Computing

This quick reference is for Python 3.

Quick Reference

Certification Test Goals

This module sets out essential concepts and skills relating to the ability to use computational thinking and coding to create simple computer programs.

Successful candidates will be able to:

- Understand key concepts relating to computing and the typical activities involved in creating a program.
- Understand and use computational thinking techniques like problem decomposition, pattern recognition, abstraction and algorithms to analyse a problem and develop solutions.
- Write, test and modify algorithms for a program using flowcharts and pseudocode.
- Understand key principles and terms associated with coding and the importance of well-structured and documented code.
- Understand and use programming constructs like variables, data types, and logic in a program.
- Improve efficiency and functionality by using iteration, conditional statements, procedures and functions, as well as events and commands in a program.
- Test and debug a program and ensure it meets requirements before release.

1 Computing Terms

1.1 Key Concepts

1.1.1 Define the term computing.

- Performing calculations or processing data, especially using computer systems.

1.1.2 Define the term computational thinking.

- The process of analysing problems and challenges and identifying possible solutions to solve them.

1.1.3 Define the term program.

- Sets of sequenced steps, known as an algorithms, expressed in a form that can be understood and executed by a computer.

1.1.4 Define the term code.

- The text of a computer program, written as a series of instructions for a computer to execute.

1.1.4 Distinguish between source code, machine code.

- Source code is code written by a programmer that humans can read and understand.
- Machine code is a series of 1's and 0's created by a computer from source code that can be understood by the computers' electronic circuits.

1.1.5 Understand the terms program description and specification.

- A program description explains what a program is designed to do and how it will work.
- A program specification is a set of requirements outlining what a program will do.

1.1.6 Recognise typical activities in the creation of a program:

- Analysis - This involves clearly defining the problems that need to be solved.
- Design - This involves working out the algorithms to solve the problems.
- Programming - This involves writing the program by writing the algorithms in the chosen computer language.
- Testing - This involves checking that the program does what it was intended to do.
- Enhancement - This involves adding new features to improve performance or functionality, or to extend the program's use to different situations.

1.1.7 Understand the difference between a formal language and a natural language.

- A formal language is strictly structured, with exact and precise rules - examples include mathematics, chemical equations and computer programs.
- A natural language typically needs context to be clearly understood - examples include spoken languages like English, French or Chinese.

2 Computational Thinking Methods

2.1 Problem Analysis

2.1.1 Outline the typical methods used in computational thinking:

- Decomposition - The process of breaking down a complex problem into smaller, simpler, easier to understand problems.
- Pattern recognition - The process of finding patterns or repetition within complex problems or among smaller related problems.
- Abstraction - The process of extracting the most important or defining features from a problem or challenge.
- Algorithms - These are well-defined instructions in the form of steps, designed to solve problems or successfully complete tasks.

2.1.2 Use problem decomposition to break down data, processes, or a complex problem into smaller parts.

- For example, when organising a party venue, you can use problem decomposition to break the problem down into smaller parts such as selecting a suitable venue, a date and a budget and checking the availability and cost.

2.1.3 Identify patterns among small, decomposed problems.

- For example, when creating party invitations, you can identify a pattern in the use of the same design and text across all of the party invitations with a unique name and address inserted.

2.1.4 Use abstraction to filter out unnecessary details when analysing a problem.

- For example, when organising a party you can use abstraction to filter out unnecessary details such as the colour of the decorations so only important details remain such as the date, the venue, the guests, and the entertainment.

2.1.5 Understand how algorithms are used in computational thinking.

- The decomposition of a problem into smaller parts leads to the development of one or more algorithms.
- Algorithms are collections of steps that can be followed in order to solve a problem.
- For example, an algorithm for organising a party venue could include the following steps:
  - pick a venue
  - pick a date
  - pick a budget
  - check if the venue is available on the chosen date
  - check if the venue is in budget
  - pick a new venue or date if the venue isn't available or in budget
  - book the venue if it is available and in budget

2.2 Algorithms

2.2.1 Define the programming construct term sequence.

- A sequence is a number of simple instructions that need to be carried out one after the other.

2.2.1 Outline the purpose of sequencing when designing algorithms.

- Sequencing ensures that all the required actions to accomplish a task or set of tasks are performed in the right order.

2.2.2 Recognise possible methods for problem representation like:

- Flowcharts - A pictorial way of representing the steps in an algorithm with arrows showing the progression from one step to the next.
- Pseudocode - An informal way of representing an algorithm using written instructions in a natural language, for example English.

2.2.3 Recognise flowchart symbols like:

- Start/stop - ⬭
- Process - ▭
- Decision - ◇
- Input/output - ▱
- Connector - ●
- Arrow - →

2.2.4 Outline the sequence of operations represented by a flowchart.

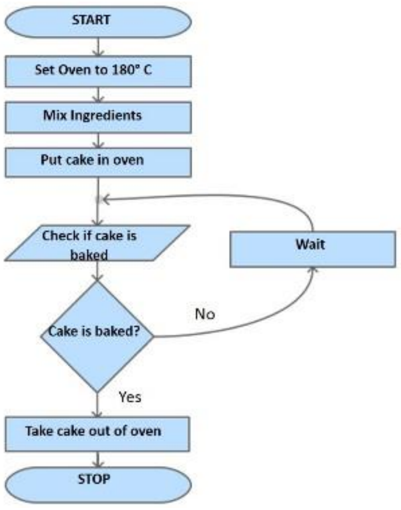
- Start/stop - In a flowchart an algorithm starts at the Start box and runs until it reaches a Stop box.
- Process - Contains simple instructions to do something. For example, the instruction could be 'put on warm clothes'.
- Decision - Allows for a choice in an algorithm. For example, if the answer to a question is 'yes' one route is taken and if the answer is 'no' the other route is taken.
- Input/output - Allows for an interaction with the world outside the computer. For example, an input could be a temperature reading from a sensor and an output could be turning off a heater.
- Connector - Connects two flow lines to show a jump from one point in the process flow to another. For example when answering a 'Yes/No' question.
- Arrow - Connects two shapes to show the direction of flow in an algorithm. For example, instructions in a sequence will have arrows between them.

2.2.4 Outline the sequence of operations represented by pseudocode.

- Pseudocode informally describes the sequence of operations in an algorithm using informal natural language with some of the structure and conventions of programming languages.

2.2.5 Write an accurate algorithm based on a description using a technique like: flowchart

- For example, the following flowchart describes an algorithm for baking a cake:



2.2.5 Write an accurate algorithm based on a description using a technique like: pseudocode

- For example, the following pseudocode describes an algorithm for baking a cake:

```
Set oven to 180⁰C
Mix ingredients
Put cake in oven
Check if cake is baked and if it isn't
    Wait
Take cake out of oven
STOP
```

2.2.6 Fix errors in an algorithm like:

- Missing program element - To fix this type of error in an algorithm, identify and then add the missing steps.
- Incorrect sequence - To fix this type of error in an algorithm, identify any instructions that are not in the correct sequence and then move them to the correct order.
- Incorrect decision outcome - To fix this type of error in an algorithm, identify any incorrect decision outcomes and then include a test to check if certain conditions are met.

3 Starting to Code

3.1 Getting Started

3.1.1 Describe the characteristics of well-structured and documented code like:

- Indentation - This involves indenting blocks of code to help people read and understand the code more easily.
- Appropriate comments - This involves describing the purpose of the code in natural language to help people understand what each section of code is doing.
- Descriptive naming - This involves giving meaningful names to items, functions and procedures to help people understand their purpose.

3.1.2 Use simple arithmetic operators to perform calculations in a program:

+	addition	`1+2`
		`3`

-	subtraction	`2-1`
		`1`

/	division	`4/2`
		`2`

*	multiplication	`1*2`
		`2`

3.1.3 Understand the precedence of operators and the order of evaluation in complex expressions.

- The precedence of operators determines the order in which operators are applied in complex expressions.
- The order of evaluation of operators in complex expressions is multiplication, division, addition, subtraction, comparison, and logical.

3.1.3 Understand how to use parenthesis to structure complex expressions.

- Parenthesis are used in complex expressions to indicate which part of the expression should be calculated first.
- The expression inside the parenthesis is calculated first.
- For example, 10-(6-4) = 8.

3.1.4 Understand the term parameter.

- A parameter is a special type of variable used in subroutines.
- You can add multiple parameters to a subroutine by separating them with commas.
- Parameters are the names used to describe the information passed into subroutines while the actual values are called arguments.

3.1.4 Outline the purpose of parameters in a program.

- Parameters are used in subroutines to influence what they do by passing information into them.

3.1.5 Define the programming construct term comment.

- A comment is a piece of text that explains what some part of the code does.

3.1.5 Outline the purpose of a comment in a program.

- Comments are used to help people understand what is happening within a program.

3.1.6 Use comments in a program.

- To add a comment to a program, you use the # symbol followed by the comment text.
- The comment is considered the text after the # until the end of the line.
- For example

```
# This is a comment
```

3.2 Variables and Data Types

3.2.1 Define the programming construct term variable.

- A variable is used to represent a piece of data.
- It is like a placeholder for an actual value.
- It can hold a value and retrieve it for use later.

3.2.1 Outline the purpose of a variable in a program.

- Variables are used so that data can be used several times in a program.

3.2.2 Define and initialise a variable.

- Defining a variable means defining what data type the variable will hold. Data types are used to determine how to store data in the computer's memory and what operations can be carried out on the data.
- Initialising a variable means assigning an initial value to a variable. This must be done before you can use the variable.
- For example, a variable named 'price' can be defined with the data type integer and assigned an initial value of 0 as follows:

```
price = int(0)
```

3.2.3 Assign a value to a variable.

- Assigning a value to a variable means specifying what value the variable will represent.
- You can update the value stored in a variable by assigning a new value to the variable.
- For example the variable 'price' can be assigned the value 7 as follows:

```
price = 7
```

3.2.4 Use appropriately named variables in a program for calculations, storing values.

- Variables should have easy to read, descriptive names that start with a letter and avoid reserved words.
- Some good examples include: `counter, weight, age, height`
- And some weak examples include: `x, y, my, eg_3, 2_var, var_1,`

3.2.5 Use data types in a program: string.

- A string data type is used for text, for example hello world.
- A string data type can be defined using the syntax

```
string1 = str("string")
```

- For example

```
yourName = str("Sam")
```

3.2.5 Use data types in a program: character

- The character data type is used in some programming languages but not in Python.
- Python uses the string data type instead of the character data type.

3.2.5 Use data types in a program: integer.

- An integer data type is used for whole numbers, for example 1 and 67.
- An integer data type can be defined using the syntax

```
integer1 = int(1)
```

- For example

```
yourAge = int(14)
```

3.2.5 Use data types in a program: float.

- A float data type is used for decimal numbers, for example 1.0 and 67.9.
- A float data type can be defined using the syntax

```
float1 = float(1.0)
```

- For example

```
yourHeight = float(160.5)
```

3.2.5 Use data types in a program: Boolean

- A Boolean data type is used for True or False values.
- A Boolean data type of True can be defined using the syntax

```
Boolean1 = True
```

- For example

```
yourHeight = True
```

- A Boolean data type of false can be defined using the syntax

```
Boolean1 = False
```

- For example

```
yourHeight = False
```

3.2.6 Use an aggregate data type in a program like: array.

- The array aggregate data type is used in some other programming languages but not in Python.
- Python uses the aggregate data type 'list' instead of 'array'.

3.2.6 Use an aggregate data type in a program like: list.

- A list aggregate data type is used to hold multiple items known as elements.
- Elements are referenced by numbers starting from zero.
- A list can be modified by adding items, removing items or changing items.
- The syntax is

```
List1 = [element0, element1]
```

- For example

```
myPets = ["cat", "dog"]
```

3.2.6 Use an aggregate data type in a program like: tuple.

- A tuple aggregate data type is like a list, except once created, it can't be modified, unless you assign a new variable to the tuple which then rewrites the tuple contents.
- The syntax is

```
Tuple1 = (element0, element1)
```

- For example

```
myPets = ("pig", "hen")
```

3.2.7 Use data input from a user in a program.

- You can request a user to input data while a program is running using the input statement.
- The syntax is

```
input()
```

- For example

```
name = input("your name is?")
```

3.2.8 Use data output to a screen in a program.

- You can output data to a screen in a program using the print statement.
- The syntax is

```
print()
```

- For example

```
print("Hello, world!")
```

4 Building using Code

4.1 Logic

4.1.1 Define the programming construct term logic test.

- A logic test is a Boolean expression that results in a Boolean value that is either True or False.

4.1.1 Outline the purpose of a logic test in a program.

- A logic test is used as a way to test if certain conditions exist in order to control what happens next in a program.

4.1.2 Recognise types of Boolean logic expressions to generate a true or false value like:

==	Equals
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
<>	Not equal to - Python uses != instead
=	Equals - Python uses == instead
AND	Combines two Boolean values and gives a result of True if both values are True, otherwise gives a result of False. Note in python this is written as lower case.
OR	Combines two Boolean values and gives a result of True if either one or both values are True, otherwise gives a result of False. Note in python this is written as lower case.
NOT	Converts a single Boolean value from True to False, or False to True. Note in python this is written as lower case.

4.1.3 Use Boolean logic expressions in a program.

==	`1 == 1`
	`True`

!=	`1 != 2`
	`True`

>	`1 > 2`
	`False`

<	`1 < 2`
	`True`

>=	`1 >= 2`
	`False`

<=	`1 <= 2`
	`True`

<>	Python uses != instead
=	Python uses == instead

AND	`True and True`
	`True`
	`True and False`
	`False`
	`False and True`
	`False`
	`False and False`
	`False`

OR	`True or True`
	`True`
	`True or False`
	`True`
	`False or True`
	`True`
	`False or False`
	`False`

NOT	`not True`
	`False`
	`not False`
	`True`

4.2 Iteration

4.2.1 Define the programming construct term loop.

- A loop is a piece of code that runs repeatedly under certain conditions – either a specified number of times or until a specified condition is met.

4.2.1 Outline the purpose and benefit of looping in a program.

- Looping, also referred to as iteration, saves programming time and results in shorter code by repeating steps.

4.2.2 Recognise types of loops used for iteration: for

- A for loop is a type of loop that executes a sequence of statements a specified number of times.
- A for loop is used when the number of loops needed is already known.

4.2.2 Recognise types of loops used for iteration: while.

- A while loop is a type of loop that tests a Boolean expression and executes a statement or group of statements as long as the Boolean expression is true.
- A while loop is used when the number of loops required depends on certain conditions being met.

4.2.2 Recognise types of loops used for iteration: repeat.

- A repeat loop is used in some other programming languages but not Python.
- Python uses the 'While' loop instead.

4.2.3 Use iteration (looping) in a program like: for.

- The for loop syntax is

```
for variable_name in range ():
    statement(s)
```

- For example:

```
for count in range(10):
      print("*")
```

4.2.3 Use iteration (looping) in a program like: while.

- The while loop syntax is

```
while expression:
   statement(s)
```

- For example:

```
number = 10
   while number > 1:
    print( number )
    number = number - 1
```

4.2.3 Use iteration (looping) in a program like: repeat.

- Python doesn't use the repeat loop.

4.2.4 Understand the term infinite loop.

- An infinite loop is a type of while loop where the logic test is always true, so the loop will repeat forever.
- An infinite loop can cause a program to fail or in some cases it can be used on purpose so that the program runs indefinitely.

4.2.5 Understand the term recursion.

- Recursion is the process of a subroutine dividing a problem into simpler parts and calling itself to solve those simpler parts.

4.3 Conditionality

4.3.1 Define the programming construct term conditional statement.

- A conditional statement is used to evaluate an expression as True or False. The outcome, a True or False value, determines what is done next in a program.

4.3.1 Outline the purpose of conditional statements in a program.

- A conditional statement is used in a program to determine what happens next in the program depending on the result of the evaluation of the expression.

4.3.2 Use IF…THEN...ELSE conditional statements in a program.

- An If conditional statement tests a Boolean expression. If the Boolean expression is True, it executes a statement.
- The syntax is

```
if expression:
    statement if True
```

- For example:

```
if 100 < 250:
    print ("Yes")
```

4.3.2 Use IF…THEN...ELSE conditional statements in a program.

- An Else conditional statement tests a Boolean expression and if the Boolean expression is true, it executes a statement and if the Boolean expression is false, it executes a different statement.
- The syntax is

```
if expression:
    statement if True
else:
    statement if False
```

- For example:

```
if 100 < 250:
    print ("Yes")
else:
    print ("No")
```

4.4 Procedures and Functions

4.4.1 Understand the term procedure.

- A Procedure is a subroutine that executes an action, but does not return a value.
- Python includes pre-defined procedures, for example print().
- Or you can create your own procedures known as user-defined or custom-made.

4.4.1 Outline the purpose of a procedure in a program.

- Procedures enable code to be reused, which reduces programming time and makes code shorter and easier to read.
- If changes are required they only have to be implemented once in the procedure code and the updates are applied automatically wherever the procedure is used in the program.

4.4.2 Write and name a procedure in a program.

- When you write and name a procedure you can use it anywhere in the program.
- To write and name a procedure, include its name and brackets – this is known as executing or calling a procedure.
- The syntax for writing, naming and calling a procedure is

```
def procedure_name():
    statement(s)

procedure_name()
```

- For example:

```
def line_vertical():
    for count in range (10):
        print("*")

line_vertical()
```

4.4.3 Understand the term function.

- A function is a subroutine that calculates a value for the program that contains it.
- Python includes pre-defined functions, for example, input().
- Or you can create your own functions known as user-defined or custom-made.

4.4.3 Outline the purpose of a function in a program.

- Functions enable code to be reused, which reduces programming time and makes code shorter and easier to read.
- If changes are required they only have to be implemented once in the function code and the updates are applied automatically wherever the function is used in the program.

4.4.4 Write and name a function in a program.

- When you write and name a function you can use it anywhere in the program.
- To write and name a function, include its name and brackets – this is known as executing or calling a function.
- The syntax for writing, naming and calling a function is

```
def function_name():
    statement(s)
    return statement
function_name()
```

- For example:

```
y = int(input("Enter value "))
answer= int(0)
def square(y):
    answer = y*y
    return answer
answer = square(y)
print(answer)
```

4.5 Events and Commands

4.5.1 Understand the term event.

- Events are actions that are triggered by an action such as a user pressing a key press, or a mouse click, or a button click. They can also be triggered by a timer.

4.5.1 Outline the purpose of an event in a program.

- Events are used to impact the flow of a program and to trigger something happening.

4.5.2 Use event handlers like: mouse click, keyboard input, button click, timer.

- An event handler is a piece of code designed to do something once an event has been triggered.
- Mouse click - a user clicking the mouse.
- Keyboard input - a user pressing a keyboard button.

- Button click - an on-screen button being activated.
- Timer - a predefined time triggering an event.

4.5.3 Use available generic libraries.

- A library is a useful collection of pre-defined procedures and functions that come with the programming language.
- Python refers to generic libraries as standard libraries.
- To use a Python standard library in a program it must be imported near the start of the program.
- You can import an entire library using the syntax

```
import library_name
```

- Or you can import specific functions from a library using the syntax

```
from library_name import
function_name
```

4.5.3 Use available generic libraries like: math.

- The math library contains lots of mathematical functions.
- The sqrt function calculates the square root of a number. For example:

```
import math
print (math.sqrt(4))
```

- The factorial function calculates the factorial of a number. For example:

```
from math import factorial
print (factorial(4))
```

- The pow function raises a number to a power. For example to raise 10 to power 3:

```
from math import pow
print (pow(10,3))
```

4.5.3 Use available generic libraries like: random.

- The random library contains functions for generating random numbers.
- The choice function picks one random item from a list. For example:

```
import random
fruit = [“apple”, “grapefruit”]
print(random.choice (fruit))
```

- The randit function picks a random integer from between two specified integers. For example:

```
from random import randit
print(“roll the dice 10 times”)
for i in range (0,10):
    print( randit(1,6))
```

4.5.3 Use available generic libraries like: time.

- The time library contains lots of functions related to time.
- The strftime function takes a 'time object' and converts it to a string. It contains many formatting options for time.
- The gmtime function gives a 'time object' for the current time of day.
- For example:

```
from time import strftime, gmtime
print (strftime(“%x”, gmtime()))
```

5 Test, Debug and Release

5.1 Run, Test and Debug

5.1.1 Understand the benefits of testing and debugging a program to resolve errors.

- Testing and debugging a program to resolve errors ensures that the program is working as intended before release.

5.1.2 Understand types of errors in a program like:

- Syntax error - This is an error due to a construct in the programming language being written incorrectly. Examples include incorrect syntax, omission of a required colon or bracket, misspelling a keyword, or the incorrect format used for numbers.
- Logic error - This is an error due to flaws in the logic where the program may operate correctly but doesn’t do what is required.

5.1.3 Run a program.

- Open a program.
- Click Run in the menu bar and click Run Module.
- Or press F5 on the keyboard.

5.1.4 Identify and fix a syntax error in a program like: incorrect spelling, missing punctuation.

- It is important to check your code for incorrect spelling and missing punctuation.
- In the following example, ‘print’ is spelt incorrectly and there are colons missing after the ‘if’ and ‘else’ statements.

```
myAge = int(14)


if(myAge > 16)
    prinr("You can go!")
else
    prinr("You are too young.")
```

5.1.5 Identify and fix a logic error in a program like: incorrect Boolean expression, incorrect data type.

- It is important to check your code for incorrect Boolean expressions and incorrect data types.
- In the following example, the Boolean expression is incorrect. The Boolean expression should be ‘greater than’, rather than ‘less than’. And the ‘myAge’ data type is incorrect. The data type should be ‘int’, rather than ‘str’.

```
myAge = str(14)
if(myAge < 16):
    print("You are old enough")
else:
    print("You are too young")
```

5.2 Release

5.2.1 Check your program against the requirements of the initial description.

- Before release, it is important to systematically test your program against the requirements of the initial description, documented in the specification document.

5.2.2 Describe the completed program, communicating purpose and value.

- When you are describing the completed program be aware of the audience and what is relevant to them.
- When communicating purpose explain what problem the program solves. It can also be useful to explain what problems were beyond the scope of the project.
- When communicating value highlight what is new and unique about the program and try to support your claims with facts and figures. For example, the program might save the end user time or it might save the client money.

5.2.3 Identify enhancements, improvements to the program that may meet additional, related needs.

- Examine how enhancements and improvements to the program such as, small additions could generalise the program and make it useful in a wider context to meet additional or related needs.
- For example, you might localise the program for different countries, adapt it for different systems or modify it for different uses.

For more information,
visit: **www.ecdl.org**