Notes and guidance: Python

The Python code is described below to help students prepare for their AQA GCSE Computer Science exam (8525/1). It is based on Python version 3 only.

We will use this consistent style of Python code in all assessment material. This will ensure that, with enough preparation, students will understand the syntax of the code used in assessments. Students do not have to use this style of code in their own work or written assessments, although they are free to do so. The only direction to students when answering questions or describing algorithms written in code is that their code is clear, consistent and unambiguous.

This resource may be updated as required and the latest version will always be available on our website. It is not confidential and can be freely shared with students.

General Syntax

- Code is shown in `this font`.
- `Exp` means any expression.
- `IntExp`, `RealExp`, `BoolExp`, `StringExp` and `ListExp` mean any expression which can be evaluated to an integer, real, Boolean (`False` or `True`), string or list respectively.

Indentation

Python uses indentation to indicate the range of statements controlled by iteration and selection statements (as well as declarations for subroutines and classes when used to implement records). Indentation will be shown with three spaces per indentation level, although if doing so makes lines too long for the page this may be reduced to two spaces. Questions will show indentation guides (vertical lines) within the answer space. Students should be encouraged to use these to explicitly show their indentation.

Comments

Single line comments	`# A comment`	
Multi-line comments	`# A comment` `# Another comment`	

String and character literals

String and character literals will be delimited using the " (double quote) character.

Type of variables

Since Python does not type variables, but only the value that a variable 'has', it is possible to use a variable to hold a string, then an integer, then a Boolean and so on. Students should be taught that, for the purposes of assessment, the type of the value **first** assigned to a variable will be taken to declare the type of that variable. For example, `totalCost = 2.45` would result in `totalCost` having a data type of Real. Within that program `totalCost` will then always contain real values.

Variables and constants

Variable names will be written in camel case eg `numberOfItemsSold`. Camel case is the practice of writing phrases without spaces or punctuation, indicating the separation of words with a single capitalised letter and the first word starting with either case.

Constant names will be written in upper case, using an underscore to indicate a break between words, eg `ACCELERATION_DUE_TO_GRAVITY`.

Although Python considers names such as `message` and `Message` to refer to distinct variables, questions will not include any variables whose names differ solely in case.

Questions will use meaningful variable names wherever possible, eg `quantityInStock`, `quantity` or `qty` rather than just `n` to hold the quantity of an item in stock. For layout and/or lack of context reasons#, this resource may not always follow this advice. This rule may not be followed for common idioms, eg using `i` as a loop index or for exam-related reasons.

Variable assignment	`Identifier = Exp`	`aNumber = 3`
		`anotherNumber = aNumber + 1`
		`theString = "Hello"`
		`message = "Invalid number"`
		`totalNumberOfItems = 0`

Constant assignment	`IDENTIFIER = Exp`	`CLASS_SIZE = 23` `PI = 3.141`

Arithmetic operations

Standard arithmetic operations	`+`
	`-`
	`*`
	`/`
		Used in the normal way with brackets to indicate precedence where needed. So, `a + b * c` would multiply `b` and `c` together and then add the result to `a`, whereas `(a + b) * c` would add `a` and `b` together and then multiply the result by `c`.
		
		Brackets may be used to indicate precedence even where not strictly necessary, eg testing for divisibility by 3 could be written as `(n % 3) == 0` rather than `n % 3 == 0`

Integer division	`IntExp // IntExp`	`9 // 5` evaluates to `1`
		`5 // 2` evaluates to `2`
		`8 // 4` evaluates to `2`

Modulus operator	`IntExp % IntExp`	`9 % 5` evaluates to `4`
		`5 % 2` evaluates to `1`
		`8 % 4` evaluates to `0`

Relational operators for types that can be clearly ordered (numbers, strings, characters)

Less than	`Exp < Exp`	`4 < 6`
		`"A" < "B"`
		`"adam" < "adele"`

Greater than	`Exp > Exp`	`4.1 > 4.0`
Equal to	`Exp == Exp`	`3 == 3`
Not equal to	`Exp != Exp`	`qty != 7`
Less than or equal to	`Exp <= Exp`	`3 <= 4` `4 <= 4`
Greater than or equal to	`Exp >= Exp`	`4 >= 3` `4.5 >= 4.5`

Boolean operations

Logical AND	`BoolExp and BoolExp`	`(3 == 3) and (3 <= 4)`
Logical OR	`BoolExp or BoolExp`	`(x < 1) or (x > 9)`
Logical NOT	`not BoolExp`	`not (a < b)`

Indefinite (condition controlled) iteration

<table>
<tr>
<td>WHILE (while the Boolean expression is <code>True</code>, repeat the statements). If the Boolean expression is <code>False</code> the first time the <code>while</code> statement is reached then the indented statements are never executed.</td>
<td><pre>while BoolExp:
    # indented statements here</pre></td>
<td><pre>a = 1
while a < 4:
    print(a)
    a = a + 1
# outputs 1, 2, 3</pre>
whereas
<pre>a = 5
while a < 4:
    print(a)
    a = a + 1
# does not output anything since
# a < 4 is false the first time the
# while is encountered</pre></td>
</tr>
<tr>
<td>Python does not have the equivalent of a REPEAT-UNTIL (repeat the statements until the Boolean expression is <code>True</code>) but this will be simulated using a while structure if required.</td>
<td><pre># statements here
while BoolExp:
    # copy of statements here (indented)</pre></td>
<td><pre>a = 1

print(a)
a = a + 1
carryOn = input("Continue? ")
while carryOn != "N":
    print(a)
    a = a + 1
    carryOn = input("Continue? ")
# outputs 1, … until "N" is entered</pre></td>
</tr>
</table>

Definite (count controlled) iteration

FOR
(repeat the statements the number of times indicated by the range function, each time giving the loop variable (`Identifier`) the value of the next value/number in the range).
	`for Identifier in range(IntExp):`
	`  # indented statements here`
	
	`# With only one IntExp inside the brackets`
	`# Identifier will first have the value 0, 1,`
	`# 2, all the way up (in steps of 1) to the`
	`# value of IntExp - 1`
	
	`for Identifier in range(IntExp1, IntExp2):`
	`  # indented statements here`
	
	`# With two IntExps inside the brackets`
	`# Identifier will first have the value IntExp1,`
	`# then IntExp1 + 1, IntExp1 + 2, all the way`
	`# up (still in steps of 1) to the value of`
	`# IntExp2 - 1`
	
	`for Identifier in range(IntExp1, IntExp2, IntExp3):`
	`  # indented statements here`
	
	`# With three IntExps inside the brackets`
	`# Identifier will first have the value IntExp1,`
	`# then IntExp1 + IntExp3, IntExp1 + 2 * IntExp3,`
	`# all the way up (or down if IntExp3 is negative)`
	`# to the value of IntExp2 - 1 (or IntExp2 + 1 if`
	`# IntExp3 is negative)`
		`for i in range(7):`
		`  print(i)`
		
		`# outputs 0, 1, 2, 3, 4, 5, 6`
		`# 7 values are output, the last`
		`# one being 6`
		
		`for i in range(1, 7):`
		`  print(i)`
		
		`# outputs 1, 2, 3, 4, 5, 6`
		
		`for i in range(1, 7, 2):`
		`  print(i)`
		
		`# outputs 1, 3, 5`
		
		`for i in range(7, 1, -2):`
		`  print(i)`
		
		`# outputs 7, 5, 3`

Definite (count controlled) iteration (continued)

FOR
(repeat the statements the number of times that there are characters in a string, each time giving `Identifier` the value of the next character in the string).
	`for Identifier in StringExp:`
	`  # indented statements here`
		`length = 0`
		`for char in message:`
		`  length = length + 1`
		`print(length)`
		`# calculate the number of`
		`# characters in message`
		`# and output it`
		
		`reversed = ""`
		`for char in message:`
		`  reversed = char + reversed`
		`print(reversed)`
		`# reversed is set to the`
		`# reverse of message and`
		`# output`
		`# eg if message == "Hello"`
		`# then reversed will become`
		`# "olleH"`

Selection

IF-THEN-ENDIF (execute the statements only if the Boolean expression is `True`: see *Python Boolean expressions* above).	`if BoolExp:`
	`  # indented statements here`
		`a = 1`
		`if (a % 2) == 0:`
		`  print("even")`

IF-THEN-ELSE-ENDIF (execute the statements following the THEN if the Boolean expression is `True`, otherwise execute the statements following the ELSE)/.	`if BoolExp:`
	`  # indented statements here`
	`else:`
	`  # indented statements here`
		`a = 1`
		`if (a % 2) == 0:`
		`  print("even")`
		`else:`
		`  print("odd")`

Selection (continued)

NESTED IF-THEN-ELSE ENDIF (use nested versions of the above to create more complex conditions).
 	
Note that IF statements can be nested inside the THEN part, the ELSE part or both.
	`if BoolExp:`
	`  # indented statements here`
	`else:`
	`  if BoolExp:`
	`  # indented statements here`
	`  else:`
	`  # indented statements here`
		`a = 1`
		`if (a % 4) == 0:`
		`  print("multiple of 4")`
		`else:`
		`  if (a % 4) == 1:`
		`  print("leaves a remainder of 1")`
		`  else:`
		`  if (a % 4) == 2:`
		`  print("leaves a remainder of 2")`
		`  else:`
		`  print("leaves a remainder of 3")`

IF-THEN-ELSE IF ENDIF (removes the need for multiple indentation levels).	`if BoolExp:`
	`  # indented statements here`
	`elif BoolExp:`
	`  # indented statements here`
	`  # possibly more elifs`
	`else:`
	`  # indented statements here`
		`a = 1`
		`if (a % 4) == 0:`
		`  print("multiple of 4")`
		`elif (a % 4) == 1:`
		`  print("leaves a remainder of 1")`
		`elif (a % 4) == 2:`
		`  print("leaves a remainder of 2")`
		`else:`
		`  print("leaves a remainder of 3")`

Arrays

Assignment	`Identifier = [Exp, … ,Exp]`	`primes = [2, 3, 5, 7, 11, 13]`

Accessing an element (indexing)	`Identifier[IntExp]`	`print(f"Only even prime is {primes[0]}")`
		
		`# prints "Only even prime is 2"`

Updating an element	`Identifier[IntExp] = Exp`	`primes[5] = 17`
		
		`# array is now [2, 3, 5, 7, 11, 17]`

Accessing an element in a two-dimensional array	`Identifier[IntExp][IntExp]`	`table = [[1, 2], [2, 4], [3, 6], [4, 8]]`
		
		`print(f"Row 4 column 2: {table[3][1]}")`
		
		`# prints "Row 4 column 2: 8" as the second`
		`# element (with index 1) of fourth element`
		`# (with index 3) in array is 8`
		
		`# Note that table[1][3] would give an error`
		`# since there is no fourth element of the`
		`# second element in table`

Updating an element in a two- dimensional array	`Identifier[IntExp][IntExp] = Exp`	`table[3][1] = 16`
		
		`# table is now`
		`# [[1, 2], [2, 4], [3, 6], [4, 16]]`

Arrays (continued)

Array length	`len(Identifier)`	`len(primes)`
		`# evaluates to 6 using example above`
		
		`len(table)`
		`# evaluates to 4 using example above`
		
		`len(table[0])`
		`# evaluates to 2 using example above`

FOR
(repeat the statements the number of times that there are elements in a list, each time giving Identifier the value of the next element in the list).
	`for Identifier in ListExp:`
	`  # indented statements here`
		`ages = [15, 27, 19, 18, 17]`
		`total = 0`
		`for age in ages:`
		`  total = total + age`
		`mean = total / len(ages)`
		`print(mean)`
		`# calculates the total of the ages`
		`# held in the array and the mean`
		`# age and then outputs the mean`

Records

Record declaration (exam questions will use this method to declare and use records).	`class RecordName:`
	`  def __init__(self, v1, v2, …):`
	`  self.field1 = v1`
	`  self.field2 = v2`
	`  …`
	
	`# __init__ is the constructor and`
	`# must have this name with two`
	`# underscores before init and two`
	`# after. The self parameter is`
	`# essential. The name of the class`
	`# will start with a capital letter.`
		`class Car:`
		`  def __init__(self, mk, md, rg, pr, ds):`
		`  self.make = mk`
		`  self.model = md`
		`  self.reg = rg`
		`  self.price = pr`
		`  self.noOfDoors = ds`
		
		`# each field in the record must be defined`
		`# by preceding it with self. which 'adds'`
		`# that field to the record when it is`
		`# created`

Variable instantiation	`varName = RecordName(v1, v2, …)`	`myCar = Car("Ford", "Focus", "DX17 GYT", 1399.99, 5)`

Assigning a value to a field in a record	`varName.field = Exp`	`myCar.model = "Fiesta"`
		
		`# The model field of the myCar`
		`# record is assigned the value`
		`# "Fiesta".`

Accessing values of fields within records	`varName.field`	`print(myCar.model)`
		
		`# Outputs the value stored in the`
		`# model field of the myCar record`

Subroutines

Note: subroutines that contain a `return` keyword followed by a value are functions. Those that do not contain a `return` keyword, or that contain one with no value after it, are procedures.

Subroutine definition	`def Identifier(parameters):`
	`  # indented statements here`
		`def showAdd(a, b):`
		`  result = a + b`
		`  print(result)`
		
		`def sayHi():`
		`  print("Hi")`
		
		`# Both of these subroutines are procedures`

Subroutine return value	`return Exp`	`def add(a, b):`
		`  result = a + b`
		`  return result`
		
		`# This subroutine is a function`

Calling subroutines	`# Subroutines without a return value`
	
	`Identifier(parameters)`
	
	`# Subroutines with a return value`
	
	`Identifier = Identifier(parameters)`
		`# Subroutine without a return value`
		`showAdd(2, 3)`
		
		`# Subroutine with a return value`
		`num1 = float(input("First number? "))`
		`num2 = float(input("Second number? "))`
		`answer = add(num1, num2) * 6`

String handling

String length	`len(StringExp)`	`len("computer science")` `# evaluates to 16 (includes space)`

Position of a character	`StringExp.find(CharExp)`	`"computer science".find("m")`
		`# evaluates to 2`
		
		`title = "Algorithms"`
		`space = title.find(" ")`
		`# space will have the value -1 since`
		`# title does not contain a space`

Substring using slices (the substring runs from the character **at** the first `IntExp` to the character one position **before** second `IntExp`).	`StringExp[IntExp:IntExp]`	`title = "Computer Programs"`
		`print(title[9:17])`
		
		`# prints the string "Programs"`
		
		`# If the first number is omitted it`
		`# defaults to 0 (the start of the`
		`# string) and if the second number is`
		`# omitted it defaults to the length of`
		`# the string. So`
		
		`print(title[:7])`
		`# prints the string "Compute", and`
		
		`print(title[12:])`
		`# prints the string "grams"`

String handling (continued)

Accessing a single character in a string (this treats a string as if it were an array).	`StringExp[IntExp]`	`title = "Computer Science"`
		`print(title[9])`
		`# prints "S"`

Concatenation	`StringExp + StringExp`	`print("computer" + "science")`
		`# prints the string "computerscience"`
		
		`# Note that no space is automatically`
		`# added between each string`

String and Character Conversion

Converting string to integer	`int(StringExp)`	`int("16")`
		
		`# evaluates to the integer 16`

Converting string to real	`float(StringExp)`	`float("16.3")`
		
		`# evaluates to the float (real) 16.3`

Converting integer to string	`str(IntExp)`	`str(16)`
		
		`# evaluates to the string "16"`

Converting real to string	`str(RealExp)`	`str(16.3)`
		
		`# evaluates to the string "16.3"`

Converting character to character code	`ord(CharExp)`	`ord("a")`
		
		`# evaluates to 97 using ASCII/Unicode`

Converting character code to character	`chr(IntExp)`	`chr(97)`
		
		`# evaluates to "a" using ASCII/Unicode`

Input/output

User input	`input()`
	
	`input(StringExp)`
	In this second form `StringExp` is printed to the screen and the cursor waits for input directly after it, so `StringExp` acts as a prompt.
		`name = input()`
		`# input returns a string, so int() or`
		`# float() must be used to convert the`
		`# string to an integer or a float`
		
		`quantity = int(input())`
		`price = float(input())`
		
		`name = input("What is your name? ")`
		
		`quantity = int(input("How many cans? "))`
		`price = float(input("How much? "))`

Input/output (continued)

Output	`print(Exp, …, Exp)`	`print(a)`
		`print(a, g)`
		
		`# Each Exp is printed with a space between`
		`# items. For example`
		
		`qty = 15`
		`print(f"Quantity {qty}")`
		`# prints the string "Quantity 15"`
		
		`# Once all the Exps are printed then the`
		`# cursor moves to a new line (this new`
		`# line behaviour can be changed by adding`
		`# end="").  So`
		
		`print("Mary had ")`
		`print("a little lamb")`
		`# will print the text over two lines, but`
		
		`print("Mary had ", end="")`
		`print("a little lamb")`
		`# will print the text on the same line`

Formatted outputs will be shown using interpolated strings (`f""` strings).	`f"…{identifier}…{identifier}…"`	`name = "BT"`
		`staff = 125000`
		
		`print(f"{name} has {staff} staff")`
		
		`# outputs "BT has 125000 staff"`

Random number generation

Random integer generation (The first `IntExp` is inclusive, the second `IntExp` is exclusive).	`from random import randrange`
	`Identifier = randrange(IntExp, IntExp)`
		`from random import randrange`
		`num1 = randrange(3, 7)`
		
		`# generates an integer between 3 and 6`