("As a timestamp:", timestamp)`
`timestamp_int = int(timestamp)`
`print("As an integer to store:", timestamp_int)`
	This example shows how to convert a UTC to a timestamp, display the timestamp and converting the timestamp to an integer which can be saved to a file.
	
	The output is:
	`As a timestamp: 1770910365.950631`
	`As an integer to store: 1770910365`

`utc_time =`
`  datetime.fromtimestamp(`
`  timestamp_int,`
`  timezone.utc)`
`print(f"Integer to UTC datetime`
`  object: {utc_time.date()}`
`  {utc_time.strftime(`
`  '%H:%M:%S')}")`
	This example shows converting a timestamp integer back to a UTC datetime object.
	
	The output is:
	`Integer to UTC datetime object:`
	`  2026-02-12 15:41:48`

`iso_string = "2026-10-21 15:38:21"`
`naive_datetime =`
`  datetime.strptime(iso_string,`
`  "%Y-%m-%d %H:%M:%S")`
`utc_datetime =`
`  naive_datetime.replace(`
`  tzinfo=timezone.utc)`
`print(f"Without part seconds:`
`  {utc_datetime.date()}`
`  {utc_datetime.strftime(`
`  '%H:%M:%S')}")`
	This example shows converting an ISO 8601 to a local UTC datetime object.
	
	The output is:
	`Without part seconds:`
	`  2026-10-21 15:38:21`

Time library module

The subprograms in this library are used to suspend execution of a program for a time period and for timing the execution of programs, subprograms or blocks of code.

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

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

Example	Description

`import time`
`print("Sleeping ...")`
`time.sleep(5)`
`print("Awake")`
	This example demonstrates suspending a program for five seconds.
	
	The output is:
	`Sleeping ...`
	`Awake`

`import time`
`def do_sleep (ptime):`
`  print("Sleeping ...")`
`  time.sleep(ptime)`
`  print("Awake")`
 	
`start = time.time()`
`do_sleep(5)`
`end = time.time()`
`print("Start:", start)`
`print("End:", end)`
`difference = end - start`
`print("Diff:", difference)`
	This example demonstrates how to determine the execution time of a block of code, the do_sleep() subprogram in this case.
	
	The output is:
	`Sleeping ...`
	`Awake`
	`Start: 1770915048.3525927`
	`End: 1770915053.3531816`
	`Diff: 5.000588893890381`

Turtle graphics library module

This section is applicable to Unit 2 only.

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.

Clean code

Consistency

White space

Good use of white space helps make programs easier to read. Learners should be encouraged to use white space to separate code blocks.

Double spacing code is not helpful and should be avoided.

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 horizontal scrolling. See below about fixed line length.

Python syntax allows long lines to be broken inside round brackets (), square brackets [], and curly 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 good practice to add an extra set of round brackets () in order to break long lines.

Here is an example of a line that has been broken around logical operators and between the outermost round brackets. Notice that extra round brackets, (), have been included around the highest-level if statement, to allow this breaking to happen.

```
if ((choice != "Q") and (choice != "A") and
        (choice != "B") and (choice != "C") and
        (choice != "D") and (choice != "E")):
    print("Hello World!")
```

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

```
record = [id_num, last_name, first_name,
          birth_month, birth_year]
```

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 at a location that does not support breaking between brackets.

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

```
record = tree_names[index] + "," + \
         str(tree_counts[index]) + "," + \
         str(tree_score[index])
```

Fixed line length

One good practice is to limit the length of a line to 80 characters. Editors track the column number and display it.

Readability

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 the Unit 2 and Unit 4 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 learners avoid unintended use of nested subprogram definitions and using an excessive number of global variables.

Identifiers

Learners 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 snake case. This is where the identifiers are all in lowercase letters. Individual words in an identifier are separated by underscore (_).

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

count	**big_list**	**up_counter**	**count_animal_types**	**is_valid_customer()**
name	**primary_key**	**last_name**	**student_first_name**	**find_student_name()**
key	**discount_code**	**blue_green**	**db_foreign_key**	**show_keys()**

In common with other programming languages, identifiers for constants use all uppercase letters.

This table shows examples of using uppercase letters for constants.

TAX
MAX_ROWS

In common with other object-oriented programming languages, identifiers that use Pascal case, where each word starts with an uppercase letter, are reserved for class names.

This table shows examples of using Pascal case for class names.

MY_CLASS
node = MY_CLASS("Chemistry")

Snake case, with Pascal case reserved for class names, is the method used when setting the Unit 2 and Unit 4 assessment tasks. However, it is not obligatory for learners to use these methods.

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.

Learners 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 excessive, as are comments that simply duplicate the code.

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

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.

Additional programming paradigms

This section is applicable to Unit 4 only.

Object-oriented paradigm

The object-oriented programming paradigm (OOP) is a way of designing and organising code. It involves creating virtual objects that represent real-world entities. Each object has its own properties (attributes) and behaviours (methods). Objects can interact with each other in an organised and structured way.

Working with objects requires a slight shift in thinking. The process, e.g. the procedural programming, is not the priority. The priority is thinking about how the objects interact or communicate with other objects to cause actions.

In OOP:

- the attributes of a class can be thought of as variables
- attributes can be private, protected or public
- the methods of a class can be thought of as subprograms
- methods can be private or public
- the collection of public methods makes up an API for the class

Some useful tips are:

- understand the terminology associated with OOP, such as method, attribute, instance, encapsulation, inheritance, aggregation, composition, and polymorphism
- adhere to the principle of encapsulation by never exposing attributes
- adhere to the principle of encapsulation by exposing the minimum number of methods
- favour composition (has a) over inheritance (is a) to promote reusability, easier maintenance, easier testing, and a reduced potential for side effects
- ensure a class has a single well-defined responsibility, rather than multiple, unrelated tasks. A class is not a library of subprograms
- define a clear and consistent set of methods (API) for each class.

Good practice in light of some Python behaviours:

- Python does not enforce encapsulation, so always write code in a way that protects attributes, e.g. write getter and setter methods.
    - Use the convention of a single underscore prefix for attributes to indicate a protected status, e.g. _name_.

    - Use the convention of a double underscore prefix for attributes to indicate a private status, e.g. __name__.
    - Use the convention of omitting underscores for attributes to indicate a public status.
- Python supports multiple inheritance, i.e. inheritance from more than one class. Avoid using this as it can cause confusion with trying to identify which methods are to be activated.
- Python supports instance variables. Instance variables are defined inside the __init__ constructor and are accessed through an instance of the class. Always use instance variables for attributes.
- Python supports class variables. Class variables are defined outside all methods, at the highest level of scope. Every object of a class shares a single copy of the class variables. Use class variables with extreme caution, or avoid altogether, as they can cause unpredictable behaviours and side effects.

The presentation of the OOP section is different to the other sections of this document. Rather than examples of individual subprograms, the examples are presented as part of a larger context. The context is a library of books. The program presented here does not solve the whole problem, but only serves as an example of different aspects of OOP.

The program demonstrates features of OOP, using Python. The features are identified in the block comments. The output is from the fully-functional program.

Output

```
Title: Pride and Prejudice, Author: Jane Austen, Publication Year:
1813
Genre: Romance

Title: A Brief History of Time, Author: Stephen Hawking, Publication
Year: 1988
Category: Physics

Describe books:
Pride and Prejudice by Jane Austen
Title: A Brief History of Time, Author: Stephen Hawking, Publication
Year: 1988
Category: Physics

Book reviews:
Book: Pride and Prejudice
Reviewer: John Doe
Review: A great romance novel!
Book: A Brief History of Time
Reviewer: Sue Felts
Review: A fascinating book on astrophysics.
```

Program

```
# ------------------------------------------------------------------------
# Node: Represent a Book
#
# Encapsulation: Demonstrates use of private attributes
# ------------------------------------------------------------------------
class Book:
    def __init__(self, title, author, publication_year):
        self.__title__ = title
        self.__author__ = author
        self.__publication_year__ = publication_year

    def display_info(self):
        print(f"Title: {self.__title__}, Author: {self.__author__}, "
              f"Publication Year: {self.__publication_year__}")

    # ------------------------------------------------------------------------
    # Encapsulation:  Demonstrates use of getter method to protect
    #                 private attributes
    # ------------------------------------------------------------------------
    def get_title(self):
        return(self.__title__)

    # ------------------------------------------------------------------------
    # Polymorphism: Demonstrates a change of behaviour based on an input
    #               parameter
    # ------------------------------------------------------------------------
    def describe_book(self, detail_level):
        if detail_level == "brief":
            print(f"{self.__title__} by {self.__author__}")
        elif detail_level == "full":
            self.display_info()
        else:
            print("Invalid detail level. Please use 'brief' or 'full'.")

# ------------------------------------------------------------------------
# Inheritance: FictionBook inherits from Book
# Polymorphism: FictionBook overrides the display method of Book
# ------------------------------------------------------------------------
class FictionBook(Book):
    def __init__(self, title, author, publication_year, genre):
        super().__init__(title, author, publication_year)
        self.__genre__ = genre

    def display_info(self):
        super().display_info()
        print(f"Genre: {self.__genre__}")

```

```python
# ------------------------------------------------------------------------------
# Inheritance: NonFictionBook inherits from Book
# Polymorphism: NonFictionBook overrides the display method of Book
# ------------------------------------------------------------------------------
class NonFictionBook(Book):
    def __init__(self, title, author, publication_year, category):
        super().__init__(title, author, publication_year)
        self.__category__ = category

    def display_info(self):
        super().display_info()
        print(f"Category: {self.__category__}")

# ------------------------------------------------------------------------------
# Composition: Library has a list of Books.  Library is the container.  Book
#              is being contained.  The list of books cannot exist outside
#              the library.
# ------------------------------------------------------------------------------
class Library:
    def __init__(self):
        self.__books__ = []

    def add_book(self, book):
        self.__books__.append(book)

    def display_books(self):
        for book in self.__books__:
            book.display_info()
            print()

# ------------------------------------------------------------------------------
# Aggregation: BookReview has an attribute of book, which is a reference
#              (pointer) to an instance of a Book.  The Book object can
#              exist outside the BookReview object.  The Book object
#              is not owned by the BookReview object.
# ------------------------------------------------------------------------------
class BookReview:
    def __init__(self, book, reviewer, review):
        self.__book__ = book
        self.__reviewer__ = reviewer
        self.__review__ = review

    def display_review(self):
        print(f"Book: {self.__book__.get_title()}")
        print(f"Reviewer: {self.__reviewer__}")
        print(f"Review: {self.__review__}")

```

```
# ------------------------------------------------------------------------
# Main program
# ------------------------------------------------------------------------
library = Library()

book1 = FictionBook("Pride and Prejudice", "Jane Austen", 1813, "Romance")
book2 = NonFictionBook("A Brief History of Time", "Stephen Hawking",
                       1988, "Physics")

# ------------------------------------------------------------------------
# Polymorphism: Different book types can be added to the library
# ------------------------------------------------------------------------
library.add_book(book1)
library.add_book(book2)

# ------------------------------------------------------------------------
# Polymorphism: Output of display_info, overridden in each of FictionBook and
#               NonFictionBook is different.
# ------------------------------------------------------------------------
library.display_books()

# ------------------------------------------------------------------------
# Polymorphism : Demonstrate a type of polymorphism implemented by changing
#                behaviour based on input parameter
# ------------------------------------------------------------------------
print("\nDescribe books:")
book1.describe_book("brief")
book2.describe_book("full")

# ------------------------------------------------------------------------
# Aggregation: Create some book reviews, using a reference (pointer) to
#              existing books
# ------------------------------------------------------------------------
review1 = BookReview(book1, "John Doe", "A great romance novel!")
review2 = BookReview(book2, "Sue Felts", "A fascinating book on astrophysics.")

print("\nBook reviews:")
review1.display_review()
review2.display_review()

```

Functional paradigm

This section is applicable to Unit 4 only.

Programming in a functional paradigm requires a slight shift in thinking. Just like with NumPy, the process, e.g. the procedural programming, is not the priority. The priority is thinking about the data first, the shape of the data, the arrangement of the data, and how to apply subprograms to refine, filter, change, and reshape the data to get to a solution.

Some useful tips are:

- data is never changed in place, instead, new data structures are created and passed along the chain of operations
- focus on pure functions, which have no side effects, i.e. they do not modify anything outside themselves
- the output of a function depends entirely and only on its inputs and never a state, variable, etc. outside itself
- visualise the result, before tackling the problem
- apply subprograms one at a time
- investigate the result of applying a subprogram, to find out its shape, dimensions, and type.

This table shows examples of using functional programming techniques.

Note:

- The pprint library module is used for demonstration purposes only. Learners are not required to use the pprint library module.
- The use of very short identifiers is to aid presentation. Learners are advised to use more meaning identifier names to aid understanding of the logic.

Example	Description

`nums = [1, 2, 3]`
`def triple (pval):`
`  return (pval ** 3)`
 	
`out_map = map(triple, nums)`
`print(out_map)`
`print(type(out_map))`
 	
`out_list = list(out_map)`
`print(out_list)`
`print(type(out_list))`
	This example demonstrates the use of map to apply a user-defined subprogram to each element of a list.
	
	Note that the output of map() is a map object, which is converted to a list using list().
	
	The output is:
	`<map object at 0x0000024FD75B3C00>`
	`<class 'map'>`
	`[1, 8, 27]`
	`<class 'list'>`

`songs = [`
`  {"title": "Song 1",`
`  "artist": "AAAAA",`
`  "duration": 3.53},`
`  {"title": "Song 2",`
`  "artist": "BBBBB",`
`  "duration": 4.38},`
`  {"title": "Song 3",`
`  "artist": "CCCCC",`
`  "duration": 3.56},`
`  {"title": "Song 4",`
`  "artist": "DDDDD",`
`  "duration": 3.53},`
`  {"title": "Song 5",`
`  "artist": "EEEEE",`
`  "duration": 4.55}`
`]`
`get_title = lambda song:`
`  song["title"]`
`titles = list(map(get_title,`
`  songs))`
`print(titles)`
	The data structure used for these examples is a playlist, represented as a dictionary.
	
	This example demonstrates creating a small anonymous function to extract the title field from each parameter (song) passed to it. A reference to the anonymous function is assigned to the get_title identifier.
	
	The map function is then used to apply the small anonymous function to each member of the songs data structure.
	
	The result from map is a map object. It is converted to a list using list().
	
	The output is:
	`['Song 1', 'Song 2', 'Song 3',`
	`  'Song 4', 'Song 5']`

`import math`
`total_ceiled_duration =`
`  sum(map(math.ceil,`
`  (song["duration"]`
`  for song in songs)))`
`print(total_ceiled_duration)`
	This example demonstrates using list comprehension (see **Python idioms** section) to extract all the durations, using map to apply math.ceil() to each in turn, and them sum them up.
	
	The output is:
	`22`

`long_songs = list(filter(`
`  lambda song:`
`  song["duration"] > 4.00,`
`  songs))`
`print(long_songs)`
	This example demonstrates how to filter data on a condition. In this case, find all the songs with a duration greater than 4.00.
	
	The output is:

`from functools import reduce`
`total_duration = reduce(lambda`
`  x, song: x +`
`  song["duration"], songs, 0)`
`print(total_duration)`
	This example demonstrates the use of a lambda function, which takes two arguments (x, song). It returns the sum of x and the duration of the song. The reduce ensures the function is applied cumulatively to each item in the list.
	
	The output is:
	`19.55`

`import pprint`
`both = list(zip([song["artist"]`
`  for song in songs],`
`  [song["title"]`
`  for song in songs]))`
`pprint.pprint(both)`
	This example demonstrates the use of zip to extract elements from multiple iterables, combine them into tuples, and reshape the result into a format that can be worked with. In this case, the artist column and the title column are extracted, combined into tuples by corresponding row, then converted to a list.
	
	The output is:
	`[('AAAAA', 'Song 1'),`
	`('BBBBB', 'Song 2'),`
	`('CCCCC', 'Song 3'),`
	`('DDDDD', 'Song 4'),`
	`('EEEEE', 'Song 5')]`

def song_generator(songs):
    for song in songs:
        yield song["title"]
 	
generator =
song_generator(songs)
print(next(generator))
print(next(generator))
print(next(generator))
print(next(generator))
print(next(generator))
	This example demonstrates creation of a generator function that suspends on each iteration, and calls to it.
	
	Note, this is not the same as calling next(generator)) in a for loop. Between each call, any amount of other operations could take place.
	
	The output is:
	Song 1
	Song 2
	Song 3
	Song 4
	Song 5

Idioms

In programming languages, idioms are mechanisms that:

- express common operations concisely
- are widely used and recognised
- reflect the language syntax and design philosophy.

Think of idioms as the text-speak of programming languages. Once you know it, it's all fine. Getting there can be challenging. However, learners will see these in tutorials and examples. Learners don't have to use idioms. There is always another way to express the same operation. It may be a bit longer, but could be much easier to read and understand.

One such Pythonic idiom is lambda functions, which were covered in the **functional programming** section.

This table shows examples of using Python idioms.

Note:

- The pprint library module is used for demonstration purposes only. Learners are not required to use the pprint library module.
- The use of very short identifiers is to aid presentation. Learners are advised to use more meaning identifier names to aid understanding of the logic.

Example	Description

`numbers = [1, 2, 3, 4, 5]`
`doubles = [x * 2 for x in numbers]`
`pprint.pprint(doubles)`
`print(type(doubles))`
	List comprehension is a concise way to create a new list by performing an operation on each element of an existing iterable object.
	
	The output is:
	`[2, 4, 6, 8, 10]`
	`<class 'list'>`

keys = ['a', 'b', 'c'] values = [1, 2, 3] dict = {key: value for key, value in zip(keys, values)} pprint.pprint(dict) print(type(dict))	Dictionary comprehension is a concise way to create a new dictionary by performing an operation on each element of an existing iterable object. Note: zip() has been used to create a single object for the value in the key:value pair. The output is {'a': 1, 'b': 2, 'c': 3} <class 'dict'>
dict = {} for key, value in zip(keys, values): dict[key] = value person = {'name': 'John', 'age': 30} print(person.get('name')) print(person.get('city', 'Not found')) print('name' in person) print('city' in person)	This example demonstrates using dictionary idioms. The .get() subprogram returns the value for the key, if it exists. Otherwise, it returns None. The in operator returns true if the key exists and false if it does not exist. The output is John Not found True False
age = 25 status = "adult" if age >= 18 else "minor" print(status)	This example demonstrates the use of conditional expression idioms. X, if (condition), else Y. The output is adult
nums = [10, 11, 12, 13, 14] odds = [x for x in nums if (x % 2) != 0] print(odds)	This is another example that demonstrates the use of conditional expression idioms. This time, the conditional is applied to every item in the list of numbers. The output is [11, 13]
a, b = (1, 2) print(a) print(b)	This example demonstrates the unpacking idiom, on both a tuple and a list.

`c, d = [10, 20]`
`print(c)`
`print(d)`
	The output is
	`1`
	`2`
	
	`10`
	`20`

Functionalities not in the PLS

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

Flow control constructs

Although some languages use flow control constructs such as goto, break, continue, pass, or exit, these are not supported in the PLSs. It's tempting to use these to get out of iteration, selection or subprograms.

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 programmers believe they need to use them, then they are advised to reconsider the design of their code.

Break

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

```
2   user_num = 0
3   # Using an infinite loop and a break statement
4   while (True):
5       user_num = int(input("Enter a number: "))
6       if (user_num == 0):
7           break
8       else:
9           print("The number is: ", user_num)
```

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.

```
11  user_num = 0
12  # Using a condition-controlled loop
13  user_num = int(input("Enter a number: "))
14  while (user_num != 0):
15      print("The number is: ", user_num)
16      user_num = int(input("Enter a number: "))
```

Another advantage of restricting the use of the break statement is transferability to other programming languages. Using the latter 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  user_num = int(input("Enter a number"))
26  while ((user_num >= 0) and (user_num <= 10)):
27      if (user_num > 5):
28          print("Upper Half")
29      elif (user_num == 0):
30          exit()
31      else:
32          print("Lower Half")
33      user_num = 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  user_num = int(input("Enter a number"))
39  while ((user_num > 0) and (user_num <= 10)):
40      if (user_num > 5):
41          print("Upper Half")
42      else:
43          print("Lower Half")
44      user_num = int(input("Enter a number"))
45  print("Bye")
```

More than working code

There is more to producing good programming solutions, than functionality. When creating solutions, programmers need to also consider:

- Decomposition into relevant parts
- Appropriate and efficient data structures
- Appropriate and efficient flow control constructs
- Separation of concerns, where one subprogram or block of code does one and only one job
- Minimisation of side effects, often caused by using global variables and multiple exits from blocks of code or subprograms
- Scope isolation
- Fitness for purpose
- Consistency of presentation
- Readability
- Robustness (handling all inputs and error conditions in an informative way)

Hints and tips

- Fully bracket relational and Boolean expressions to aid readability and ensure order of evaluation
- Include spaces before and after brackets to aid readability. The syntax of Python is not affected, but it can make understanding the code easier.
- Design code to iterate over data only once, rather than multiple iterations over the same data.
- Choose a storage type for data that matches the processing to be carried out on it. Process data in its most native form.
- Use the UTC time format for all internal storage and processing. Convert it to a human-readable form, which could be strings or integers, only when it is being output.
- Only format data, particularly numeric data, when it is going to be sent out of the program, as displayed or saved to a file.
- Use only programming constructs that support computational thinking in all languages. Simple, straightforward code is easier to read and easier to debug.
- Clean up resources, especially when an error occurs. These could include half-completed output files and closing database connections.
- Embrace the fail early design approach. This means to validate inputs and report errors quickly. Don't process invalid data.

For information about Pearson Qualifications, including Pearson Edexcel and BTEC qualifications, visit qualifications.pearson.com

Pearson Education Limited. Registered in England and Wales No. 872828
Registered Office: 80 Strand, London WC2R 0RL

VAT Reg No GB 278 537121

Cover image: Shutterstock / archy13