

School of Computing, Creative Technology and Engineering

**Module: Fundamentals of Computer Programming**

**Academic Year: 2023/24**

**Level 4: Semester 1**

**Assignment Title: Practical - 03**

**Date Due: Jan 16, 2025**

**Tutor: Saurav Gautam**

**Student Name: Sange Doma Tamang**

**Student ID: 10260**

GitHub link: <https://github.com/Sangedoma/Programming-Portfolio/tree/main/portfolio>

**Introduction to Programming**

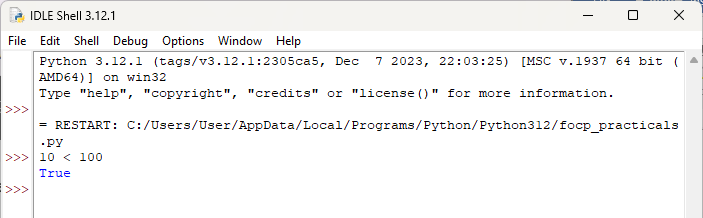
**Lab Worksheet**

**Week 3**

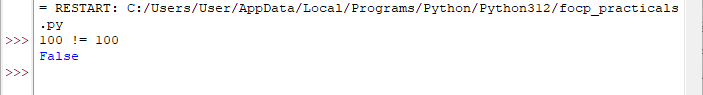
**Boolean Expressions:**

TASK: Start the Python Interpreter and input the following expressions, noting each result.

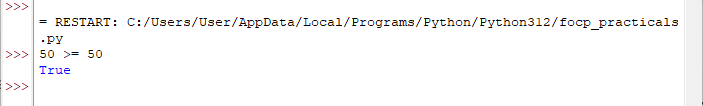
1. 10 < 100



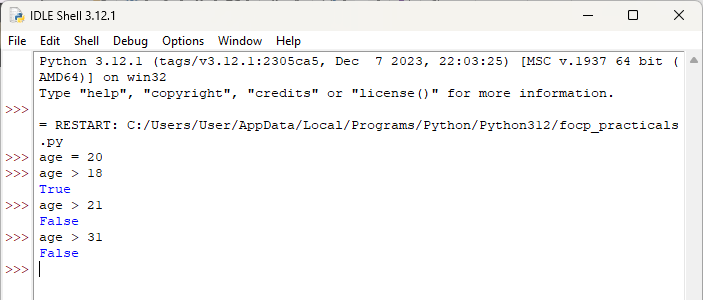
1. 100 != 100



1. 50 >= 50

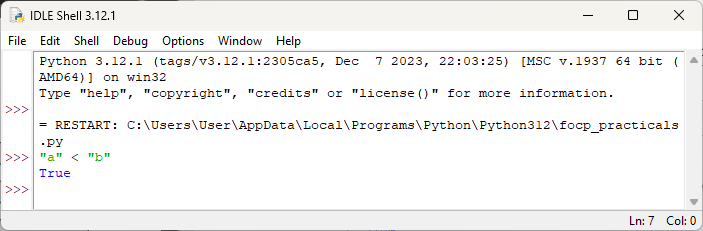


TASK: Input a program that defines a variable called ‘age’ that is initialised to your own age. Then type several Boolean expressions that compare the variable to see whether it is less than ‘18’, ‘21’ then ‘31’.

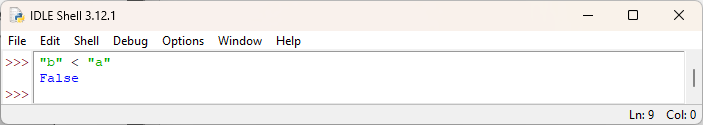


TASK: Try inputting the following code and note the results.

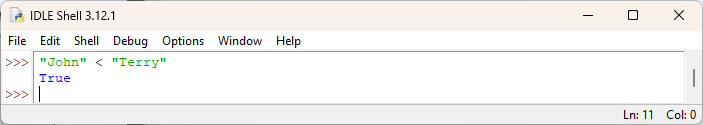
1. “a” < “b”



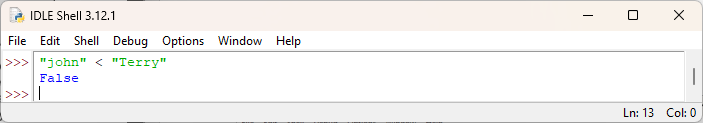
1. “b” < “a”



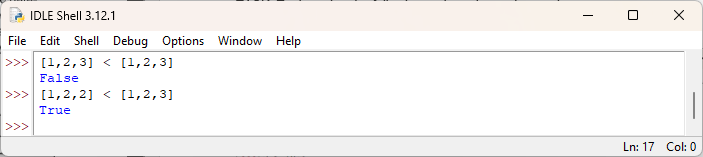
1. “John” < “Terry”



TASK: Try inputting the following code and note the result. Try to work out why the answer is different from the previous expression (look carefully, it is different).

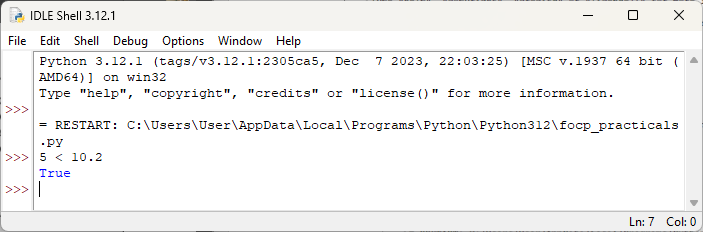


It is even possible to apply relational operators to lists (but less commonly seen), e.g.

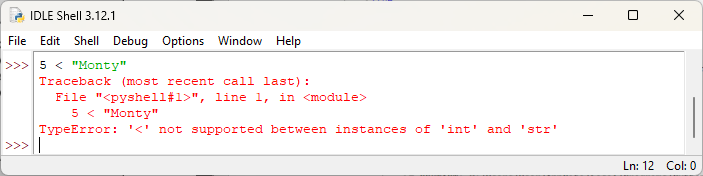


TASK: Try inputting the following code and note the results.

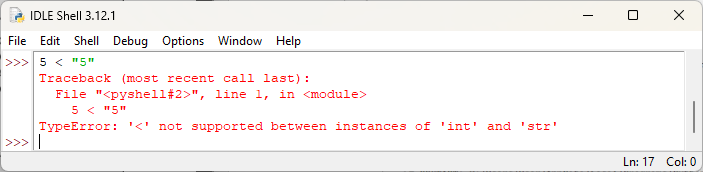
1. 5 < 10.2



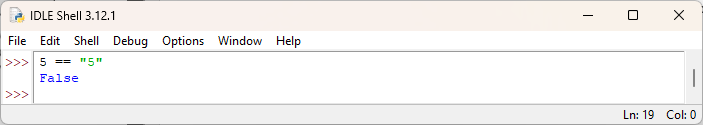
1. 5 < “Monty”



1. 5 < ”5”

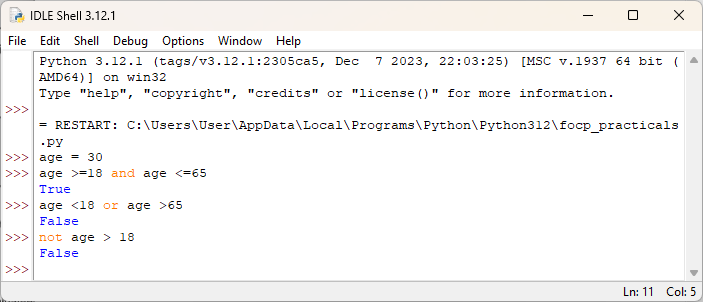


The equality operators are slightly more forgiving however, and rather than report an error due to type mismatch, can return unexpected results. For example, note how the result of the following expression is False, rather than an error.

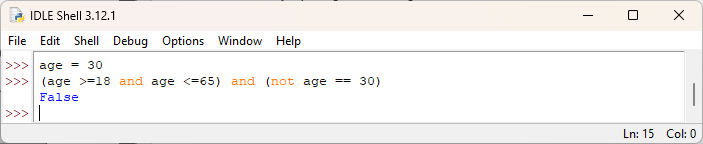


**Logical Operators within Expressions:**

TASK: Try inputting the following code and examine the results.

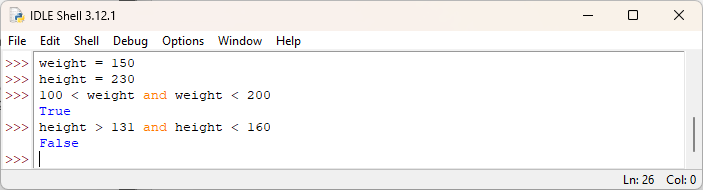


TASK: Try inputting the following code and examine the result.



**Chaining relational operators**

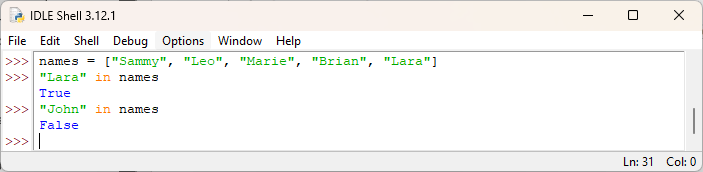
TASK: Try inputting two expressions that use operator chaining and are equivalent to the two expressions shown below. (note: you may first want to first assign values to the ‘weight’ and ‘height’ variables for testing purposes)



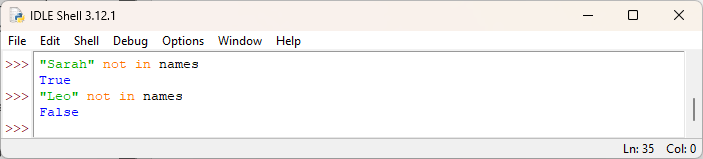
**Membership Testing**

TASK: Input the examples above but with alternative operand values, that result in both True and False results.

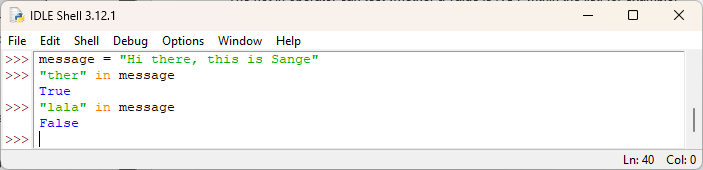
For example, the in operator can be used to test if a value is within a specific list.



The not in operator can test whether a value is NOT within the list, for example:

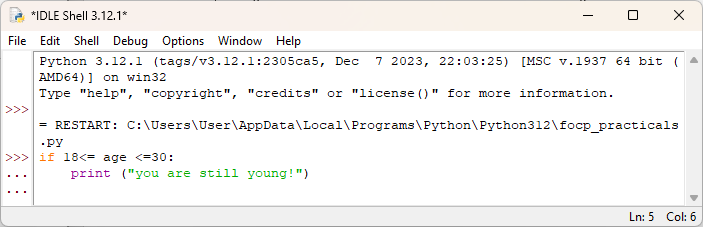


When the right-hand operand value is a string, then membership tests will return True if a substring is present, for example:



**The ‘if’ statement**

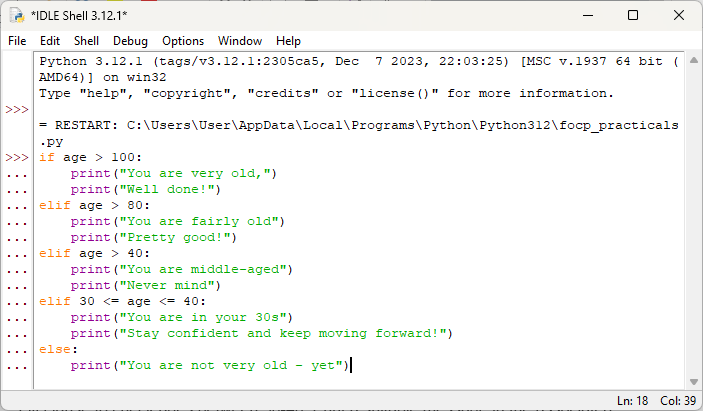
TASK: Try writing an if statement that checks if someone is between the ages of 18 and 30 inclusive. If they are, then print a message saying “you are still young!”



**Using the ‘elif’ clause**

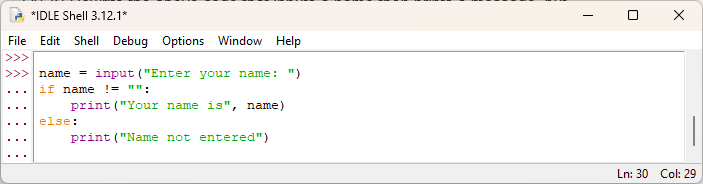
If there are multiple conditions that need to be checked then if statements can be chained using one or more elif clauses. These must follow the initial if statement. If the optional else clause is provided it must always appear last.

TASK: Try writing an if statement similar to the last example that includes an extra elif clause to check ages between 30-40. Print a suitable message in the associated code block.



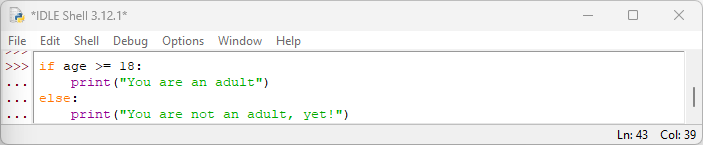
**Non-Boolean conditions**

TASK: Rewrite the above code that inputs a name then prints a message, but change the condition so it explicitly uses a Boolean expression. Use the example below to help.

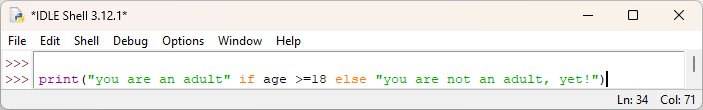


**The Ternary Operator**

TASK: Rewrite the code shown below as a single line Ternary expression.

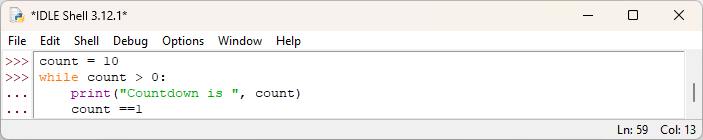


The above code as a single line Ternary expression:



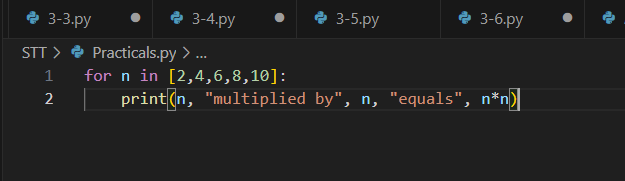
**Using ‘while’ and ‘for’ loops**

For example, a while loop may look like the following:

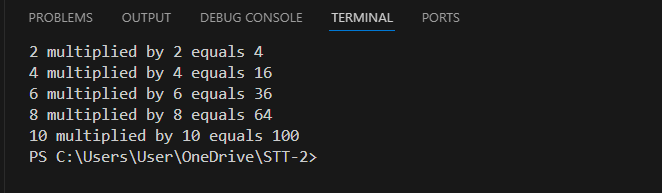


For example, a for loop may look like the following:

Code:

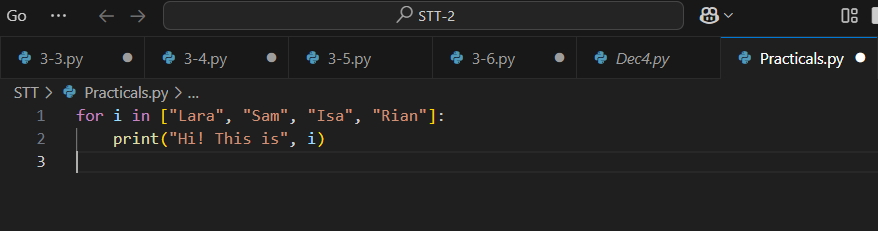


Output:

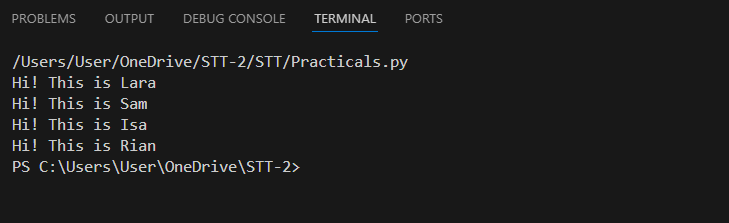


TASK: Input and execute a for loop that iterates over a list of four names, printing each of them to the screen.

Code:



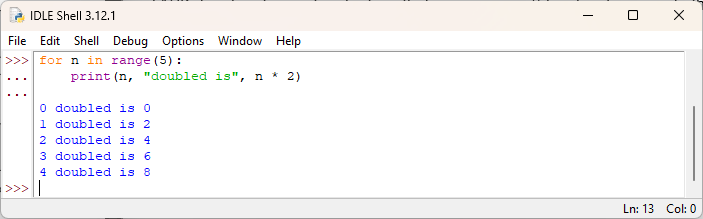
Output:



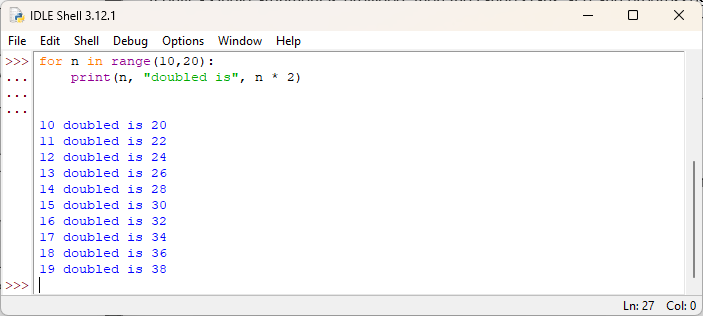
**The range () function**

When using for to iterate over a set of numeric values it is often more convenient to use the range () function, which allows a range of values to be specified. The range function generates an arithmetic progression between two boundaries with an optional ‘step’ value.

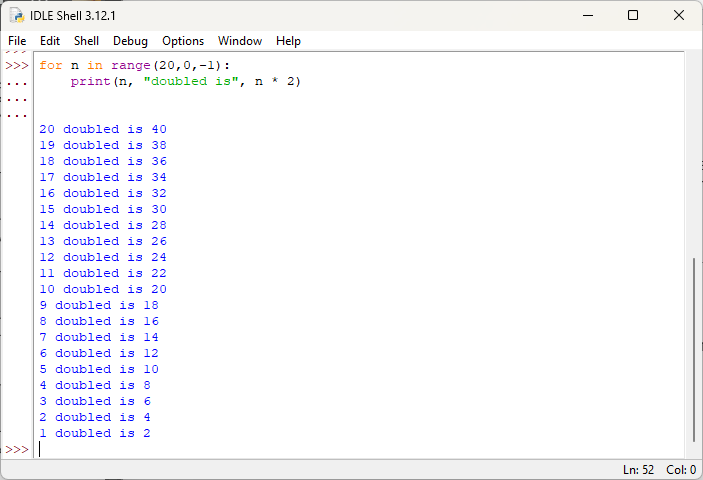
If only a single argument is provided, then the range starts at 0 and progresses to the given value -1, so to iterate over the values between 0 to 4 we could use the following:

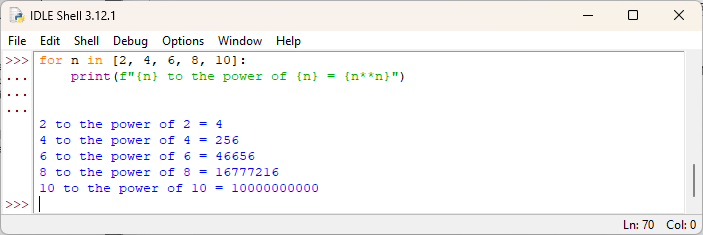


A lower bound can also be specified, so to iterate over the values between 10 to 19 we could use the following:



Finally, a ‘step’ value can be included. This allows the generated values to either increase (or decrease) by values other than 1. So to iterate over the values from 20 down to 1 we could use the following:

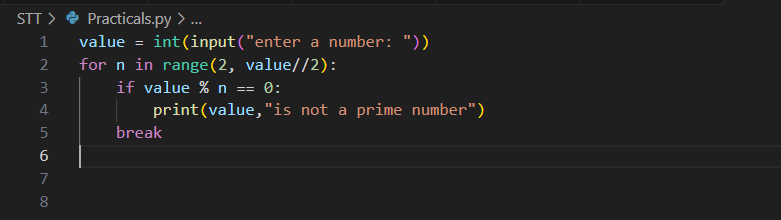


TASK: Input and execute a for loop that uses a range () function to generate the following output:

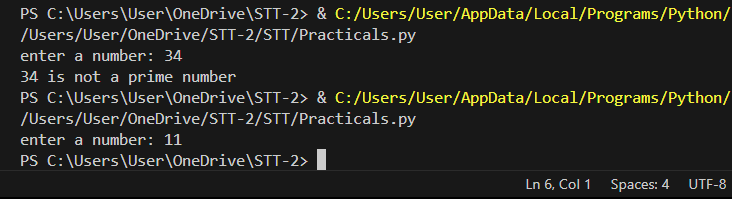
**Using ‘break’ within a loop**

Being able to break out of a loop early is convenient when some condition occurs indicating that the remaining iterations are not required. For example, the following code will break out of the loop early if the condition indicates that value is found not to be a prime number, hence there is no reason to finish the remaining iterations.

Code:

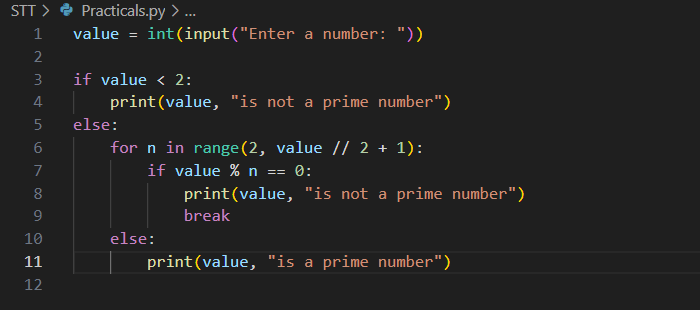
****

Output:

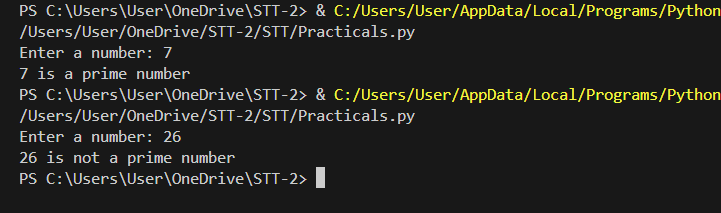


An optional else clause can be associated with loops that contains a code block which is only executed when the loop terminates normally, i.e. not due to a ‘break’ statement. For example, the code below shows a slight improvement to the previous example that ensures an appropriate message is shown if the loop terminated as normal.

Code:



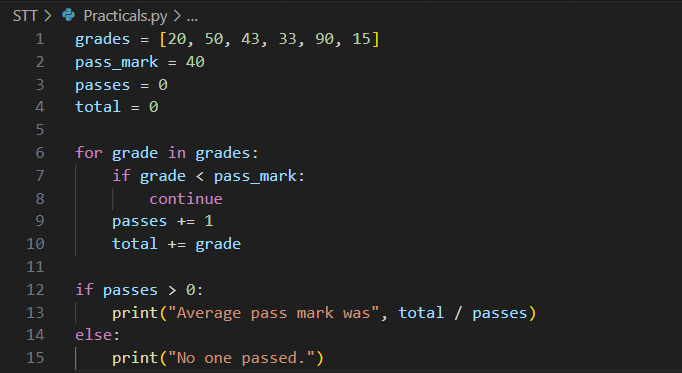
Output:



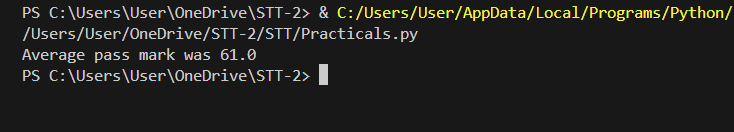
**Using ‘continue’ within a loop**

For example, the following code only executes all statements within the loop if a grade is a pass, otherwise it skips the remaining statements and continues onto processing the next grade.

Code:

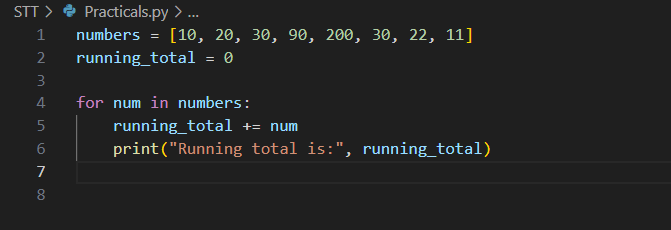


Output:

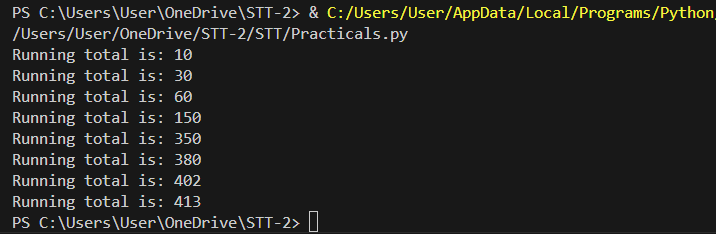


TASK: Input code containing a for loop that iterates over a list of numbers, printing a running total during each iteration. You may wish to first define the numbers list as follows:

Code:

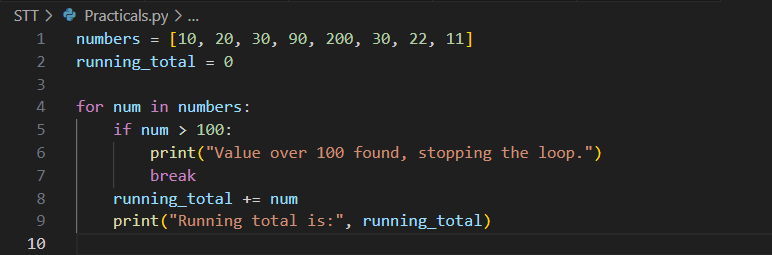


Output:

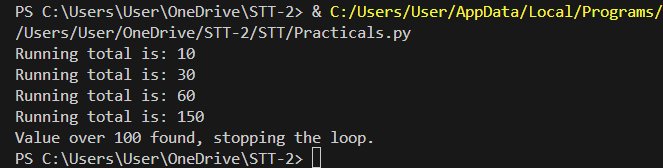


TASK: Amend your previous solution so that if any value within the list is found to be over 100 then the loop should break immediately.

Code:

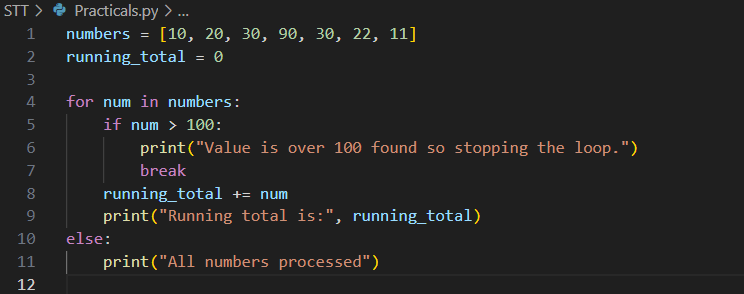


Output:



TASK: Amend your previous solution once again, so that the message “all numbers processed” is printed when the loop completes, but only if all values were less or equal to 100 (i.e. the loop did not break early)

Code:



Output:

