Number of weeks (between 6&8)

6

Content of the unit

Programming unit covering chapter 3 and 4. Here students will learn Python. They will cover everything from how to create a program to what makes a program

Assumed prior learning (tested at the beginning of the unit)

No previous learning is necessary with this unit. Pupils may have had some experience of using variables and with a variety of relational operators such as **If** and **Repeat** in graphical block-based languages such as Scratch. Applying this knowledge will help their understanding of a text-based language such as Python.

Assessment points and tasks

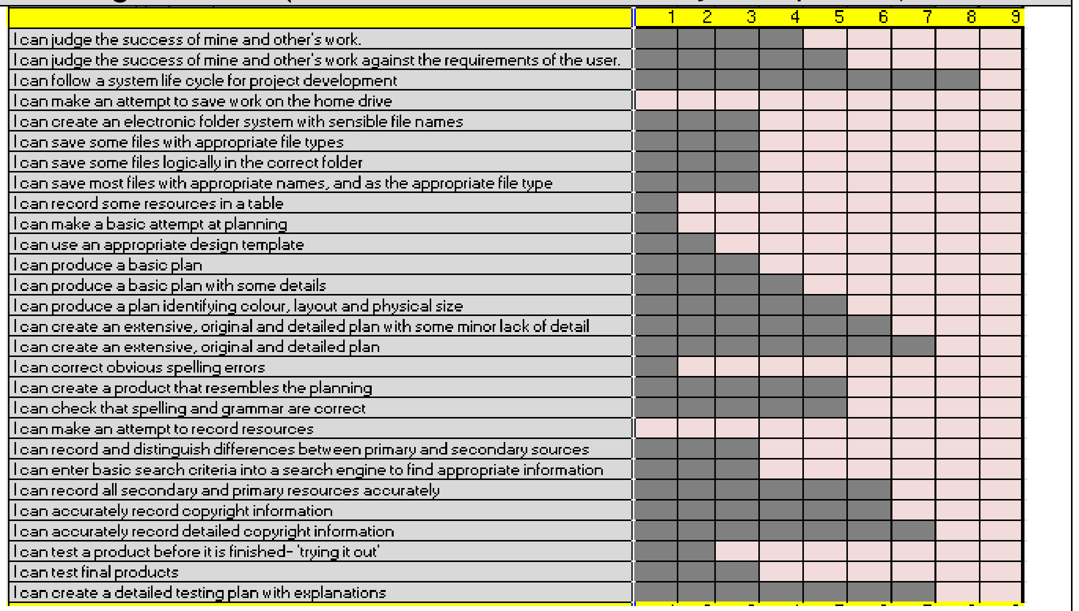
Each week there will be exam style questions as part of the teaching which will be assessed. There are also tasks which need to be completed each lesson.

This Schemes of work is aimed towards GCSE Grades A*- C

Written feedback points

Provided every 2 weeks

Learning Outcomes (tested at the end and related to subject competences)



	1	2	3	4	5	6	7	8	9
I can judge the success of mine and other's work.	■	■	■	■	□	□	□	□	□
I can judge the success of mine and other's work against the requirements of the user.	■	■	■	■	■	□	□	□	□
I can follow a system life cycle for project development	■	■	■	■	■	■	■	■	□
I can make an attempt to save work on the home drive	□	□	□	□	□	□	□	□	□
I can create an electronic folder system with sensible file names	■	■	■	□	□	□	□	□	□
I can save some files with appropriate file types	■	■	■	□	□	□	□	□	□
I can save some files logically in the correct folder	■	■	■	□	□	□	□	□	□
I can save most files with appropriate names, and as the appropriate file type	■	■	■	□	□	□	□	□	□
I can record some resources in a table	■	□	□	□	□	□	□	□	□
I can make a basic attempt at planning	■	□	□	□	□	□	□	□	□
I can use an appropriate design template	■	■	□	□	□	□	□	□	□
I can produce a basic plan	■	■	■	□	□	□	□	□	□
I can produce a basic plan with some details	■	■	■	■	□	□	□	□	□
I can produce a plan identifying colour, layout and physical size	■	■	■	■	■	□	□	□	□
I can create an extensive, original and detailed plan with some minor lack of detail	■	■	■	■	■	■	□	□	□
I can create an extensive, original and detailed plan	■	■	■	■	■	■	■	□	□
I can correct obvious spelling errors	■	□	□	□	□	□	□	□	□
I can create a product that resembles the planning	■	■	■	■	■	□	□	□	□
I can check that spelling and grammar are correct	■	■	■	■	■	□	□	□	□
I can make an attempt to record resources	□	□	□	□	□	□	□	□	□
I can record and distinguish differences between primary and secondary sources	■	■	■	□	□	□	□	□	□
I can enter basic search criteria into a search engine to find appropriate information	■	■	■	□	□	□	□	□	□
I can record all secondary and primary resources accurately	■	■	■	■	■	■	□	□	□
I can accurately record copyright information	■	■	■	■	■	■	□	□	□
I can accurately record detailed copyright information	■	■	■	■	■	■	■	□	□
I can test a product before it is finished- 'trying it out'	■	■	□	□	□	□	□	□	□
I can test final products	■	■	■	□	□	□	□	□	□
I can create a detailed testing plan with explanations	■	■	■	■	■	■	■	□	□

At the end of this Unit all pupils should be able to:

- Run simple Python programs in Interactive and Script mode
- Write pseudocode to outline the steps in an algorithm prior to coding
- Write programs using different types of data (e.g. strings and integers)
- Correctly use different variable types (e.g. integer and floating point), assignment statements, arithmetic operators
- Distinguish between syntax and logic errors and be able to find and correct both types of error
- Use relational operators to control the order in which program statements are executed and in what order (if and while statements)
- Use comments to document their programs and explain how they work
- Write an error-free, well-documented program involving selection and iteration, but with some help given

		Most pupils will be able to: • Write an error-free, well-documented program involving selection and iteration • Describe how a binary search is carried out • Explain the advantages of a binary search over a linear search for an ordered list **Some pupils will be able to:** • Devise their own algorithms to solve reasonably complex problems, e.g. a binary search • Test and debug their programs, and correct both syntax and logic errors • Make allowances in their programs for user input errors, ensuring that the program still runs to a successful conclusion – which may include printing an error message and stopping the run

Lesson	Clear learning intentions	Clear success criteria	Hook	Presentation of content	Guided practice	Independent practice (homework)	Closure

1	• Learn what Python is and some of the applications it is used for • Run a simple Python program in Interactive mode using the **input** and **print** functions • Write, save and run a program in Script mode • Understand what a syntax error is and how to interpret an error message • Know the rules for variable names and use variables in a program • Understand the use and value of comments in a program	Understand what Python is and where it is used. Run a simple Python program in interactive mode. Write, save and run a program in Script mode Understand what a syntax error is. Understand what a variable is in a program. Understand the use of comments in a program.	Introduce pupils to the concept of sequencing instructions to fry an egg, for example, and highlight the importance of getting them all in the right order. Correct order given.	Introduce pupils to Python's development environment – IDLE. Begin in Interactive mode and explain that this enables them to get instant results on their code, line by line. They will proceed to Script mode later. Demonstrate a simple *Hello World* program. Pupils can then have a go themselves. Pupils can look at the error feedback in the interpreter window and try to work out what is causing the error **Python Script Mode** Introduce pupils to Script mode by opening a new window in IDLE. (**File, New Window**) Here, programs can be written, saved and then run. Files should be appended manually with *.py*. Demonstrate a simple input and output program using the *input* and *print* statements. (e.g. **MyFirst.py** in PP Guide). **Colours used by Python code** The colour-coding, called syntax highlighting, is used to help the programmer make sense of the code and makes it easier to spot syntax errors. *Strings are green* *Output is blue* *Comments are red* Explain that text is known as a string. This is highlighted in green in IDLE. Explain what a variable is and what variables are used for. Pupils may already have used the variable *firstname* in their earlier program.	Pupils open up Python and write a simple program called 'Hello World'. Pupils to open a new window in IDLE. (**File, New Window**) Here, programs can be written, saved and then run. Pupils save files with *.py* to ensure they are maintained in association with IDLE when they are subsequently opened. File extension *.py* is not always automatically added. Pupils use **Worksheet 1 Strings and Variables** to practise their programming so far. Solutions to each exercise are given.	Describe what is Python? Research on MagPi and read the first version of MagPi.	Re-cap on what pupils know about strings, variable and syntax error.

2	• Understand the importance of using correct data types *string*, *integer*, *float*
	• Understand how to use assignment statements correctly
	• Perform arithmetic using the BIDMAS rule
	• Use the *int*, *float* and *round* functions
	• Write a program involving input, calculation and output
		Understand how to use a correct data types such as string, integer and float when programming.
		Know what the BIDMAS rule is.
		Know how to use integer, float and round functions.
		Know how to write a program involving, input, calculation and output.
			Show pupils the starter Animal program and ask them as individuals, or in pairs, to think about and explain what each line of code does, and what the output would be for a given input. *Animal[0]* displays the first letter of the string value stored in 'Animal'.	Discuss the different data types. (Boolean is included for completeness but will not be used in this unit.) Explain that Python automatically assigns a data type to a variable unless you manually override it. Pupils to complete **Worksheet 6-2: Data Types**. Use the table in the PowerPoint guide to show answers.
				Demonstrate the importance of data types in Python using the *mynumber* example.
				`mynumber = input("Enter a number")`
				`print (mynumber + mynumber)`
				The output of *print 7+7*, given an input of 7, would be 77. This is because *mynumber* is automatically given a string data type. Python is concatenating two strings and prints the character '7' twice. To fix this, the *int()* function is required to manually override Python and force the *mynumber* variable to become an integer.
				`mynumber = `**`int(`**`input("Enter a number")`**`)`**
				`print (mynumber + mynumber)`
				This method assigns the data type on input. Alternatively, you can change the variable's type on output:
				`mynumber = input("enter a number: ")`
				`enter a number: 4`
				`print (int(num)+int(num))`
				`8`
				`print (num + num)`
				`44`
					Pupils to extend the program **Sleep Calculator** to find the total hours asleep every month assuming 4.35 weeks per month. This code will crash if they enter a non-integer value for *hourspernight*. They will need to change *int()* to use *float()* instead to cope with the decimal fraction.
					Pupils continue to develop the program to find the number of days they sleep for. Pupils use *float()* here since any value multiplied by 4.35 will become a decimal fraction.
					Pupils to tidy up their code by adding comments and a title and improving the user experience with appropriate prompts.
						Develop a program to calculate the monthly cost of their mobile phone usage. Remember to include at least one helpful comment in the program!
						Pupils to be given **Solution MobileCosts.py** for guidance.
							Re-cap on what pupils know about integer, string and float as well as the BIDMAS rule.

3	• Use selection statements *if*, *else* and *elif* in a program • Use indentation correctly to define a block of code	Understand how to use *if*, *else* and *elif* in a program. Understand how to use indentation to correctly define a block of code.	Pupils to understand the IF statement in the **password checker program.** Highlight the use of the colon ':' in the code and the requirement to indent their code. Python, unlike other languages, uses indentation as part of its syntax and will not execute unless it is done correctly. The block of code within the IF statement (or loop) ends when the indentation decreases one step (e.g. one tab stop or two spaces).	Demonstrate the Password Checker program to verify their ideas on its function. Remove a colon and make changes to the indentation to highlight the errors that occur as a result, and emphasise the importance of syntax. Code can be indented using either the space bar or the tab key. The important thing is to be consistent. For example, use one tab stop or two spaces. Go over the comparison operators, make up a few examples and ask pupils to write the Python code. E.g. **If bank balance is less than 0, print "Overdrawn".** **If lives = 0, print "Game Over"** Ask pupils to think of applications and areas where an *if* statement might occur in code. Using a mobile phone as a class example, the security PIN checker or unlock screen function, taking a call, sending a message or simply touching the screen all provide examples of where an IF statement might be used in the program code running inside the mobile. When pupils are comfortable with the *if* and *else* statements, introduce the *elif* statement in order to increase the number of options. Help pupils to write their code and debug as necessary. Demonstrate the finished program **SpeedChecker.py**. Make sure pupils all include at least one comment to say what their program is about.	Pupils identify two applications where *if* statements may be used using **Python Worksheet 4 If statements**. They should then try writing the algorithms that may have been used. For example: *If end of washing cycle then* *Sound alert* *Else* *Continue cycle* Pupils will now begin to write some code themselves that might be used in the software inside a police speed camera. The syntax for the IF statement on a previous slide may be a useful reference for them. Pupils should add an *elif* statement to their speed camera program to issue a warning between 70 and 75mph.	Describe what are conditional statements? Write a conditional statement to check the number entered by using is bigger or smaller than the one stored in an array.	Demonstrate some of the pupils **SpeedChecker.py** programs on screen.

4	• Learn to write algorithms in pseudo-code • Review the difference between syntax errors, run-time errors and logic errors • Learn techniques for debugging programs	Understand how to write algorithms in pseudo-code. Understand the difference between syntax errors, run-time errors and logic errors. Understand techniques for debugging programs.	Pupils to write a short piece of pseudo-code as a class exercise to unlock a mobile phone. Pupils to write their own pseudo-code algorithm to display the correct hat size for a user based on their input.	Explain the different types of error. They will have experienced dozens of syntax errors by now! A run-time error is caused by a problem that the program cannot resolve whilst it is running. An example would be a division by zero, or being unable to locate a file that the program is looking for.	Pupils to look at two programs which contain various errors. **SeasonFinder.py** is simpler to debug than **TaxCalulator.py**. Each program has a version with the bugs highlighted as comments for demonstration purposes. **SeasonFinder.py** contains only syntax errors. **TaxCalculator.py** contains more syntax errors and one logic error for eagle-eyed pupils!	Describe what are Algorithms? State the difference between different form of algorithms. .	Review what the difference between syntax, run-time error and logic error are.

5	• Use a *while* loop in a program • Use an *if* statement within a while loop • Use a function to generate a random number • Understand and apply the principle of a binary search • Compare the efficiency of a binary search with a linear search	Understand how to use a *while* loop in a program Understand how to use an *if* statement within a while loop Understand how to use a function to generate a random number Understand how to apply the principle of a binary search Understand how to compare the efficiency of a binary search with a linear search	Pupils look at a flowchart to describe a simple decision loop.	Display the pseudo-code example of using a while loop to create a simple program to repeatedly ask a quiz question until the correct answer is entered. Remind pupils of the *method* syntax (which they used briefly in Worksheet 1, question 5) to change the answer to title case – do they understand why this step is desirable? Pupils should copy the syntax and create the program themselves. Introduce the concept of a counter variable to count the number of times the loop is repeated. Show the next pseudo-code example for a program designed to add up a sequence of numbers and display the total and the number of entries made. Highlight the *Count* variable. Demonstrate with the **PasswordEntry** program how an *if* statement can be used inside a *while* loop. Compare this with the corresponding pseudo-code in the PowerPoint guide. When they are ready to begin coding, show them the syntax to import the random number module, and how to call the module using the *randint()* function. `number = random.randint(1, 10)`	**Extension challenge** Pupils may be able to write a program where they think of a random number between 1 and 100 (or 1 and 1000!) and the computer does a binary search to guess the number. A solution is given in **Binary search.py**.	Describe what is iteration? What are the different methods of iteration and why they are useful?	Review back the lesson and ask the class how and where to use a *while* and an *if* statement within a program.

6	• Compare alternative algorithms for a given problem • Use a linear search to find a number • Understand how a binary search works	Understand alternative algorithms for a given problem Understand how to use a linear search to find a number Understand how a binary search works	Pupils should look at different ways of searching an ordered list and compare these in order to find the most efficient.	Show pupils that there are three methods of searching a sorted list: a linear search, a random search and a binary search. Display and discuss the pseudo-code methods of the linear search and the random search. Ask pupils what the maximum number of items that would need to be read for each type of search might be if there were 100 items in the list. A linear search may need 100 iterations of the loop if the item was the last in the list. A random search may never find the item if it never generates the correct position to look in. Explain the principle of a binary search: find the midpoint of the range 1-100, if 50 is too low bisect the new range 51-100, and guess 75, etc until the number is correctly guessed. The maximum number of guesses needed for a number of 2^n is n. 100 is between 2^6 and 2^7 so maximum number of guesses needed is 7. Let pupils try this out in pairs for numbers between 1-16, 1-32, 1-64.	Pupils to plan the code for a binary search to get the computer to guess a number they thought of. They would tell the computer, with each guess, if it was too high, too low or correct and it would do the rest.	Revise all the topics covered in the test for next lesson	Re-cap on linear, random and binary search.
7	• Assessment of learning for the unit.			Briefly revise at the start of the lesson. Explain rules for the assessment. Explain how the assessment is to be completed.	9.1 Final Assessment		