# Python Programming for GCSE

This booklet is designed to guide you through the programming tasks required for GCSE Computer Science and to aid revision. The pseudocode examples use OCR’s *Exam Reference Language* (ERL) and aren’t required for other exam boards.

## Getting Started

You can use what IDE or development environment you like for your Python programming – the programs you create will be the same.

Here are some alternatives that you can use at home:

* [Basthon](https://console.basthon.fr/) is a French site that doesn’t require you to create an account – this means that you don’t need to log in, but you do need to save your program to your computer to keep a copy. It’s responsive, but working with files is a bit fiddly – and being French, you need to click the *Exécuter* button to run your code. NB. If your browser offers to translate the page, say no – it causes the console to do strange things!
* You can create an account at [replit.com](https://replit.com/) – this will also allow you to create programs in other programming languages, and also things such as web-pages. It’s not as good as it used to be – the console on the right is more restricted than the one in *Basthon* and you can’t share your programs unless you pay, but it’s good when working with files.
* You might have a [Trinket](https://trinket.io/) account from school – *Trinket* saves your work and has a console, but you can’t save Python 3 programs unless you pay. You can work with files, but it’s difficult to see where they are.
* [Download and install an IDE such as IDLE](https://www.python.org/downloads/) on your computer – use the yellow button at the top of the page to download the latest version. This will work for Windows, MacOS and Linux.

## Help

You can watch videos on any of the techniques in the course – as well as introductions to IDLE and Replit – in [this YouTube playlist](https://www.youtube.com/playlist?list=PL_EbyzYKBbbVa4yBZsmkgH08YCDXe_qMX).

## Problem Solving

The following two terms are related to the computational thinking required when creating a program – they are not programming techniques themselves.

**Abstraction** means:

|  |
| --- |
|  |

**Decomposition** means:

|  |
| --- |
|  |

## Programming Fundamentals

### Sequence

Sequencing just means putting the commands into a suitable order to complete the task. If your program has more than one line of code then those lines will likely be sequenced. There are occasions when the strict sequence isn’t important – e.g. the order in which you define the value of your variables often doesn’t matter as long as it’s done before you make use of the variable.

### Variables

Variables are named values, stored in RAM, that can be accessed and changed at run-time – i.e. while the program is running.

In Python, the main **types** are *integer* (whole numbers), *float* (numbers with decimal parts), *string* (i.e. text) and *bool* (a Boolean value, i.e. *True* or *False*). In other programming languages *float*s can be know as *real* numbers.

Giving a variable is value is known as *assigning* or *defining* the value. When you give a variable a value, Python sets the type automatically (but not all programming languages do).

What type of variables are created by the following assignments?

|  |  |
| --- | --- |
| **Assignment** | **Type** |
| a = "Hello" |  |
| b = 123 |  |
| c = True |  |
| d = 1.5 |  |

Named values that can’t be changed while the program is running are called **constants** – they are good for things that don’t change, such as the value of pi – but Python doesn’t support constants.

### Casting

The names of the fundamental types in Python are also keywords to convert to those types, e.g. int(), str(), float() and bool(). You can also use eval() if you’re not sure whether to expect an integer or a decimal.

*Example*: complete line three of the code below to *concatenate* (join together) the text and the number:

|  |
| --- |
| a = "I was born in "b = 1969c = print(c) |

### Input and Output

*Example*: create a program that asks the user for two numbers and adds them together. Copy the code into the box below:

|  |
| --- |
|  |

### Count-Controlled Loops

*Definition*: a count-controlled loop is a type of repetition in which a certain section of code is repeated:

|  |
| --- |
|  |

*Example*: create a program that prints the numbers from 1 to 10 and copy the code into the box below:

|  |
| --- |
|  |

*NB:* For loops in Python aren’t inclusive – i.e. they don’t count up to and including the last number – but in ERL (and some other languages) they are. For example, printing the numbers from 1 to 10 in ERL could be done as follows:

### For n = 1 to 10

 print(n)

next n

Note also that structures such as loops, selection, etc., in ERL (and other languages) have a keyword to end them, such as next, endwhile or endif. You do no need to write ERL, but might need to be able to read it, so this guide only includes ERL examples where they are significantly different from Python.

### Condition-Controlled Loop

*Definition*: a condition-controlled loop is a type of repetition in which a certain section of code is repeated:

|  |
| --- |
|  |

*Example*: create a program that prints all square numbers less than 200 and copy the code into the box below:

|  |
| --- |
|  |

### Selection

Most programming languages have an if… then… else-type structure. Python also has elif. Whether it matters if you use elif and else or have multiple ifs is something we’ll look at in more detail in the programming lessons.

*Example*: create a program that asks the user for a number and tells them whether it is negative. Copy the code into the box below.

|  |
| --- |
|  |

### Why would this be more tricky if you wanted to identify positive and negative numbers? Can you update your code to do that?

|  |
| --- |
|  |

There is another type of selection that can be performed in most programming languages (and ERL), called switch/case. This has recently been added to Python where it’s called match/case instead. In most languages (and ERL), and default is like the else case, but in Python we us \_ instead.

In the following examples, entry is just a variable:

|  |  |
| --- | --- |
| **Python** | **ERL** |
| match entry: case "A": print("You selected A") case "B": print("You selected B") case \_: print("Unrecognised")  | switch entry: case "A": print("You selected A") case "B": print("You selected B") default: print("Unrecognised") endswitch |

NB. The match/case construct was only added to Python in version 3.10, so if it doesn’t work for you then check which version you’re using.

### String Manipulation

We can find the length of a string (i.e. the number of characters it contains) using the len() function, e.g. len("hello") would return 5. This can be used with a literal value or a variable (and can also be used tell you how many items there are in a list).

There are *methods* (a method is something that comes after a variable or data structure, with a full stop before it), e.g.

* "hElLo".upper() returns 'HELLO'
* "hElLo".lower() returns 'hello'
* "hElLo".title() returns 'Hello' (i.e. it capitalises the first letter of each word)

Python allows you to pick out certain characters of a string using a technique called *string slicing* – it’s essentially the same as picking items out of a list. You can use one, two or three values (separated by colons) inside the square brackets, and you can use this technique with variables or “string literals” (i.e. ordinary text in speech marks.

1. the position of an individual character, where positions start at 0. For example, "hello" [1] would return the 'e'. You can also use negative numbers to count in from the right – e.g. "hello" [-1] would return the 'o'. Note that there is no -0!
2. a selection of the characters with positions starting at the first number and going up to, *but not including*, the second number. For example, "hello" [1:4] would return 'ell'. You can use negative numbers here as well, and you can also miss out one or both of the numbers – missing the first number means “start at the start” and missing out the second number means “go up to the end”, so "hello" [:] returns 'hello'.
3. the third number is the step size, e.g. "hello" [0:4:2] would return 'hl', which is every second letter from character 0 to character 4. This third number can also be negative if you want to count backwards, which gives us a simple way to reverse a string – e.g. "hello" [::-1] would return 'olleh'

Complete the following table. If subject = "Computer Science" then how would you use string-slicing to pick out the text in the left column? For the first four you just need to add the numbers and punctuation.

|  |  |  |
| --- | --- | --- |
| **Text** | **Python Code Required** | **ERL Equivalent** |
| 'o' | subject[] | subject.substring(1,1) |
| 'puter' | subject[] | subject.substring(3,5) |
| 'Comp' | subject[] | subject.left(4) |
| 'nce' | subject[] | subject.right(3) |
| 15 (i.e. the length) |  | subject.length |

Notice that the approach in ERL (as in some other languages, e.g. JavaScript and BASIC) is quite different. In particular:

* the .substring example means “5 characters starting at position 3”, rather than characters from positions 3 to 5
* subject.left(4) returns the left-most 4 characters
* subject.right(3) returns the right-most 3 characters

### Arrays

Python doesn’t include support for arrays, but it does have *lists*, which have similar uses. The main differences between arrays and lists are:

* items in lists can have mixed types, but items in an array are all the same type (e.g. all integers, all strings, etc.). A list in Python could look like this:
my\_list = [1, "hello", True, 3.2] (but an array couldn’t).
* items can be added to an array in any order, but lists are filled up from the start – the first item must be item 0. Arrays are declared to contain a specific number of elements (often the number is actually the highest index allowed rather than the number of items), so you could declare array people[5] (as in the example below), but then add people[3] = "ran away" first. In a Python list you can only add an item 3 if there is already an item 2.

The example in the ERL guide for arrays is as follows:

array people[5]

people[0] = "Sir Robin"

people[1] = "Brave"

people[2] = "chicken"

people[3] = "ran away"

Lists in Python don’t need to be “declared” but they do need to be created before they can be added to, so the closest Python equivalent to this would probably be (NB .append() adds an item to the end of the list):

people = []

people.append("Sir Robin")

people.append("Brave")

people.append("chicken")

people.append("ran away")

Python also has an .insert() method that you can use to insert values at other points in a list, rather than at the end – the first argument specifies the position and all indexes higher than that are increased by one. These next two examples aren’t really equivalent as the Python version needs knowledge of what’s already in the list, but the end result is the same.

|  |  |
| --- | --- |
| **ERL Array** | **Python List** |
| array people[5]people[3] = "ran away"people[1] = "Brave"people[0] = "Sir Robin"people[2] = "chicken" | people = []people.append("ran away")people.insert(0, "Brave")people.insert(0, "Sir Robin")people.insert(2, "chicken") |

You can also just create a list that contains all of the items:

people = ["Sir Robin", "Brave", "chicken", "ran away"]

You pick items out of lists in exactly the same way as you can pick characters out of strings – see above – e.g. people[1] would return “Brave”.

There is some debate about whether students should know about *dictionaries* in Python. Dictionaries aren’t mentioned in the specification and there is no ERL equivalent, but in some languages, dictionaries are called *associative arrays*.

### Random Numbers

The specification only requires knowledge of techniques such as myVariable = random(1,6) – i.e. creating a random number within a range of values.

Python doesn’t have built-in support for random numbers, but you can add it using a *library*. The Python equivalent of the above code would be:

### from random import randint

### myVariable = randint(1,6)

Note that randint() is *inclusive* – the above code could give you a 1 or a 6.

Other programming languages (and spreadsheets) also have ways of creating a random number between 0 and 1 which you then need to scale yourself. You can [read more about that here](https://www.advanced-ict.info/interactive/random.html).

### File Handling

File handling used to be tested extensively in the coursework tasks. None of the GCSEs has coursework any longer, and all of the recent exam questions about file-handling have focussed on knowledge of the key steps:

1. open the file
2. read or write the required text
3. close the file

We will practise these techniques in the practical programming lessons.

### SQL

You are required to know a small amount of SQL for most of the GCSEs. This isn’t really a programming language so will be covered separately

### Functions and Procedures

Functions and procedures are both examples of *sub-programs*. These are sections of re-usable code that are usually used to perform a simple, single task.

The difference between the two is that a function *returns* a value and a procedure doesn’t. Returning a value isn’t the same as displaying it – print("Hello") displays the text *Hello*, but it doesn’t return a value, e.g. anything that you could assign to a variable, whereas len("Hello") returns a value (i.e. 5) but doesn’t display anything. You will usually need to do something with the returned value – e.g. print it, store it, or use it with if to make a decision.

Functions and procedures can both take an argument – inside the function or procedure this becomes a parameter. This is a value that is passed into the code and can be used in calculations, etc.

Below is a program that contains a procedure definition and a line of code that calls it:

def greet(name):

 print("Hello", name)

greet("Andrew")

In the procedure call greet("Andrew"), the string literal “Andrew” is the argument. In the procedure itself, name is the parameter – it acts like a variable that stores the string “Andrew”.

Values are returned from functions using the return keyword. You can return any *type* of variable, and also things like *lists* and *dictionaries*.

*Example*: create a **function** called double that takes a number as an argument. Add a function call to print double(2). Copy the code into the box below.

|  |
| --- |
|  |

Variables that are created in the main part of the program – i.e. outside the function or procedure are called **global** variables. Their values can be accessed anywhere – a function or procedure can use them. A variable that is created inside a function or procedure can only be used inside that function or procedure – it is called a local variable.

For example, have a look at the following program snippet:

def birthday():

 global age

 celebrate = True

 print("Happy birthday", name)

 age += 1

name = "Andrew"

age = 21

birthday()

print(age)

print(celebrate)

If you run this program, what do you think the output will be?

The code execution starts on line 7 by setting the values of name and age, which are *global* variables. Then the code calls the procedure birthday().

The procedure needs to use the value of the *global* variable name, but it doesn’t need to change it, it just prints it. It does need to update the age, though – global age on line 2 allows it to do this.

The variable celebrate hasn’t been used anywhere else up to this point, so it’s being created for the first time inside the procedure – it is *local* to birthday().

The output of this program is therefore:

Happy birthday Andrew

22

…but then you get an error to say that “*the name ‘celebrate’ is not defined*” on the final line of the program because celebrate doesn’t exist outside birthday().

## Additional Techniques

These aren’t in the “fundamentals” section of the specification but you are still required to know about them

### Characters

You are expected to know about ASCII codes. You do not need to know the codes for any of the characters, e.g. that A is represented by 65, but you do need to know that the codes are in alphabetical order – e.g. if you are told that A is 65 you need to know that B would be 66. Note that upper and lower case letters have different codes.

The keywords ord() and chr() convert between a single character and its ASCII code. For example:

* ord("A") would return 65
* chr(65) would return 'A'

ERL also uses CHR(), but uses ASC() instead of ORD() – as BASIC does.

### Do… While

The ERL guide, and some languages such as *Visual Basic*, have a structure like this:

do

 answer = input("New answer")

until answer != "Correct"

Python doesn’t have this, but it is similar to a while… loop except that the condition is checked at the end of the loop rather than at the start. This means that the code inside the loop will always be run at least once (whereas in a Python while loop it might not run at all if the condition at the top has already been met).

Some programming languages (e.g. BASIC again) also have something similar called repeat… until. Again the condition is checked at the end, but the logical sense of the test is reversed, e.g. do… while x == True would be the same as repeat… until x == False.

### Comments

In Python a comment begins with a #, in ERL (and the C family of languages) it begins with //.

## Yet More…

The things above are the only programming constructs that are explicitly mentioned in the OCR specifications and their programming guide.

There are other things covered in the theory content – e.g. representation, modulo arithmetic, Boolean logic, etc. that could also be useful in programming tasks, and we will look at additional techniques that you can use in your own programming and in exam questions.