

Network Programming in Python I

Justin Ellis MBA

Review

Any Questions?

Strings

9.2. A Collection Data Type

So far we have seen built-in types like: `int`, `float`, `bool`, `str` and we've seen lists. `int`, `float`, and `bool` are considered to be simple or primitive data types because their values are not composed of any smaller parts. They cannot be broken down. On the other hand, strings and lists are different from the others because they are made up of smaller pieces. In the case of strings, they are made up of smaller strings each containing one **character**.

Types that are comprised of smaller pieces are called **collection data types**. Depending on what we are doing, we may want to treat a collection data type as a single entity (the whole), or we may want to access its parts. This ambiguity is useful.

Strings can be defined as sequential collections of characters. This means that the individual characters that make up the string are assumed to be in a particular order from left to right.

A string that contains no characters, often referred to as the **empty string**, is still considered to be a string. It is simply a sequence of zero characters and is represented by `"` or `""` (two single or two double quotes with nothing in between).

Strings

9.3. Operations on Strings

In general, you cannot perform mathematical operations on strings, even if the strings look like numbers. The following are illegal (assuming that `message` has type string):

```
message - 1
"Hello" / 123
message * "Hello"
"15" + 2
```

Interestingly, the `+` operator does work with strings, but for strings, the `+` operator represents **concatenation**, not addition. Concatenation means joining the two operands by linking them end-to-end.

Strings



Save & Run Original - 1 of 1 Show in CodeLens Share Code

```
1 fruit = "banana"
2 bakedGood = " nut bread"
3 print(fruit + bakedGood)
4
```

banana nut bread

Activity: 9.3.1 ActiveCode (ch08_add)

The output of this program is `banana nut bread`. The space before the word `nut` is part of the string and is necessary to produce the space between the concatenated strings. Take out the space and run it again.

Strings

The `*` operator also works on strings. It performs repetition. For example, `'Fun'*3` is `'FunFunFun'`. One of the operands has to be a string and the other has to be an integer.

Save & Run

Original - 1 of 1

Show in CodeLens

Share Code

```
1 print("Go" * 6)
2
3 name = "Packers"
4 print(name * 3)
5
6 print(name + "Go" * 3)
7
8 print((name + "Go") * 3)
9
```

```
GoGoGoGoGoGo
PackersPackersPackers
PackersGoGoGoGo
PackersGoPackersGoPackersGo
```

Activity: 9.3.2 ActiveCode (ch08_mult)

This interpretation of `+` and `*` makes sense by analogy with addition and multiplication. Just as `4*3` is equivalent to `4+4+4`, we expect `"Go"*3` to be the same as `"Go"+"Go"+"Go"`, and it is. Note also in the last example that the order of operations for `*` and `+` is the same as it was for arithmetic. The repetition is done before the concatenation. If you want to cause the concatenation to be done first, you will need to use parenthesis.

Strings

The **indexing operator** (Python uses square brackets to enclose the index) selects a single character from a string. The characters are accessed by their position or index value. For example, in the string shown below, the 14 characters are indexed left to right from position 0 to position 13.

0	1	2	3	4	5	6	7	8	9	10	11	12	13
L	u	t	h	e	r		C	o	l	l	e	g	e
-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

It is also the case that the positions are named from right to left using negative numbers where -1 is the rightmost index and so on. Note that the character at index 6 (or -8) is the blank character.

Strings



The screenshot shows a code editor with a yellow background. At the top, there is a 'Save & Run' button, a progress indicator showing 'Original - 1 of 1', a 'Show in CodeLens' button, and a 'Share Code' button. The code is as follows:

```
1 school = "Luther College"
2 m = school[2]
3 print(m)
4
5 lastchar = school[-1]
6 print(lastchar)
7
```

Below the code editor, there is a terminal window showing the output:

```
t
e
```

At the bottom of the editor, it says 'Activity: 9.4.1 ActiveCode (chp08_index1)'.

The expression `school[2]` selects the character at index 2 from `school`, and creates a new string containing just this one character. The variable `m` refers to the result.

Remember that computer scientists often start counting from zero. The letter at index zero of `"Luther College"` is `L`. So at position `[2]` we have the letter `t`.

If you want the zero-eth letter of a string, you just put 0, or any expression with the value 0, in the brackets. Give it a try.

The expression in brackets is called an **index**. An index specifies a member of an ordered collection. In this case the collection of characters in the string. The index *indicates* which character you want. It can be any integer expression so long as it evaluates to a valid index value.

Note that indexing returns a *string* — Python has no special type for a single character. It is just a string of length 1.

Strings

We previously saw that each turtle instance has its own attributes and a number of methods that can be applied to the instance. For example, we wrote `tess.right(90)` when we wanted the turtle object `tess` to perform the `right` method to turn to the right 90 degrees. The "dot notation" is the way we connect the name of an object to the name of a method it can perform.

Strings are also objects. Each string instance has its own attributes and methods. The most important attribute of the string is the collection of characters. There are a wide variety of methods. Try the following program.



Save & Run Original - 1 of 1 Show in CodeLens Share Code

```
1 ss = "Hello, World"
2 print(ss.upper())
3
4 tt = ss.lower()
5 print(tt)
6
```

HELLO, WORLD
hello, world

Activity: 9.5.1 ActiveCode (chp08_upper)

Strings

In this example, `upper` is a method that can be invoked on any string object to create a new string in which all the characters are in uppercase. `lower` works in a similar fashion changing all characters in the string to lowercase. (The original string `ss` remains unchanged. A new string `tt` is created.)

In addition to `upper` and `lower`, the following table provides a summary of some other useful string methods. There are a few activecode examples that follow so that you can try them out.

Method	Parameters	Description
<code>upper</code>	none	Returns a string in all uppercase
<code>lower</code>	none	Returns a string in all lowercase
<code>capitalize</code>	none	Returns a string with first character capitalized, the rest lower
<code>strip</code>	none	Returns a string with the leading and trailing whitespace removed
<code>lstrip</code>	none	Returns a string with the leading whitespace removed
<code>rstrip</code>	none	Returns a string with the trailing whitespace removed
<code>count</code>	item	Returns the number of occurrences of item
<code>replace</code>	old, new	Replaces all occurrences of old substring with new
<code>center</code>	width	Returns a string centered in a field of width spaces
<code>ljust</code>	width	Returns a string left justified in a field of width spaces
<code>rjust</code>	width	Returns a string right justified in a field of width spaces
<code>find</code>	item	Returns the leftmost index where the substring item is found, or -1 if not found
<code>rfind</code>	item	Returns the rightmost index where the substring item is found, or -1 if not found
<code>index</code>	item	Like find except causes a runtime error if item is not found
<code>rindex</code>	item	Like rfind except causes a runtime error if item is not found
<code>format</code>	substitutions	Involved! See String Format Method , below

You should experiment with these methods so that you understand what they do. Note once again that the methods that return strings do not change the original. You can also consult the [Python documentation for strings](#).

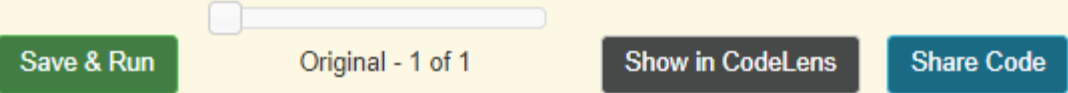
Strings

9.5.1. String Format Method

In grade school quizzes a common convention is to use fill-in-the blanks. For instance,

Hello ____!

and you can fill in the name of the person greeted, and combine given text with a chosen insertion. *We use this as an analogy:* Python has a similar construction, better called fill-in-the-braces. The string method `format`, makes substitutions into places in a string enclosed in braces. Run this code:



A code editor interface with a yellow background. At the top, there is a slider control and three buttons: 'Save & Run' (green), 'Original - 1 of 1' (grey), 'Show in CodeLens' (dark grey), and 'Share Code' (teal). Below the buttons is a code editor with a white background and a light grey line number margin on the left.

```
1 person = input('Your name: ')
2 greeting = 'Hello {}'.format(person)
3 print(greeting)
4
```

Strings

There are several new ideas here!

The string for the `format` method has a special form, with braces embedded. Such a string is called a *format string*. Places where braces are embedded are replaced by the value of an expression taken from the parameter list for the `format` method. There are many variations on the syntax between the braces. In this case we use the syntax where the first (and only) location in the string with braces has a substitution made from the first (and only) parameter.

In the code above, this new string is assigned to the identifier `greeting`, and then the string is printed.

The identifier `greeting` was introduced to break the operations into a clearer sequence of steps. However, since the value of `greeting` is only referenced once, it can be eliminated with the more concise version:



The screenshot shows an ActiveCode editor interface. At the top, there are buttons for 'Save & Run', 'Original - 1 of 1', 'Show in CodeLens', and 'Share Code'. Below these buttons is a code editor with the following Python code:

```
1 person = input('Enter your name: ')
2 print('Hello {}'.format(person))
3
```

Below the code editor is a text area displaying the output of the code: 'Hello Justin!'.

Strings

There are several new ideas here!

The string for the `format` method has a special form, with braces embedded. Such a string is called a *format string*. Places where braces are embedded are replaced by the value of an expression taken from the parameter list for the `format` method. There are many variations on the syntax between the braces. In this case we use the syntax where the first (and only) location in the string with braces has a substitution made from the first (and only) parameter.

In the code above, this new string is assigned to the identifier `greeting`, and then the string is printed.

The identifier `greeting` was introduced to break the operations into a clearer sequence of steps. However, since the value of `greeting` is only referenced once, it can be eliminated with the more concise version:



The screenshot shows an ActiveCode editor interface. At the top, there are buttons for 'Save & Run', 'Original - 1 of 1', 'Show in CodeLens', and 'Share Code'. Below these buttons is a code editor with the following Python code:

```
1 person = input('Enter your name: ')
2 print('Hello {}'.format(person))
3
```

Below the code editor is a text area displaying the output of the code: 'Hello Justin!'.

Strings

There are several new ideas here!

The string for the `format` method has a special form, with braces embedded. Such a string is called a *format string*. Places where braces are embedded are replaced by the value of an expression taken from the parameter list for the `format` method. There are many variations on the syntax between the braces. In this case we use the syntax where the first (and only) location in the string with braces has a substitution made from the first (and only) parameter.

In the code above, this new string is assigned to the identifier `greeting`, and then the string is printed.

The identifier `greeting` was introduced to break the operations into a clearer sequence of steps. However, since the value of `greeting` is only referenced once, it can be eliminated with the more concise version:



The screenshot shows a web-based code editor interface. At the top, there are four buttons: "Save & Run" (green), "Original - 1 of 1" (light gray), "Show in CodeLens" (dark gray), and "Share Code" (teal). Below the buttons is a text area containing the following Python code:

```
1 person = input('Enter your name: ')
2 print('Hello {}'.format(person))
3
```

Below the code area is a gray box representing the output, which contains the text "Hello Justin!".

Strings

There can be multiple substitutions, with data of any type. Next we use floats. Try original price \$2.50 with a 7% discount:



The screenshot shows a code editor interface with a yellow background. At the top, there are buttons for 'Save & Run', 'Original - 1 of 1', 'Show in CodeLens', and 'Share Code'. Below these is a code editor with the following Python code:

```
1 origPrice = float(input('Enter the original price: $'))
2 discount = float(input('Enter discount percentage: '))
3 newPrice = (1 - discount/100)*origPrice
4 calculation = '{} discounted by {}% is {}'.format(origPrice, discount, newPrice)
5 print(calculation)
6
```

Below the code editor is a terminal window showing the output:

```
$2.5 discounted by 7.0% is $2.325.
```

At the bottom of the editor, it says 'Activity: 9.5.1.3 ActiveCode (ch08_methods5)'.

The parameters are inserted into the braces in order.

If you used the data suggested, this result is not satisfying. Prices should appear with exactly two places beyond the decimal point, but that is not the default way to display floats.

Format strings can give further information inside the braces showing how to specially format data. In particular floats can be shown with a specific number of decimal places. For two decimal places, put `:.2f` inside the braces for the monetary values:

Strings

There can be multiple substitutions, with data of any type. Next we use floats. Try original price \$2.50 with a 7% discount:



The screenshot shows a code editor interface with a yellow background. At the top, there are buttons for 'Save & Run', 'Original - 1 of 1', 'Show in CodeLens', and 'Share Code'. Below these is a code editor with the following Python code:

```
1 origPrice = float(input('Enter the original price: $'))
2 discount = float(input('Enter discount percentage: '))
3 newPrice = (1 - discount/100)*origPrice
4 calculation = '{} discounted by {}% is {}'.format(origPrice, discount, newPrice)
5 print(calculation)
6
```

Below the code editor is a terminal window showing the output:

```
$2.5 discounted by 7.0% is $2.325.
```

At the bottom of the editor, it says 'Activity: 9.5.1.3 ActiveCode (ch08_methods5)'.

The parameters are inserted into the braces in order.

If you used the data suggested, this result is not satisfying. Prices should appear with exactly two places beyond the decimal point, but that is not the default way to display floats.

Format strings can give further information inside the braces showing how to specially format data. In particular floats can be shown with a specific number of decimal places. For two decimal places, put `:.2f` inside the braces for the monetary values:

Strings

inside the braces for the monetary values.

Save & Run

Original - 1 of 1

Show in CodeLens

Share Code

```
1 origPrice = float(input('Enter the original price: $'))
2 discount = float(input('Enter discount percentage: '))
3 newPrice = (1 - discount/100)*origPrice
4 calculation = '${:.2f} discounted by {}% is ${:.2f}.'.format(origPrice, dis
5 print(calculation)
6
```

\$2.50 discounted by 7.0% is \$2.32.

Activity: 9.5.1.4 ActiveCode (ch08_methods6)

Strings

9.6. Length

The `len` function, when applied to a string, returns the number of characters in a string.

Save & Run

Original - 1 of 1

Show in CodeLens

Share Code

```
1 fruit = "Banana"
2 print(len(fruit))
3
```

6

Strings

9.6. Length

The `len` function, when applied to a string, returns the number of characters in a string.

Save & Run

Original - 1 of 1

Show in CodeLens

Share Code

```
1 fruit = "Banana"
2 print(len(fruit))
3
```

6

Strings

Alternatively in Python, we can use **negative indices**, which count backward from the end of the string. The expression `fruit[-1]` yields the last letter, `fruit[-2]` yields the second to last, and so on. Try it! Most other languages do *not* allow the negative indices, but they are a handy feature of Python!

Shantanu Sinha

Strings

9.7. The Slice Operator

A substring of a string is called a **slice**. Selecting a slice is similar to selecting a character:



The screenshot shows a code editor interface with a yellow background. At the top, there is a slider control, a "Save & Run" button, and labels "Original - 1 of 1", "Show in CodeLens", and "Share Code". The code editor contains the following Python code:

```
1 singers = "Peter, Paul, and Mary"
2 print(singers[0:5])
3 print(singers[7:11])
4 print(singers[17:21])
5
```

Below the code editor, the output is displayed in a grey box:

```
Peter
Paul
Mary
```

At the bottom of the editor, it says "Activity: 9.7.1 ActiveCode (chp08_slice1)".

The *slice* operator `[n:m]` returns the part of the string from the *n*'th character to the *m*'th character, including the first but excluding the last. In other words, start with the character at index *n* and go up to but do not include the character at index *m*. This behavior may seem counter-intuitive but if you recall the `range` function, it did not include its end point either.

If you omit the first index (before the colon), the slice starts at the beginning of the string. If you omit the second index, the slice goes to the end of the string.

There is no `Index Out Of Range` exception for a slice. A slice is forgiving and shifts any offending index to something legal.

Strings

9.8. 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.



The screenshot shows a code editor with a yellow background. At the top, there is a slider control, a green 'Save & Run' button, the text 'Original - 1 of 1', a dark grey 'Show in CodeLens' button, and a blue 'Share Code' button. The code editor contains the following Python code:

```
1 word = "banana"
2 if word == "banana":
3     print("Yes, we have bananas!")
4 else:
5     print("Yes, we have NO bananas!")
6
```

Below the code editor, there is a grey output box containing the text: "Yes, we have bananas!"

Strings

Other comparison operations are useful for putting words in [lexicographical 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.

Save & Run

Original - 1 of 1

Show in CodeLens

Share Code

```
1 word = "zebra"
2
3 if word < "banana":
4     print("Your word, " + word + ", comes before banana.")
5 elif word > "banana":
6     print("Your word, " + word + ", comes after banana.")
7 else:
8     print("Yes, we have no bananas!")
9
```

Your word, zebra, comes after banana.

Activity: 9.8.2 ActiveCode (ch08_comp2)

Strings

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?



The screenshot shows an ActiveCode editor interface. At the top, there is a 'Save & Run' button, a progress indicator showing 'Original - 1 of 1', and 'Show in CodeLens' and 'Share Code' buttons. The code editor contains the following Python code:

```
1 print("apple" < "banana")
2
3 print("apple" == "Apple")
4 print("apple" < "Apple")
5
```

Below the code editor, the output is displayed in a grey box:

```
True
False
False
```

At the bottom of the editor, the text 'Activity: 9.8.3 ActiveCode (chp08_ord1)' is visible.

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`.

Strings



The screenshot shows a code editor interface with a yellow background. At the top, there is a slider control, a "Save & Run" button, and a "Show in CodeLens" button. The code is as follows:

```
1 print(ord("A"))
2 print(ord("B"))
3 print(ord("5"))
4
5 print(ord("a"))
6 print("apple" > "Apple")
7
```

Below the code editor, the output is displayed in a grey box:

```
65
66
53
97
True
```

At the bottom of the editor, it says "Activity: 9.8.4 ActiveCode (ch08_ord2)".

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.

Strings

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

Save & Run

Original - 1 of 1

Show in CodeLens

Share Code

```
1 print(chr(65))
2 print(chr(66))
3
4 print(chr(49))
5 print(chr(53))
6
7 print("The character for 32 is", chr(32), "!!!")
8 print(ord(" "))
9
```

```
A
B
1
5
The character for 32 is  !!!
32
```

Activity: 9.8.5 ActiveCode (ch08_ord3)

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.

Strings

9.9. 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`.

Save & Run

Original - 1 of 1

Show in CodeLens

Share Code

```
1 greeting = "Hello, world!"
2 greeting[0] = 'J'           # ERROR!
3 print(greeting)
4
```

Activity: 9.9.1 ActiveCode (cg08_imm1)

Error

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

Strings

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.

The screenshot shows a code editor interface with a yellow background. At the top, there is a slider and three buttons: "Save & Run" (green), "Original - 1 of 1" (grey), "Show in CodeLens" (dark grey), and "Share Code" (blue). The code editor contains the following Python code:

```
1 greeting = "Hello, world!"
2 newGreeting = 'J' + greeting[1:]
3 print(newGreeting)
4 print(greeting)           # same as it was
5
```

Below the code editor, there is a grey box containing the output of the program:

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

At the bottom of the editor, it says "Activity: 9.9.2 ActiveCode (ch08_imm2)".

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

Strings

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.



The screenshot shows a code editor interface with a yellow background. At the top, there is a slider control and three buttons: "Save & Run" (green), "Original - 1 of 1" (grey), "Show in CodeLens" (dark grey), and "Share Code" (blue). The code editor contains the following Python code:

```
1 greeting = "Hello, world!"
2 newGreeting = 'J' + greeting[1:]
3 print(newGreeting)
4 print(greeting)           # same as it was
5
```

Below the code editor, the output is displayed in a grey box:

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

At the bottom of the editor, the text "Activity: 9.9.2 ActiveCode (ch08_imm2)" is visible.

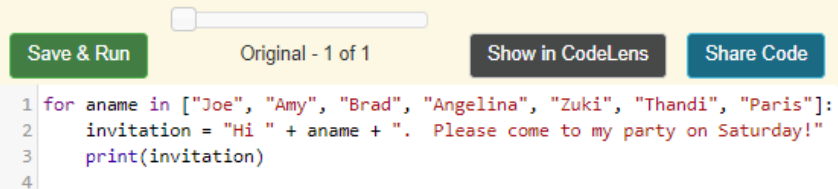
The solution here is to concatenate a new first letter onto a slice of `greeting`. This operation has no effect on the original string.

Strings

9.10. 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).



Save & Run Original - 1 of 1 Show in CodeLens Share Code

```
1 for aname in ["Joe", "Amy", "Brad", "Angelina", "Zuki", "Thandi", "Paris"]:  
2     invitation = "Hi " + aname + ". Please come to my party on Saturday!"  
3     print(invitation)  
4
```

```
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!
```

Strings

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.

Save & Run

Original - 1 of 1

Show in CodeLens

Share Code


```
1 for avalue in range(10):
2     print(avalue)
3
```

0
1
2
3
4
5
6
7
8
9

Activity: 9.10.2 ActiveCode (ch08_5)

Strings

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



The screenshot shows a code editor interface with a yellow background. At the top, there is a slider and three buttons: "Save & Run" (green), "Original - 1 of 1" (grey), "Show in CodeLens" (dark grey), and "Share Code" (teal). The code editor contains the following Go code:

```
1 for achar in "Go Spot Go":  
2     print(achar)  
3
```

Below the code editor, the output is displayed in a grey box, showing the characters of the string "Go Spot Go" printed one by one on separate lines:

```
G  
o  
  
S  
p  
o  
t  
  
G  
o
```

At the bottom of the editor, the text "Activity: 9.10.3 ActiveCode (ch08_6)" is visible.

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.

Strings

9.11. 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 code example.

The screenshot shows the Python Tutor CodeLens interface. On the left, a code editor displays the following code:

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

Line 2 is highlighted with a green arrow, indicating it is the line that just executed. A red arrow points to line 3, indicating the next line to execute. Below the code editor is a progress bar and two buttons: "< Prev" and "Next >".

On the right, a window titled "Print output (drag lower right corner to resize)" shows the output of the program:

```
p
p
l
e
```

Below the print output window are two tabs: "Frames" and "Objects". The "Frames" tab is selected, showing the following frame:

Global frame	
fruit	"apple"
idx	4
currentChar	"e"

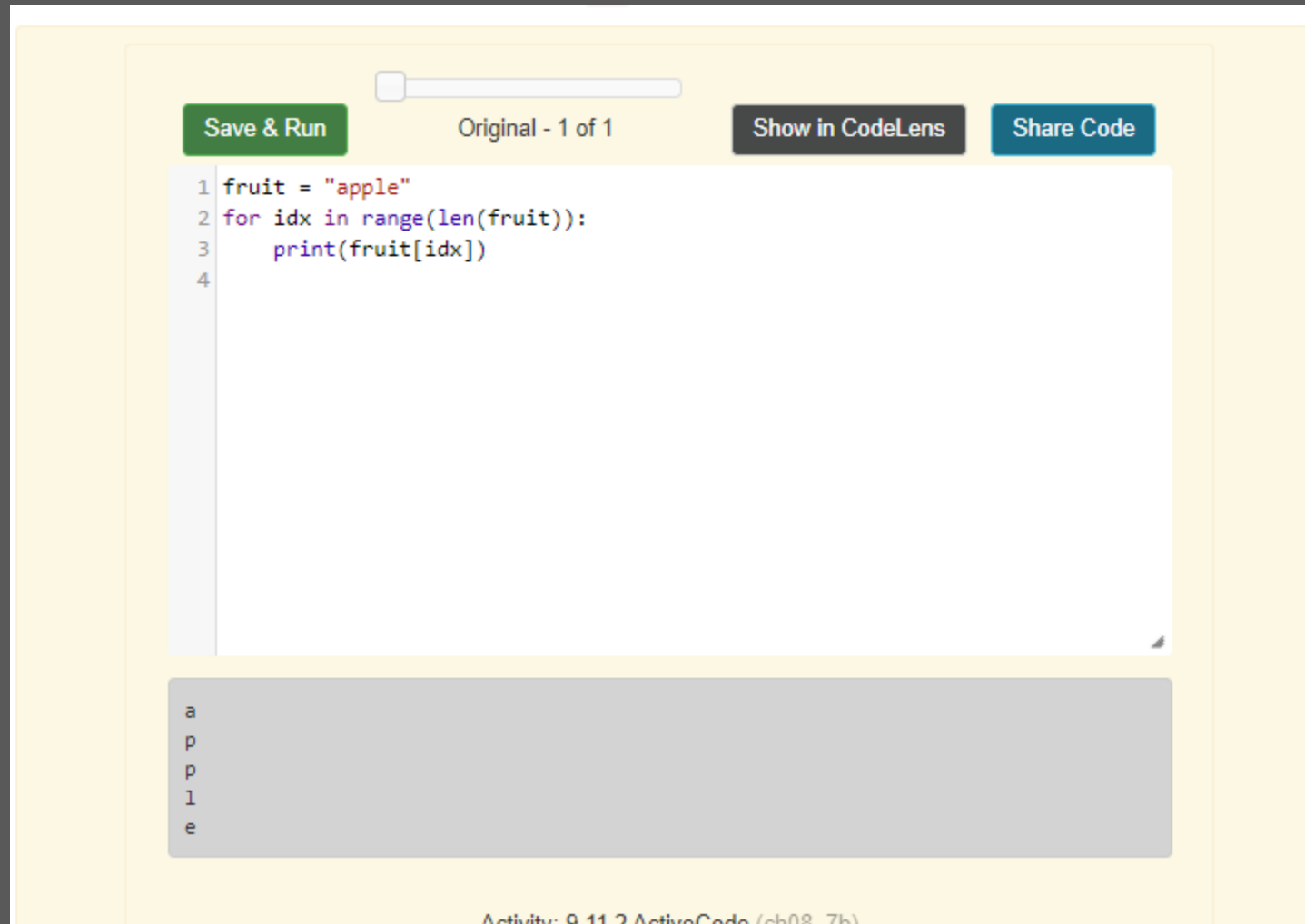
At the bottom of the interface, it says "Done running (17 steps)", "Python Tutor by Philip Guo", and "Customize visualization (NEW!)"

Activity: CodeLens 9.11.1 (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.

Strings



The screenshot shows a code editor interface with a yellow background. At the top, there is a slider control, a green "Save & Run" button, the text "Original - 1 of 1", a dark grey "Show in CodeLens" button, and a blue "Share Code" button. The code editor contains the following Python code:

```
1 fruit = "apple"
2 for idx in range(len(fruit)):
3     print(fruit[idx])
4
```

Below the code editor, the output is displayed in a grey box, showing the characters of the string "apple" printed on separate lines:

```
a
p
p
l
e
```

At the bottom of the editor, the text "Activity: 9.11.2 ActiveCode (ch08_7b)" is visible.

Strings

9.12. 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.



The screenshot shows a code editor interface with a yellow background. At the top, there is a slider control, a 'Save & Run' button, and labels 'Original - 1 of 1', 'Show in CodeLens', and 'Share Code'. The code is as follows:

```
1 fruit = "apple"
2
3 position = 0
4 while position < len(fruit):
5     print(fruit[position])
6     position = position + 1
7
```

Below the code editor, the output is displayed in a grey box, showing the characters of the string 'apple' printed on separate lines:

```
a
p
p
l
e
```

Activity: 9.12.1 ActiveCode (ch08_7c)

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.

Strings

9.13. The `in` and `not in` operators

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

Save & Run

Original - 1 of 1

Show in CodeLens

Share Code

```
1 print('p' in 'apple')
2 print('i' in 'apple')
3 print('ap' in 'apple')
4 print('pa' in 'apple')
5
```

```
True
False
True
False
```

Activity: 9.13.1 ActiveCode (chp8_in1)

Strings

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

Save & Run

Original - 1 of 1

Show in CodeLens

Share Code

```
1 print('x' not in 'apple')
2
```

True

Activity: 9.13.3 ActiveCode (chp8_in3)