

Network Programming in Python I

Justin Ellis MBA

Review

Any Questions?

Errors

In writing and using this book over the last few years we have collected a lot of statistics about the programs in this book. Here are some statistics about error messages for the exercise we have been looking at.

Message	Number	Percent
ParseError:	4999	54.74%
TypeError:	1305	14.29%
NameError:	1009	11.05%
ValueError:	893	9.78%
URIError:	334	3.66%
TokenError:	244	2.67%
SyntaxError:	227	2.49%
TimeLimitError:	44	0.48%
IndentationError:	28	0.31%
AttributeError:	27	0.30%
ImportError:	16	0.18%
IndexError:	6	0.07%

Nearly 90% of the error messages encountered for this problem are ParseError, TypeError, NameError, or ValueError. We will look at these errors in three stages:

- First we will define what these four error messages mean.
- Then, we will look at some examples that cause these errors to occur.
- Finally we will look at ways to help uncover the root cause of these messages.

ParseError

Parse errors happen when you make an error in the syntax of your program. Syntax errors are like making grammatical errors in writing. If you don't use periods and commas in your writing then you are making it hard for other readers to figure out what you are trying to say. Similarly Python has certain grammatical rules that must be followed or else Python can't figure out what you are trying to say.

Usually ParseErrors can be traced back to missing punctuation characters, such as parentheses, quotation marks, or commas. Remember that in Python commas are used to separate parameters to functions. Parentheses must be balanced, or else Python thinks that you are trying to include everything that follows as a parameter to some function.

TypeError

TypeError occurs when you try to combine two objects that are not compatible. For example, you try to add together an integer and a string. Usually, type errors can be isolated to lines that are using mathematical operators, and usually the line number given by the error message is an accurate indication of the line.

NameError

Name errors almost always mean that you have used a variable before it has a value. Often NameErrors are simply caused by typos in your code. They can be hard to spot if you don't have a good eye for catching spelling mistakes. Other times you may simply mis-remember the name of a variable or even a function you want to call.

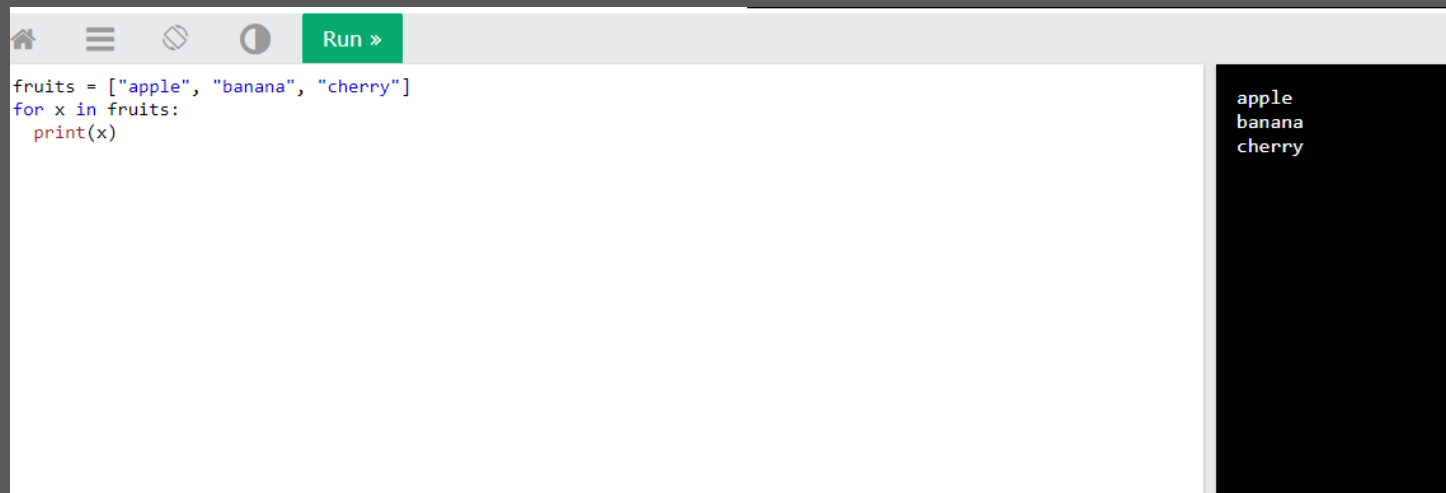
ValueError

Value errors occur when you pass a parameter to a function and the function is expecting a certain limitations on the values, and the value passed is not compatible.

For Loop

A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

This is less like the for keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages. With the for loop we can execute a set of statements, once for each item in a list, tuple, set etc.

A screenshot of a Python IDE window. The window has a title bar with icons for home, menu, search, and a green 'Run »' button. The code editor contains the following Python code:

```
fruits = ["apple", "banana", "cherry"]  
for x in fruits:  
    print(x)
```

The output console on the right side of the window shows the results of the loop:

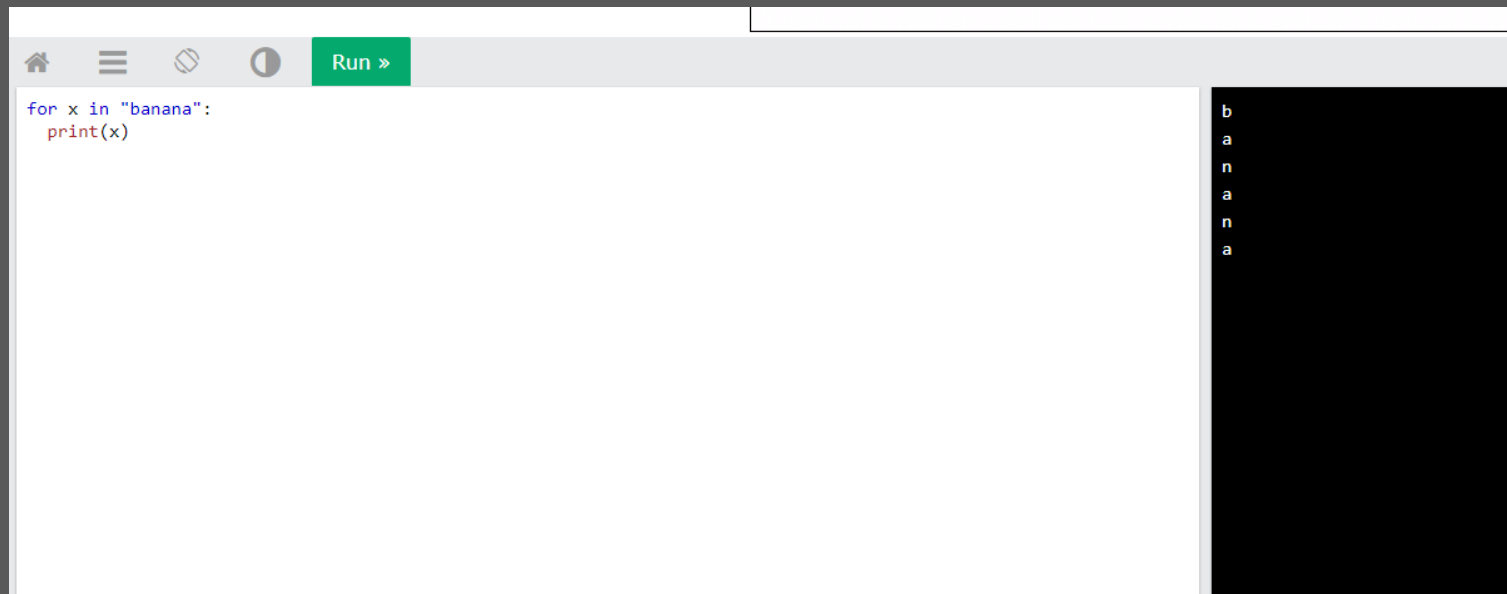
```
apple  
banana  
cherry
```

```
fruits = ["apple", "banana", "cherry"]  
for x in fruits:  
    print(x)
```

```
apple  
banana  
cherry
```


For Loop

Even strings are iterable objects, they contain a sequence of characters:

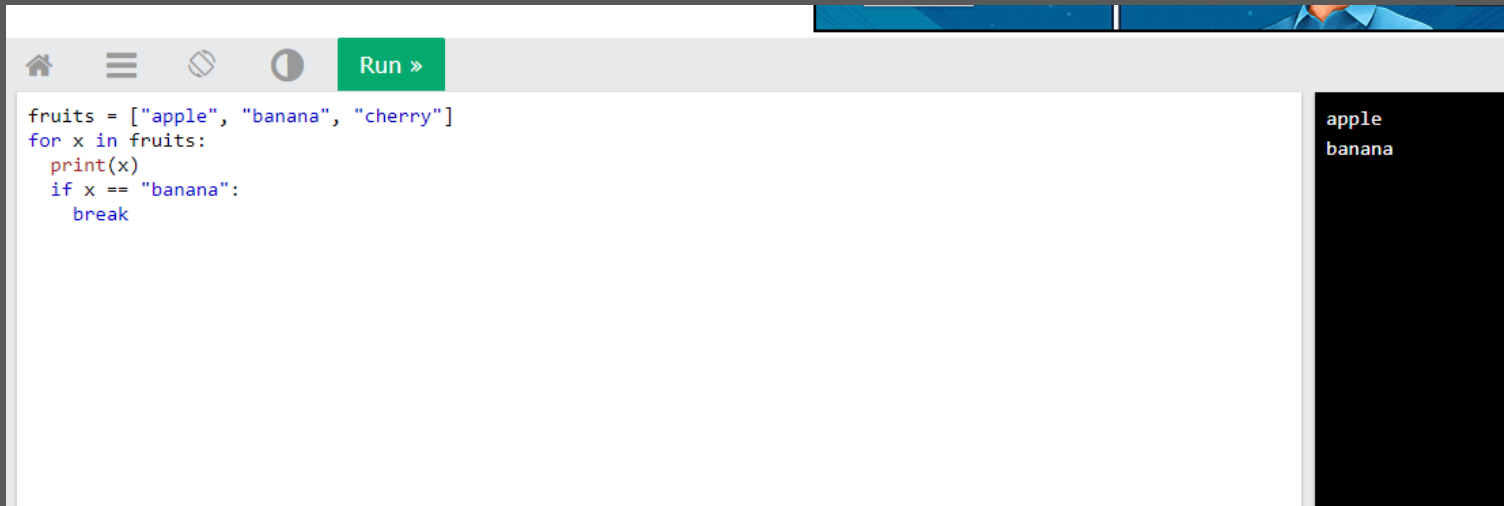


```
for x in "banana":  
    print(x)
```

The screenshot shows a Python IDE window. The code editor contains a for loop that iterates over the string "banana" and prints each character. The output console on the right shows the characters 'b', 'a', 'n', 'a', 'n', 'a' printed on separate lines.

For Loop

With the `break` statement we can stop the loop before it has looped through all the items:

A screenshot of a Python IDE interface. The top bar contains icons for home, menu, search, and a green 'Run »' button. The main editor area shows a Python script:

```
fruits = ["apple", "banana", "cherry"]  
for x in fruits:  
    print(x)  
    if x == "banana":  
        break
```

 The output console on the right shows the results of the execution:

```
apple  
banana
```

```
fruits = ["apple", "banana", "cherry"]  
for x in fruits:  
    print(x)  
    if x == "banana":  
        break
```

```
apple  
banana
```

For Loop



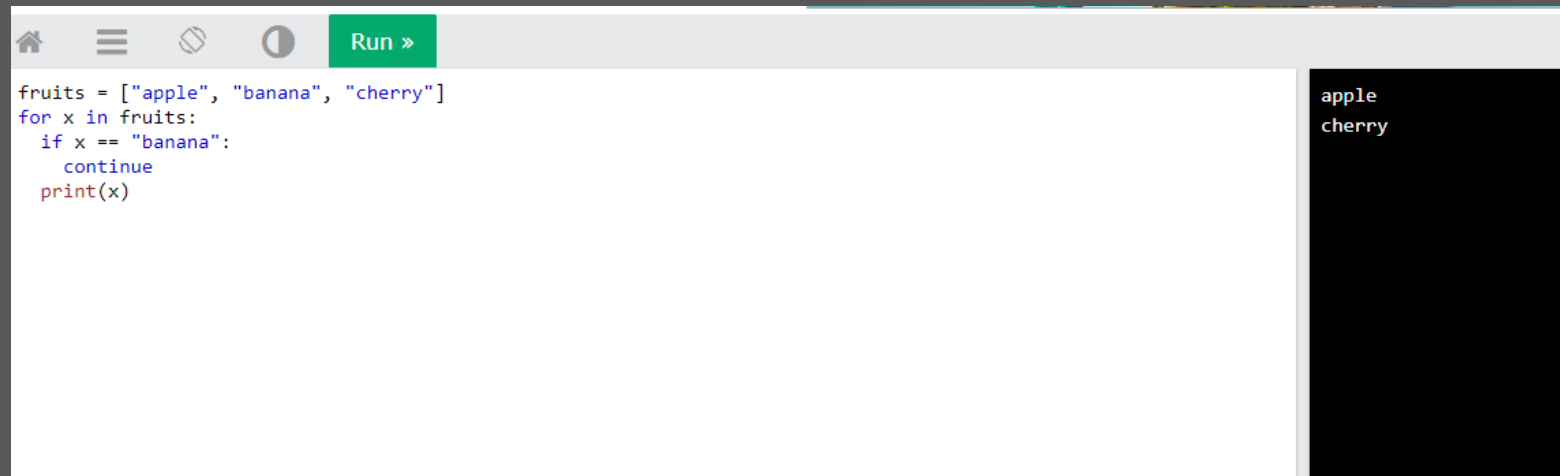
The image shows a Python IDE interface. At the top, there is a toolbar with icons for home, menu, undo, and redo, followed by a green 'Run »' button. Below the toolbar is a code editor with the following Python code:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    if x == "banana":
        break
    print(x)
```

The code is executed, and the output is displayed in a black console window on the right, showing the word 'apple'.

For Loop

With the continue statement we can stop the current iteration of the loop, and continue with the next:

A screenshot of a code editor window. The editor has a light gray header bar with icons for home, menu, search, and a green 'Run »' button. The main area is white and contains Python code. The code defines a list 'fruits' with 'apple', 'banana', and 'cherry'. It then uses a 'for' loop to iterate over 'fruits'. Inside the loop, there is an 'if' statement that checks if 'x' is equal to 'banana'. If true, it executes the 'continue' statement, which skips the rest of the loop body and moves to the next iteration. Finally, it prints 'x'. The output on the right shows 'apple' and 'cherry' on separate lines, confirming that 'banana' was skipped.

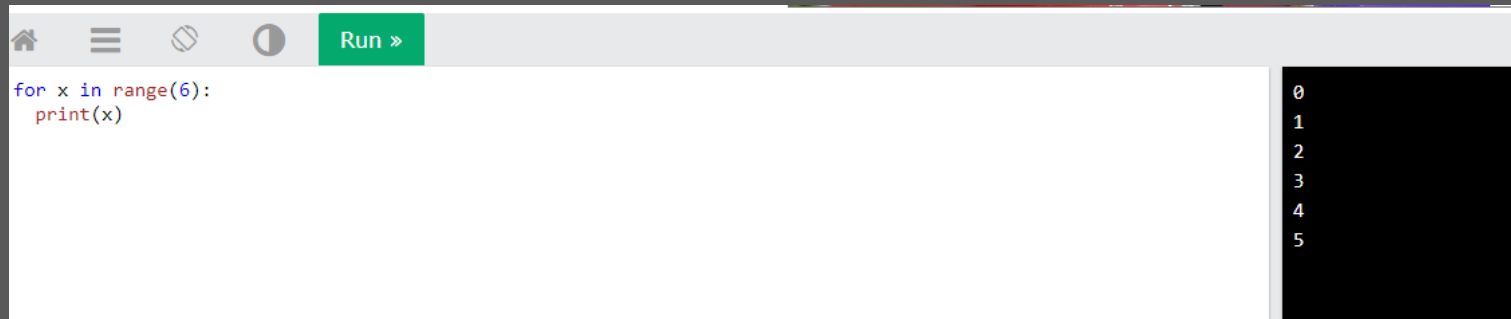
```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    if x == "banana":
        continue
    print(x)
```

apple
cherry

For Loop

To loop through a set of code a specified number of times, we can use the `range()` function. The `range()` function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

Note that `range(6)` is not the values of 0 to 6, but the values 0 to 5.



The screenshot shows a Python IDE interface. At the top, there is a toolbar with icons for home, menu, search, and a green 'Run »' button. Below the toolbar, the code editor contains the following Python code:

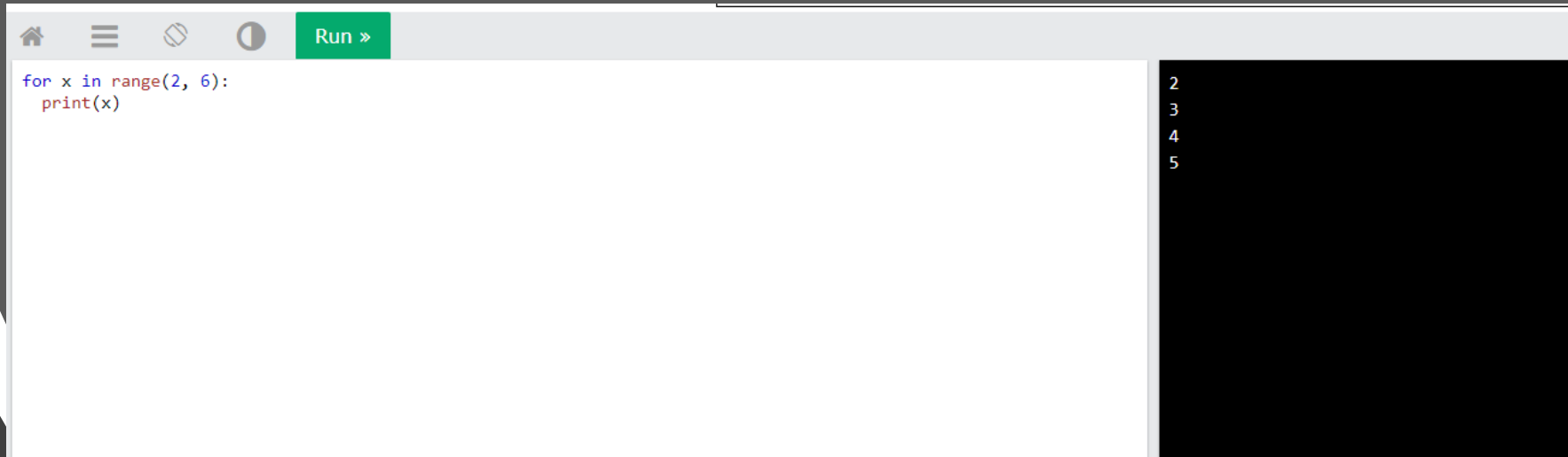
```
for x in range(6):  
    print(x)
```

To the right of the code editor, the output console displays the results of the loop, which are the numbers 0 through 5, each on a new line.

```
0  
1  
2  
3  
4  
5
```

For Loop

The `range()` function defaults to 0 as a starting value, however it is possible to specify the starting value by adding a parameter: `range(2, 6)`, which means values from 2 to 6 (but not including 6):



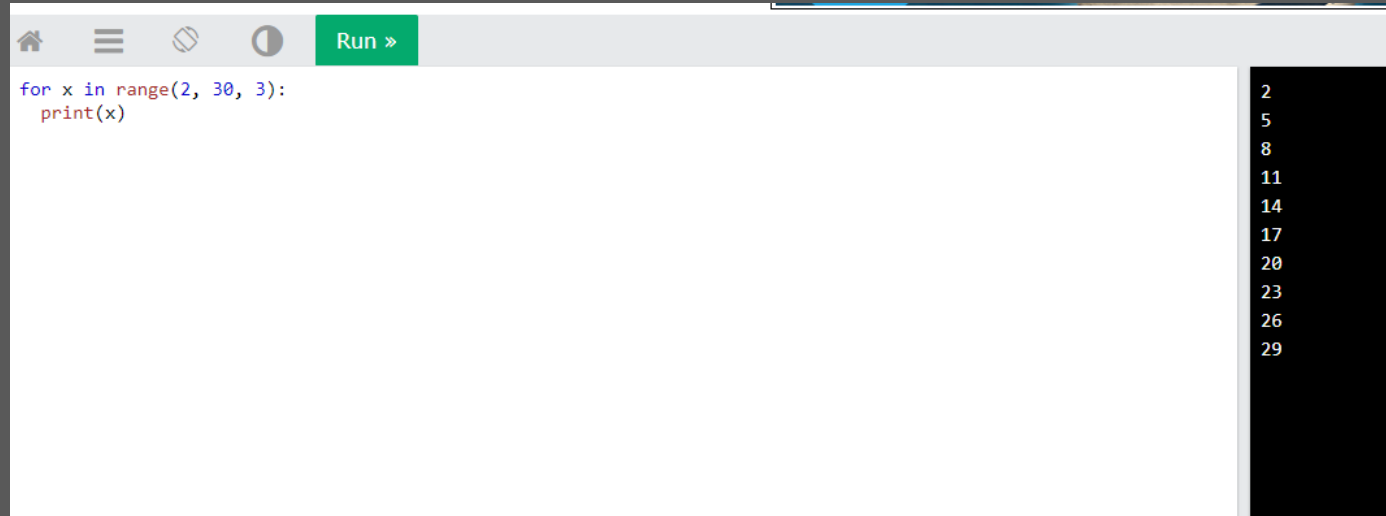
The screenshot shows a Python IDE window. The top bar contains icons for home, menu, undo, and redo, followed by a green 'Run »' button. The main editor area displays the following Python code:

```
for x in range(2, 6):  
    print(x)
```

To the right of the editor is a black output console. It displays the output of the code, which is the numbers 2, 3, 4, and 5, each on a new line.

For Loops

The range() function defaults to increment the sequence by 1, however it is possible to specify the increment value by adding a third parameter: range(2, 30, 3):



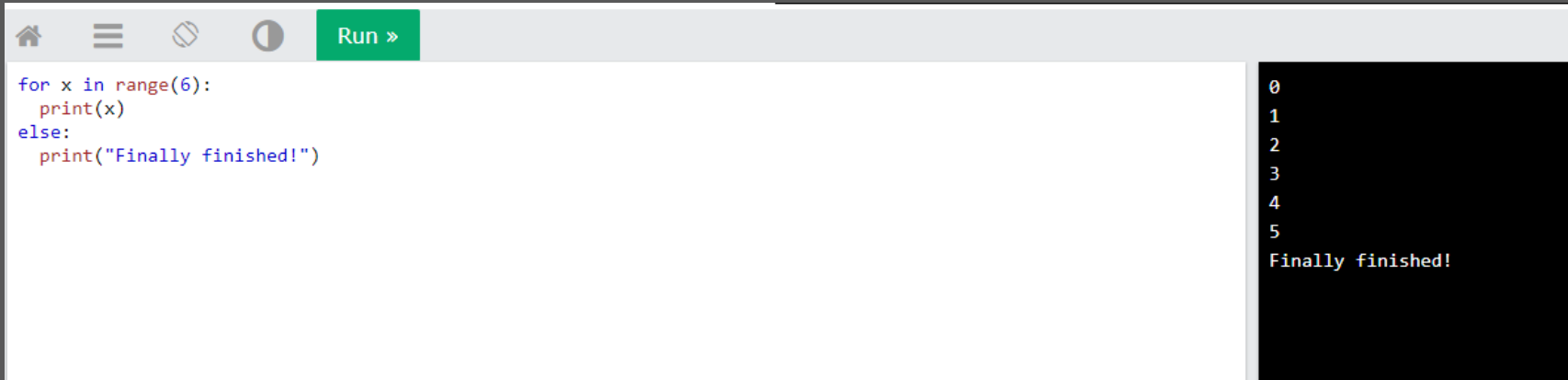
A screenshot of a Python IDE window. The window has a title bar with icons for home, menu, search, and a 'Run' button. The code editor contains the following Python code:

```
for x in range(2, 30, 3):  
    print(x)
```

The output of the code is displayed in a black console window on the right side of the IDE, showing the numbers 2, 5, 8, 11, 14, 17, 20, 23, 26, and 29, each on a new line.

For Loop

The else keyword in a for loop specifies a block of code to be executed when the loop is finished:
Note: The else block will NOT be executed if the loop is stopped by a break statement.



The screenshot shows a Python IDE window. The top bar contains icons for home, menu, search, and a green 'Run »' button. The main editor area displays the following Python code:

```
for x in range(6):  
    print(x)  
else:  
    print("Finally finished!")
```

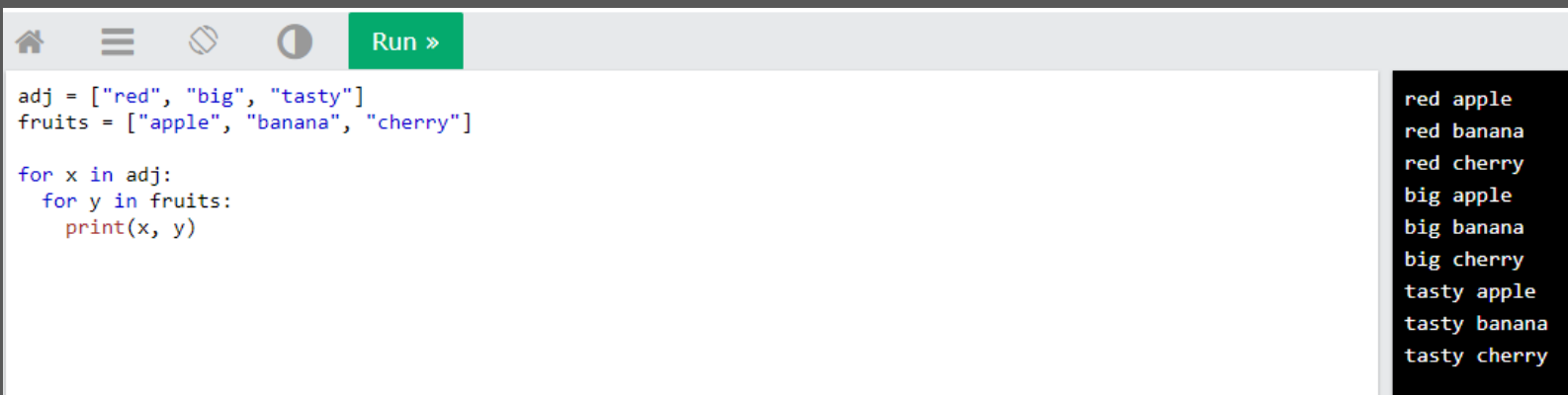
To the right of the editor is a black output console with white text showing the execution results:

```
0  
1  
2  
3  
4  
5  
Finally finished!
```


For Loops

A nested loop is a loop inside a loop.

The "inner loop" will be executed one time for each iteration of the "outer loop":



The image shows a code editor window with a light gray header bar containing icons for home, menu, search, and a play button, along with a green 'Run »' button. The main area has a white background and contains the following Python code:

```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]

for x in adj:
    for y in fruits:
        print(x, y)
```

To the right of the code editor is a black output console with white text showing the results of the nested loop:

```
red apple
red banana
red cherry
big apple
big banana
big cherry
tasty apple
tasty banana
tasty cherry
```

For Loops

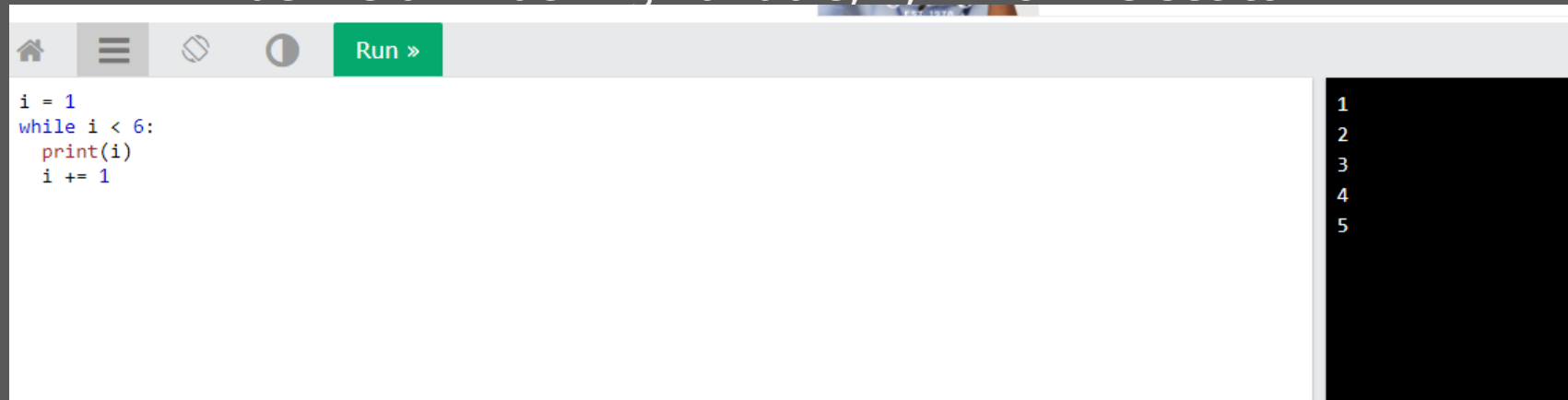
for loops cannot be empty, but if you for some reason you have a for loop with no content, put in the pass statement to avoid getting an error.

While Loops

With the while loop we can execute a set of statements as long as a condition is true.

remember to increment *i*, or else the loop will continue forever.

The while loop requires relevant variables to be ready, in this example we need to define an indexing variable, *i*, which we set to 1.

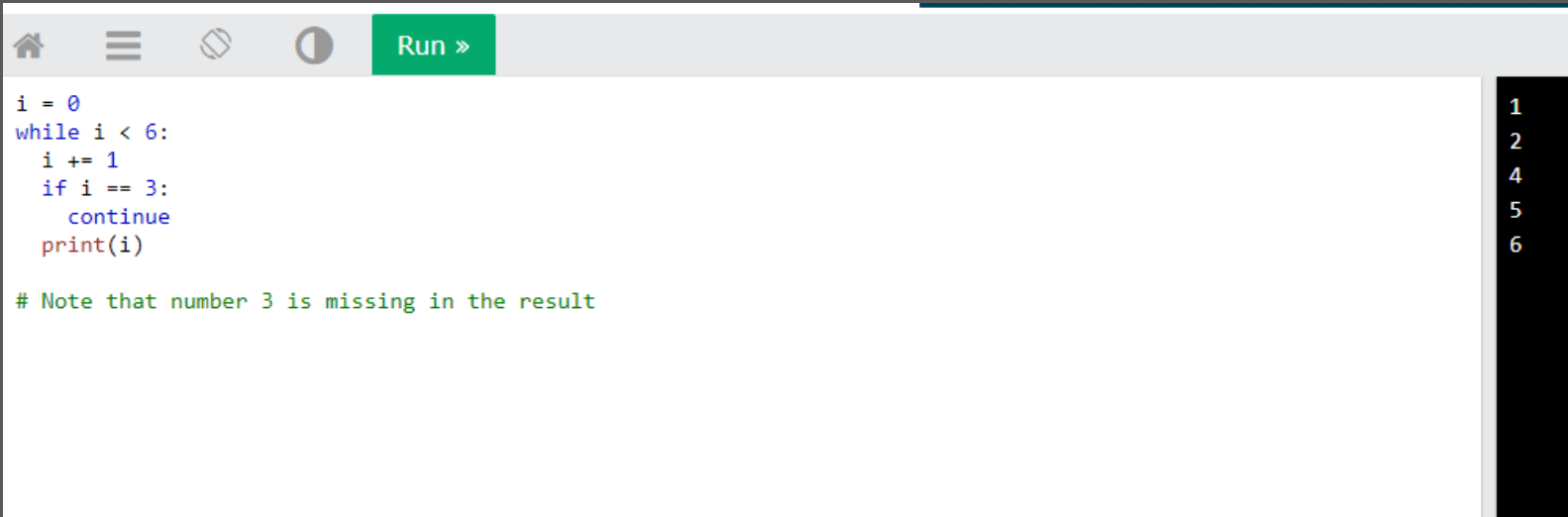
A screenshot of a code editor interface. The editor has a light gray header bar with icons for home, menu, search, and a green 'Run »' button. Below the header, the code is written in a white area. The code is a Python while loop that prints numbers 1 through 5. To the right of the code area, there is a black output console showing the numbers 1, 2, 3, 4, and 5 printed on separate lines.

```
i = 1
while i < 6:
    print(i)
    i += 1
```

```
1
2
3
4
5
```

While Loops

With the continue statement we can stop the current iteration, and continue with the next:



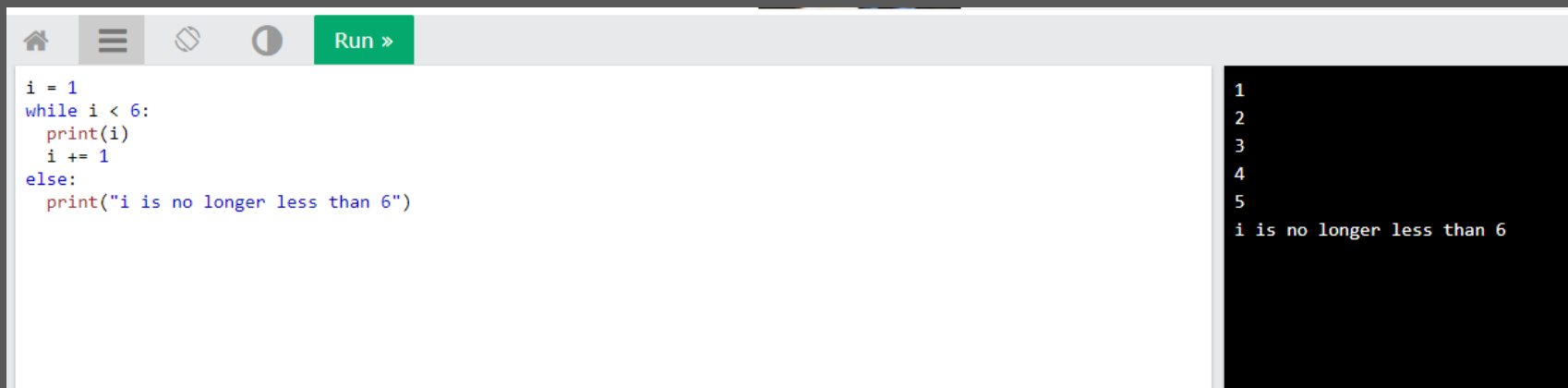
```
i = 0
while i < 6:
    i += 1
    if i == 3:
        continue
    print(i)

# Note that number 3 is missing in the result
```

The screenshot shows a code editor window with a toolbar at the top containing icons for home, menu, search, and a 'Run' button. The code is a Python while loop that increments a counter 'i' from 0 to 5. When 'i' reaches 3, the 'continue' statement is executed, skipping the 'print(i)' statement for that iteration. The output on the right shows the numbers 1, 2, 4, 5, and 6, with 3 missing.

While Loops

With the else statement we can run a block of code once when the condition no longer is true:



```
i = 1
while i < 6:
    print(i)
    i += 1
else:
    print("i is no longer less than 6")
```

The screenshot shows a code editor window with a toolbar at the top containing icons for home, menu, undo, redo, and a green 'Run »' button. The code in the editor is a Python while loop that prints the value of 'i' from 1 to 5, then prints a message when the loop ends. The output on the right shows the numbers 1 through 5, followed by the message 'i is no longer less than 6'.

While Loops

The While loop can be used the same way as a for loop

While Loops

The While loop can be used the same way as a for loop