

Python 4: Complex Selection

Teaching resource

Resources

- Slides **Python 4: Selection**
- You will need to either have Python IDLE installed or have access to an online Python IDE. We have used <https://editor.raspberrypi.org/en>
- Activity worksheets are included in this lesson. You will need to distribute these to your pupils
- We have added a walk-through video below

Prior Knowledge

- Printing, mathematical operators, data types and input using Python
- Selection in Python using `if...elif...else`

Vocabulary

- **debug** - to find and remove errors in programs
- **nested selection** - having one `if` statement inside another `if` statement



3MB

Python 4 - Complex Selection.pptx

Lesson walk-through

You need to be logged in to access



Learning Objectives

To teach pupils to

- extend their use of `if..elif..else` to make decisions in Python programs
- use nested selection
- debug programs

Slide Notes

Lesson Introduction

Show **slide 3** which extends the exit ticket from the last lesson. Ask pupils to work out what the Python program would output for each of the given input examples. You might like to split the class or ask them to do this in pairs and then show their answers on mini whiteboards.

The purpose of this exercise is to ensure that pupils can trace through a program without having to run it and that they understand the order of instructions in nested selection statements. You might need to explain some of the answers, e.g. Y and -10

The answers are:

Input

Y
0

Y
-10

Y
10

N
0

N
-1

cheese
-100

Activity 1: Nested Selection


Show **slide 6** and distribute Activity 1 and ask pupils to identify and show the route the algorithm would take through the flowchart given for specific inputs. This allows you to check that individual pupils have understood this. They could draw a line or highlight, as you see fit.

Answers are given in **slides 7–8**. Discuss these and ask pupils to check their work.

Activity 2: Nested Selection in Python


Show **slide 10** on the board and ask pupils to write a program for that algorithm. This flowchart intentionally mirrors the previous example to allow pupils to build familiarity.

As the exercise on the worksheet is going to include testing, you might like to ask the class what is the minimum number of test cases they need to demonstrate that their program produces the correct answers. If they need a hint, you might like to ask them about the routes through the algorithm. The answer is that they need at least four test cases to check that each branch produces the right answer.

 A **test case** is a complete set of inputs that allow a program to run. In this case, an example of a single test case is *15* and *Yes*

The worksheet includes test cases that pupils should use to fully test their code. They are not checking for robust code at this point, e.g. if they enter a non-integer for age, the program will crash. They are checking that the program produces the correct output, given suitable inputs.

If you haven't talked about testing before, you might like to explain the process to the class. The test table is intentionally only partially complete.

 Test tables are used to plan and record the outcomes of individual test cases. It is helpful to encourage pupils to record the **expected output** (what **should** their program produce) and **actual output** (what **does** their program actually produce?).

The Python solution is shown on **slide 11**.

Use **slides 12–14** to discuss testing further. Pupils must actually run the program with the given inputs to check that it produces the right output, rather than assuming that it does! This discipline helps pupils to create robust programs and tackle tracing questions that they might encounter in GCSE examinations.

Key Concept: Debugging

Debugging program code can be difficult and frustrating. The purpose of this section of the lesson is to give your pupils some experience of debugging. Encourage them to try these things the next time they get stuck, rather than just asking you!

Show **slide 16** and ask pupils what the user would expect to be output when they enter *14* and *Yes*. The answer is "*The ticket price is £ 5*". However, as you can see, the program outputs "*The ticket price is £ 8*". Don't allow pupils to say what the error is yet, this exercise is about methods for finding errors.

Show **slide 17** and ask those pupils who have spotted the error, what questions they could ask someone else to help them find the error or what they could do to help themselves if they were stuck. If the class is stuck, you might want to hint that they should search the code for where **price** gets set to 8 and then work backwards.

Slide 18: An example of a useful question that you or another pupil could ask someone who was stuck on this exercise is, "What must the values of age and member be for line 10 to be reached?"

Slide 19 shows another way of finding an error in a program like this is to look at what **does** work. We could check that the values have been stored correctly by outputting the value of `age` and `number` on a new line of code under line 2. If they are still stuck, they could add the print statements on lines 4 and 10. Does this output help pupils to find the error?

| | |
|---|--|
| <pre>1 age = int(input("How old are you? ")) 2 member = input("Are you a member? ") 3 if member == "Y": 4 print("They are a member") 5 if age < 15: 6 price = 5 7 else: 8 price = 7 9 else: 10 print("They are not a member") 11 if age < 15: 12 price = 8 13 else: 14 price = 10 15 print("The ticket price is £", price)</pre> | <pre>How old are you? 14 Are you a member? Yes They are not a member The ticket price is £ 8</pre> |
|---|--|

You can see from the output above, the program has decided that you are not a member! Why? It must be that member is not equal to "Y", but we asked the user to enter *Yes* or *No*! That is the error. Line 3 should be changed to

```
if member == "Yes":
```

Slide 20 describes rubber duck debugging. This is a really powerful method that you can practice with your class the next time they ask you to help them find an error in their code. If you ask them to explain why they think their code is correct, they will often find the error themselves. Explaining their code requires them to think about it in a more useful way.

Activity 3: Debugging

Slide 22: Ask pupils to do the debugging exercises. It is important to encourage them to explain how they would find the error, rather than just identifying it, but this can be a difficult skill to develop. The answers follow each question slide.

Exercise 1 - Explanation

In Exercise 1, line 4 was not actually storing the result of `mark / 100` in the `mark` variable. So when line 5 was run, `mark` was still a whole number so it was never less than 0.5

Line 4 should be changed to `mark = mark / 100`

For an even better solution, create a new variable to hold the percentage by changing line 5 to `percentage = mark / 100` and line 6 to

`if percentage < 0.50:`. This is better because it avoids the poor practice of changing the data type of a variable.

Exercise 2 - Explanation

In Exercise 2, the error is that line 10 is indented so it is part of the `else` branch of the last condition. This means that the program only outputs the result if the score is less than or equal to the high score. The solution is to remove the indentation so that the `print` is lined up with the `if`. This is shown on **slide 27**.

Plenary

In the plenary, you might like to recap on the use of nested selection and debugging techniques. Then ask pupils to spot the error in the example on **slide 28**. This is a tricky one that you might like to follow up in the next lesson. The correct is `if answer == "Yes" or answer == "yes":` because every part of a conditional statement must evaluate to a Boolean value: True or False. The string "yes" is not a Boolean value, so it must be compared to something.