String Comparison

The comparison operators also work on strings. To see if two strings are equal you simply write a boolean expression using the equality operator.

```
word = "banana"
if word == "banana":
    print("Yes, we have bananas!")
else:
    print("Yes, we have NO bananas!")

```

ActiveCode: 1 (ch08_comp1)

Run

```
Yes, we have bananas!
```

Other comparison operations are useful for putting words in lexicographical order (http://en.wikipedia.org/wiki/Lexicographic_order). This is similar to the alphabetical order you would use with a dictionary, except that all the uppercase letters come before all the lowercase letters.

```
word = "zebra"

if word < "banana":
    print("Your word, " + word + ", comes before banana.")
elif word > "banana":
    print("Your word, " + word + ", comes after banana.")
else:
    print("Yes, we have no bananas!")

```

ActiveCode: 2 (ch08_comp2)

Run

```
Your word, zebra, comes after banana.
```

It is probably clear to you that the word apple would be less than (come before) the word `banana` . After all, a is before b in the alphabet. But what if we consider the words `apple` and `Apple` ? Are they the same?

```
print("apple" < "banana")

print("apple" == "Apple")
print("apple" < "Apple")

```

ActiveCode: 3 (chp08_ord1)

Run

```
True
False
False
```

It turns out, as you recall from our discussion of variable names, that uppercase and lowercase letters are considered to be different from one another. The way the computer knows they are different is that each character is assigned a unique integer value. “A” is 65, “B” is 66, and “5” is 53. The way you can find out the so called **ordinal value** for a given character is to use a character function called `ord` .

```
print(ord("A"))
print(ord("B"))
print(ord("5"))

print(ord("a"))
print("apple" > "Apple")

```

ActiveCode: 4 (ch08_ord2)

Run

```
65
66
53
97
True
```

When you compare characters or strings to one another, Python converts the characters into their equivalent ordinal values and compares the integers from left to right. As you can see from the example above, “a” is greater than “A” so “apple” is greater than “Apple”.

Humans commonly ignore capitalization when comparing two words. However, computers do not. A common way to address this issue is to convert strings to a standard format, such as all lowercase, before performing the comparison.

There is also a similar function called `chr` that converts integers into their character equivalent.

```
print(chr(68))
print(chr(100))

print(chr(49))
print(chr(53))

print("The character for 32 is",chr(32),"!!!")
print(ord(" "))

```

ActiveCode: 5 (ch08_ord3)

Run

```
D
d
1
5
The character for 32 is   !!!
32
```

One thing to note in the last two examples is the fact that the space character has an ordinal value (32). Even though you don’t see it, it is an actual character. We sometimes call it a *nonprinting* character.

Check your understanding

str-11: Evaluate the following comparison:

```
"Dog" < "Doghouse"
```

● a) True

○ b) False

Check Me Compare Me

Correct!! Both match up to the g but Dog is shorter than Doghouse so it comes first in the dictionary.

str-12: Evaluate the following comparison:

```
"dog" < "Dog"
```

○ a) True

● b) False

○ c) They are the same word

Check Me Compare Me

Correct!! Yes, upper case is less than lower case according to the ordinal values of the characters.

str-13: Evaluate the following comparison:

```
"dog" < "Doghouse"
```

○ a) True

● b) False

Check Me Compare Me

Correct!! The length does not matter. Lower case d is greater than upper case D.

Strings are Immutable

One final thing that makes strings different from some other Python collection types is that you are not allowed to modify the individual characters in the collection. It is tempting to use the `[]` operator on the left side of an assignment, with the intention of changing a character in a string. For example, in the following code, we would like to change the first letter of `greeting`.

```
greeting = "Hello, world!"
greeting[0] = 'J'            # ERROR!
print(greeting)

```

ActiveCode: 6 (cg08_imm1)

Run

Instead of producing the output `Jello, world!`, this code produces the runtime error `TypeError: 'str' object does not support item assignment`.

Strings are **immutable**, which means you cannot change an existing string. The best you can do is create a new string that is a variation on the original.

```
greeting = "Hello, world!"
newGreeting = 'J' + greeting[1:]
print(newGreeting)
print(greeting)          # same as it was

```

ActiveCode: 7 (ch08_imm2)

Run

```
Jello, world!
Hello, world!
```

The solution here is to concatenate a new first letter onto a slice of `greeting`. This operation has no effect on the original string.

Check your understanding

str-14: What is printed by the following statements:

```
s = "Ball"
s[0] = "C"
print(s)
```

○ a) Ball

○ b) Call

● c) Error

Check Me Compare Me

Correct!! Yes, strings are immutable.

Traversal and the `for` Loop: By Item

A lot of computations involve processing a collection one item at a time. For strings this means that we would like to process one character at a time. Often we start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a **traversal**.

We have previously seen that the `for` statement can iterate over the items of a sequence (a list of names in the case below).

```
for aname in ["Joe", "Amy", "Brad", "Angelina", "Zuki", "Thandi", "Paris"]:
    invitation = "Hi " + aname + ". Please come to my party on Saturday!"
    print(invitation)

```

ActiveCode: 8 (ch08_4)

Run

```
Hi Joe. Please come to my party on Saturday!
Hi Amy. Please come to my party on Saturday!
Hi Brad. Please come to my party on Saturday!
Hi Angelina. Please come to my party on Saturday!
Hi Zuki. Please come to my party on Saturday!
Hi Thandi. Please come to my party on Saturday!
Hi Paris. Please come to my party on Saturday!
```

Recall that the loop variable takes on each value in the sequence of names. The body is performed once for each name. The same was true for the sequence of integers created by the `range` function.

```
for avalue in range(10):
    print(avalue)

```

ActiveCode: 9 (ch08_5)

Run

```
0
1
2
3
4
5
6
7
8
9
```

Since a string is simply a sequence of characters, the `for` loop iterates over each character automatically.

```
for achar in "Go Spot Go":
    print(achar)

```

ActiveCode: 10 (ch08_6)

Run

```
G
o

S
p
o
t

G
o
```

The loop variable `achar` is automatically reassigned each character in the string “Go Spot Go”. We will refer to this type of sequence iteration as **iteration by item**. Note that it is only possible to process the characters one at a time from left to right.

Check your understanding

str-15: How many times is the word HELLO printed by the following statements?

```
s = "python rocks"
for ch in s:
   print("HELLO")
```

○ a) 10

○ b) 11

● c) 12

○ d) Error, the for statement needs to use the range function.

Check Me Compare Me

Correct!! Yes, there are 12 characters, including the blank.

str-16: How many times is the word HELLO printed by the following statements?

```
s = "python rocks"
for ch in s[3:8]:
   print("HELLO")
```

○ a) 4

● b) 5

○ c) 6

○ d) Error, the for statement cannot use slice.

Check Me | Compare Me

Correct!! Yes, The blank is part of the sequence returned by slice

Traversal and the `for` Loop: By Index

It is also possible to use the `range` function to systematically generate the indices of the characters. The `for` loop can then be used to iterate over these positions. These positions can be used together with the indexing operator to access the individual characters in the string.

Consider the following codelens example.

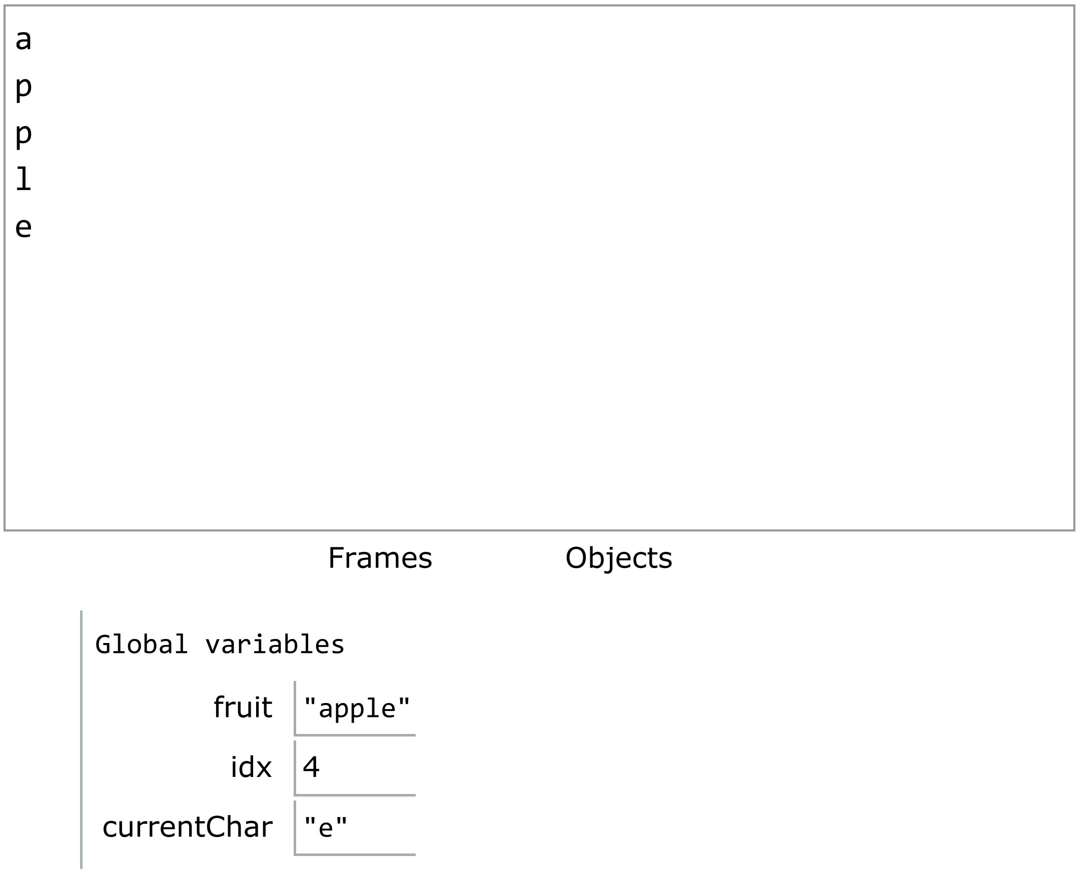
```
fruit = "apple"
for idx in range(5):
    currentChar = fruit[idx]
    print(currentChar)
```

<< First | < Back | Program terminated | Forward > | Last >>

➡ line that has just executed

➡ next line to execute

Program output:



```
a
p
p
l
e
```

Global variables

fruit	"apple"
idx	4
currentChar	"e"

CodeLens: 1 (ch08_7)

The index positions in “apple” are 0,1,2,3 and 4. This is exactly the same sequence of integers returned by `range(5)` . The first time through the for loop, `idx` will be 0 and the “a” will be printed. Then, `idx` will be reassigned to 1 and “p” will be displayed. This will repeat for all the range values up to but not including 5. Since “e” has index 4, this will be exactly right to show all of the characters.

In order to make the iteration more general, we can use the `len` function to provide the bound for `range` . This is a very common pattern for traversing any sequence by position. Make sure you understand why the range function behaves correctly when using `len` of the string as its parameter value.

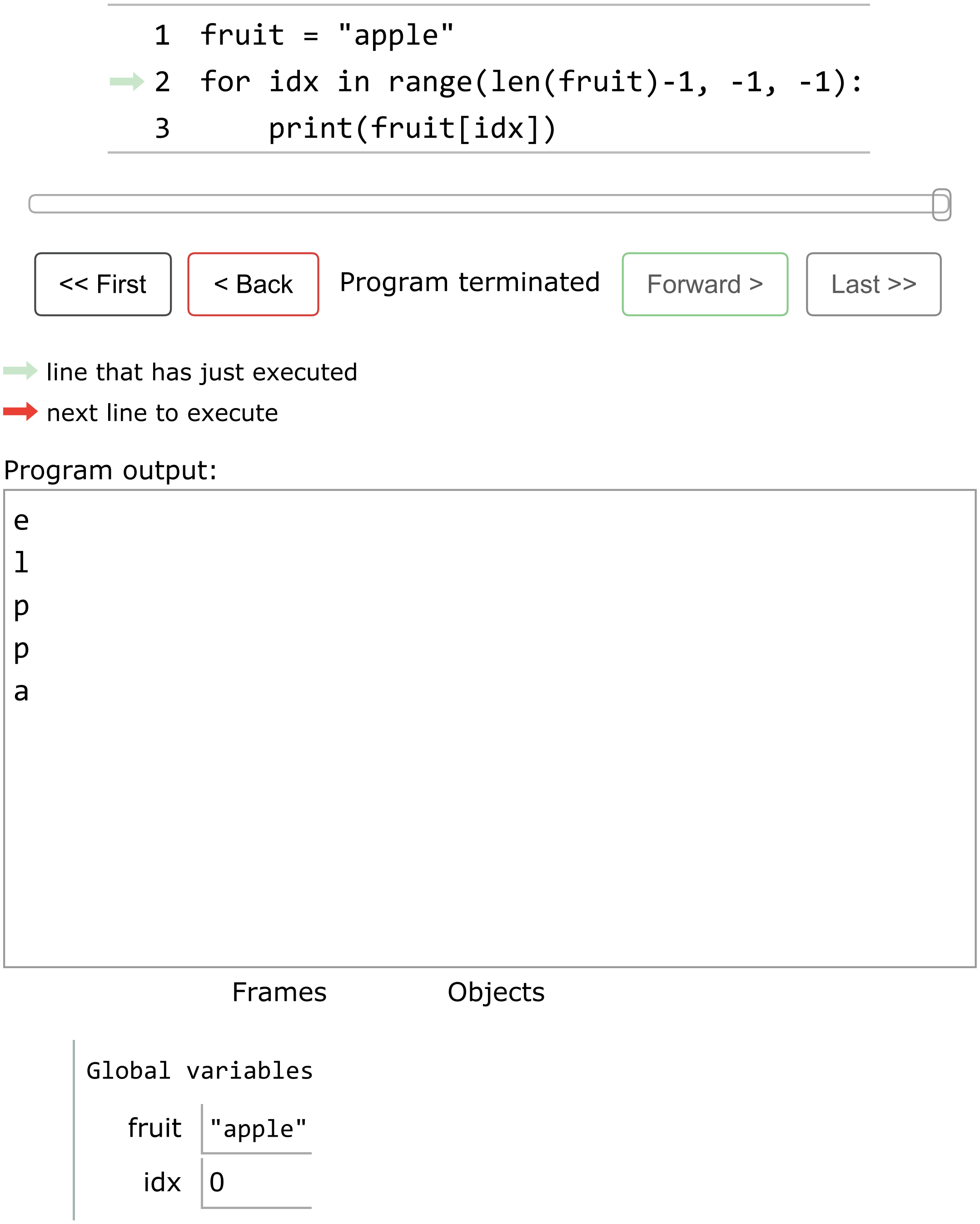
```python
fruit = "apple"
for idx in range(len(fruit)):
    print(fruit[idx])

```

ActiveCode: 11 (ch08_7b)

Run

```
a
p
p
l
e
```

You may also note that iteration by position allows the programmer to control the direction of the traversal by changing the sequence of index values. Recall that we can create ranges that count down as well as up so the following code will print the characters from right to left.



```
fruit = "apple"
for idx in range(len(fruit)-1, -1, -1):
    print(fruit[idx])
```

➡ line that has just executed

➡ next line to execute

Program output:

```
e
l
p
p
a
```

Frames Objects

Global variables

fruit	"apple"
idx	0

CodeLens: 2 (ch08_8)

Trace the values of `idx` and satisfy yourself that they are correct. In particular, note the start and end of the range.

Check your understanding

str-17: How many times is the letter o printed by the following statements?

```
s = "python rocks"
for idx in range(len(s)):
   if idx % 2 == 0:
      print(s[idx])
```

○ a) 0

○ b) 1

● c) 2

○ d) Error, the for statement cannot have an if inside.

Check Me Compare Me

Correct!! Yes, it will print all the characters in even index positions and the o character appears both times in an even location.

Traversal and the `while` Loop

The `while` loop can also control the generation of the index values. Remember that the programmer is responsible for setting up the initial condition, making sure that the condition is correct, and making sure that something changes inside the body to guarantee that the condition will eventually fail.

```
fruit = "apple"

position = 0
while position < len(fruit):
    print(fruit[position])
    position = position + 1

```

ActiveCode: 12 (ch08_7c)

Run

```
a
p
p
l
e
```

The loop condition is `position < len(fruit)`, so when `position` is equal to the length of the string, the condition is false, and the body of the loop is not executed. The last character accessed is the one with the index `len(fruit)-1`, which is the last character in the string.

Here is the same example in codelens so that you can trace the values of the variables.

```
fruit = "apple"

position = 0
while position < len(fruit):
    print(fruit[position])
    position = position + 1
```

<< First | < Back | Program terminated | Forward > | Last >>

➡ line that has just executed

➡ next line to execute

Program output:

```
a
p
p
l
e
```

Frames Objects

Global variables

fruit	"apple"
position	5

CodeLens: 3 (ch08_7c1)

Check your understanding

str-18: How many times is the letter o printed by the following statements?

```
s = "python rocks"
idx = 1
while idx < len(s):
   print(s[idx])
   idx = idx + 2
```

● a) 0

○ b) 1

○ c) 2

Check Me Compare Me

Correct!! Yes, idx goes thru the odd numbers starting at 1. o is at position 4 and 8.

Note

This workspace is provided for your convenience. You can use this activecode window to try out anything you like.

```

```

ActiveCode: 13 (scratch_08_02)

Run

The `in` and `not in` operators

The `in` operator tests if one string is a substring of another:

```
print('p' in 'apple')
print('i' in 'apple')
print('ap' in 'apple')
print('pa' in 'apple')

```

ActiveCode: 14 (chp8_in1)

Run

```
True
False
True
False
```

Note that a string is a substring of itself, and the empty string is a substring of any other string. (Also note that computer scientists like to think about these edge cases quite carefully!)

```
print('a' in 'a')
print('apple' in 'apple')
print('' in 'a')
print('' in 'apple')

```

ActiveCode: 15 (chp8_in2)

Run

```
True
True
True
True
```

The `not in` operator returns the logical opposite result of `in` .

```
print('x' not in 'apple')

```

ActiveCode: 16 (chp8_in3)

Run

```
True
```

The Accumulator Pattern with Strings

Combining the `in` operator with string concatenation using `+` and the accumulator pattern, we can write a function that removes all the vowels from a string. The idea is to start with a string and iterate over each character, checking to see if the character is a vowel. As we process the characters, we will build up a new string consisting of only the nonvowel characters. To do this, we use the accumulator pattern.

Remember that the accumulator pattern allows us to keep a “running total”. With strings, we are not accumulating a numeric total. Instead we are accumulating characters onto a string.

```
 1 def removeVowels(s):
 2     vowels = "aeiouAEIOU"
 3     sWithoutVowels = ""
 4     for eachChar in s:
 5         if eachChar not in vowels:
 6             sWithoutVowels = sWithoutVowels + eachChar
 7     return sWithoutVowels
 8
 9 print(removeVowels("compsci"))
10 print(removeVowels("aAbEefIijOopUus"))
11
```

ActiveCode: 17 (ch08_acc1)

Run

```
cmpsc
bfjps
```

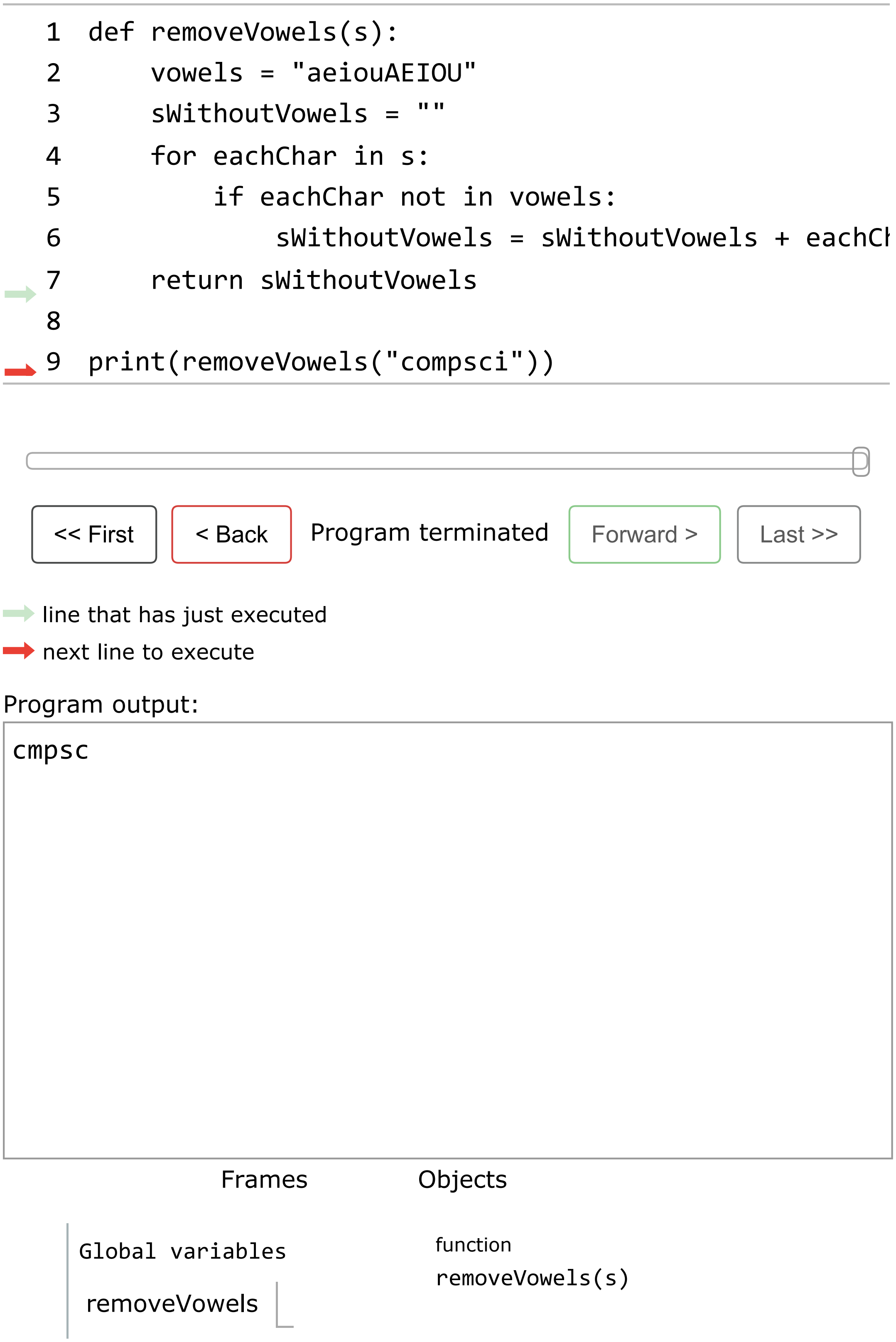
Line 5 uses the `not in` operator to check whether the current character is not in the string `vowels`. The alternative to using this operator would be to write a very large `if` statement that checks each of the individual vowel characters. Note we would need to use logical `and` to be sure that the character is not any of the vowels.

```
if eachChar != 'a'  and eachChar != 'e'  and eachChar != 'i'  and
   eachChar != 'o'  and eachChar != 'u'  and eachChar != 'A'  and
   eachChar != 'E'  and eachChar != 'I'  and eachChar != 'O'  and
   eachChar != 'U':

     sWithoutVowels = sWithoutVowels + eachChar
```

Look carefully at line 6 in the above program (`sWithoutVowels = sWithoutVowels + eachChar`). We will do this for every character that is not a vowel. This should look very familiar. As we were describing earlier, it is an example of the accumulator pattern, this time using a string to “accumulate” the final result. In words it says that the new value of `sWithoutVowels` will be the old value of `sWithoutVowels` concatenated with the value of `eachChar`. We are building the result string character by character.

Take a close look also at the initialization of `sWithoutVowels`. We start with an empty string and then begin adding new characters to the end.

Step thru the function using codelens to see the accumulator variable grow.



```
def removeVowels(s):
    vowels = "aeiouAEIOU"
    sWithoutVowels = ""
    for eachChar in s:
        if eachChar not in vowels:
            sWithoutVowels = sWithoutVowels + eachCh
    return sWithoutVowels

print(removeVowels("compsci"))
```

Program output:

```
cmpsc
```

CodeLens: 4 (ch08_acc2)

Check your understanding

str-19: What is printed by the following statements:

```
s = "ball"
r = ""
for item in s:
   r = item.upper() + r
print(r)
```

○ a) Ball

○ b) BALL

● c) LLAB

Check Me | Compare Me

Correct!! Yes, the order is reversed due to the order of the concatenation.

Note

This workspace is provided for your convenience. You can use this activecode window to try out anything you like.

ActiveCode: 18 (scratch_08_03)

Run

Error

```
TypeError: 'str' does not support item assignment on line 2
```

Description

Type errors most often occur when an expression tries to combine two objects with types that should not be combined. Like raising a string to a power

To Fix

To fix a type error you will most likely need to trace through your code and make sure the variables have the types you expect them to have. It may be helpful to print out each variable along the way to be sure its value is what you think it should be.