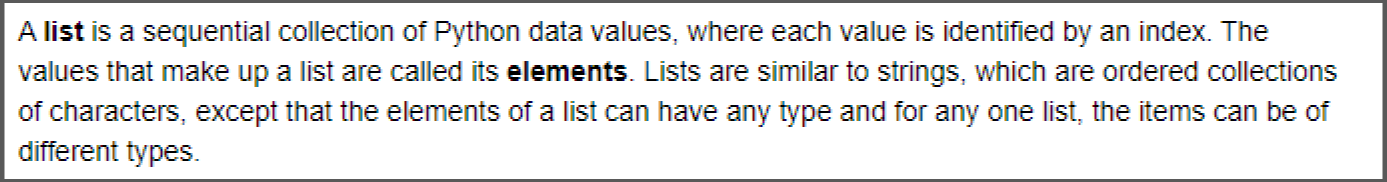
Network Programming in Python I

Justin Ellis MBA

Review

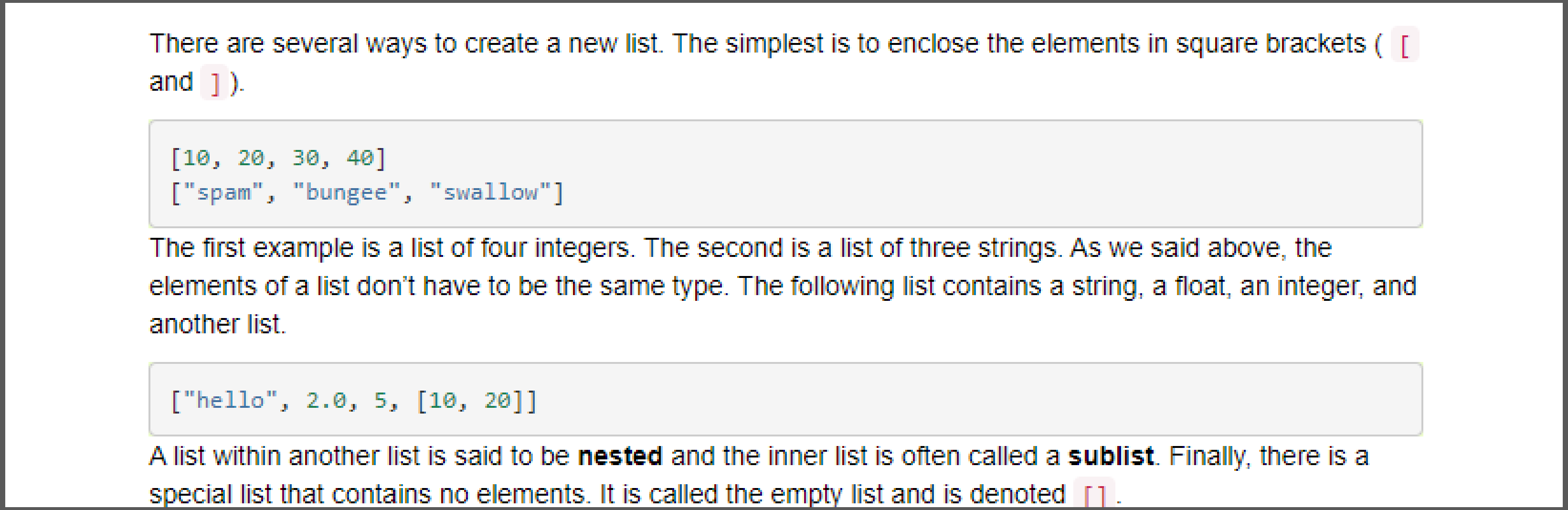
Any Questions?

Lists



A **list** is a sequential collection of Python data values, where each value is identified by an index. The values that make up a list are called its **elements**. Lists are similar to strings, which are ordered collections of characters, except that the elements of a list can have any type and for any one list, the items can be of different types.

Lists



There are several ways to create a new list. The simplest is to enclose the elements in square brackets (`[` and `]`).

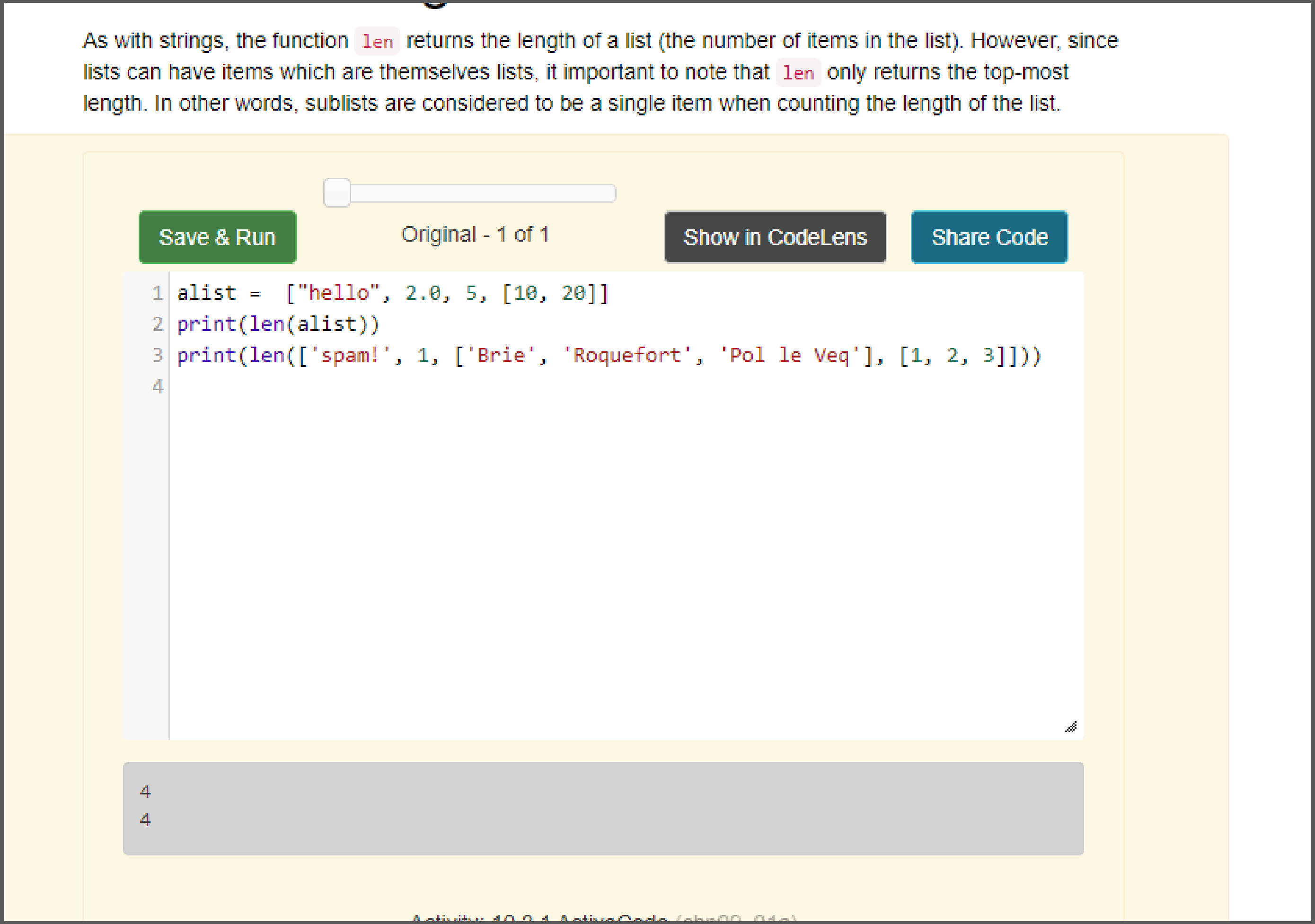
```
[10, 20, 30, 40]
["spam", "bungee", "swallow"]
```

The first example is a list of four integers. The second is a list of three strings. As we said above, the elements of a list don't have to be the same type. The following list contains a string, a float, an integer, and another list.

```
["hello", 2.0, 5, [10, 20]]
```

A list within another list is said to be **nested** and the inner list is often called a **sublist**. Finally, there is a special list that contains no elements. It is called the empty list and is denoted `[]`.

Lists

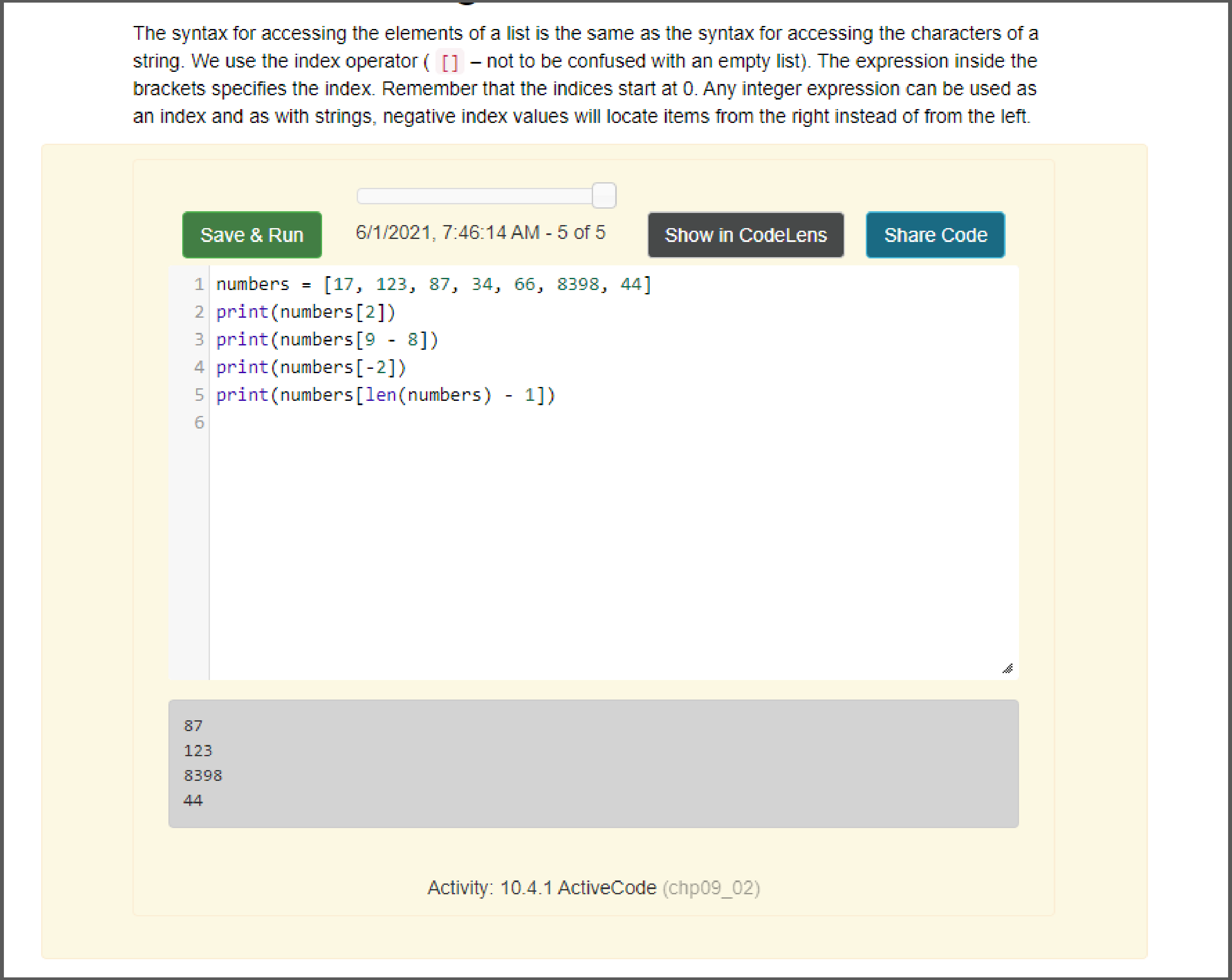


As with strings, the function `len` returns the length of a list (the number of items in the list). However, since lists can have items which are themselves lists, it important to note that `len` only returns the top-most length. In other words, sublists are considered to be a single item when counting the length of the list.

```python
alist =  ["hello", 2.0, 5, [10, 20]]
print(len(alist))
print(len(['spam!', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]))
```

```
4
4
```

Lists



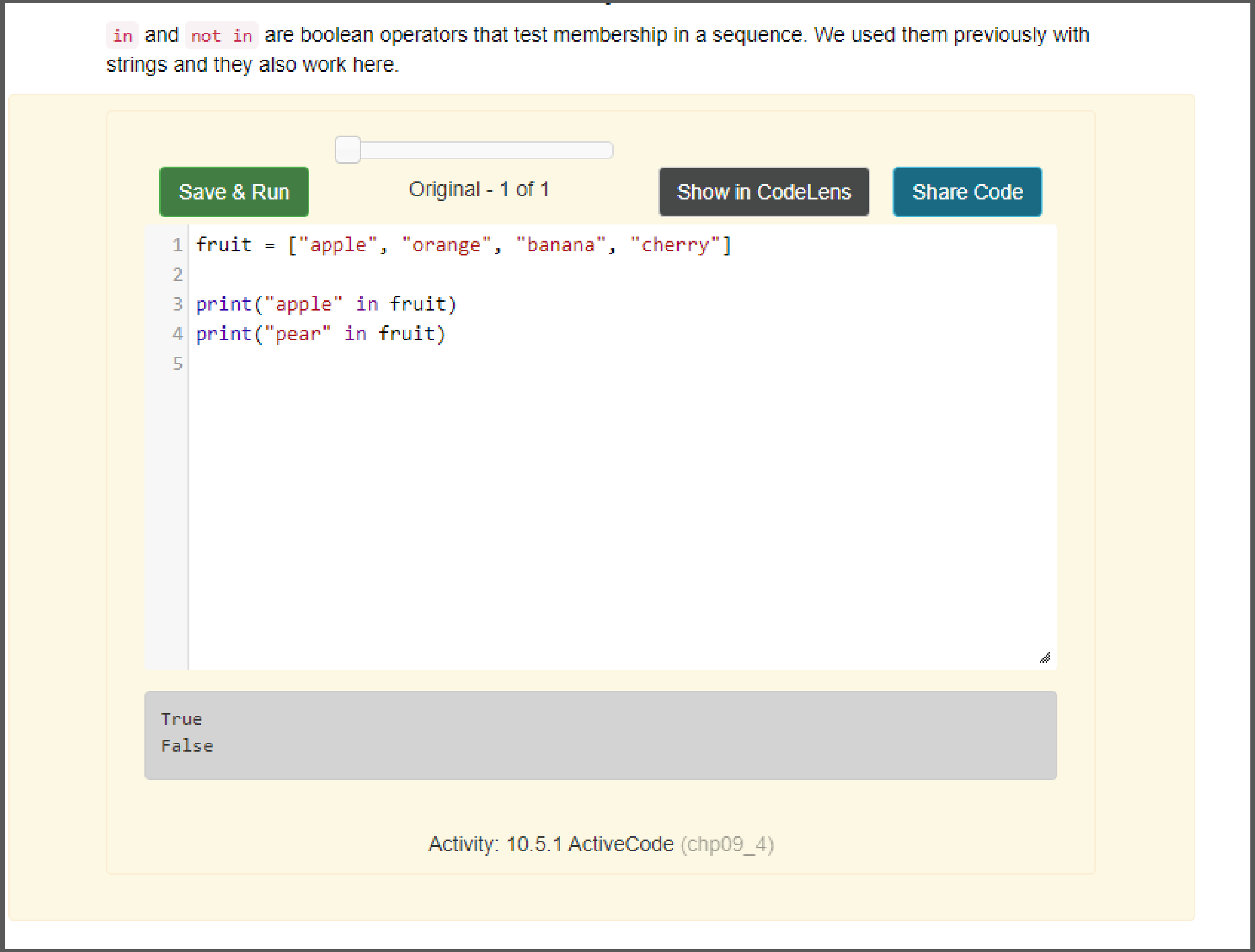
The syntax for accessing the elements of a list is the same as the syntax for accessing the characters of a string. We use the index operator (`[]` – not to be confused with an empty list). The expression inside the brackets specifies the index. Remember that the indices start at 0. Any integer expression can be used as an index and as with strings, negative index values will locate items from the right instead of from the left.

```python
numbers = [17, 123, 87, 34, 66, 8398, 44]
print(numbers[2])
print(numbers[9 - 8])
print(numbers[-2])
print(numbers[len(numbers) - 1])
```

```
87
123
8398
44
```

Activity: 10.4.1 ActiveCode (chp09_02)

Lists



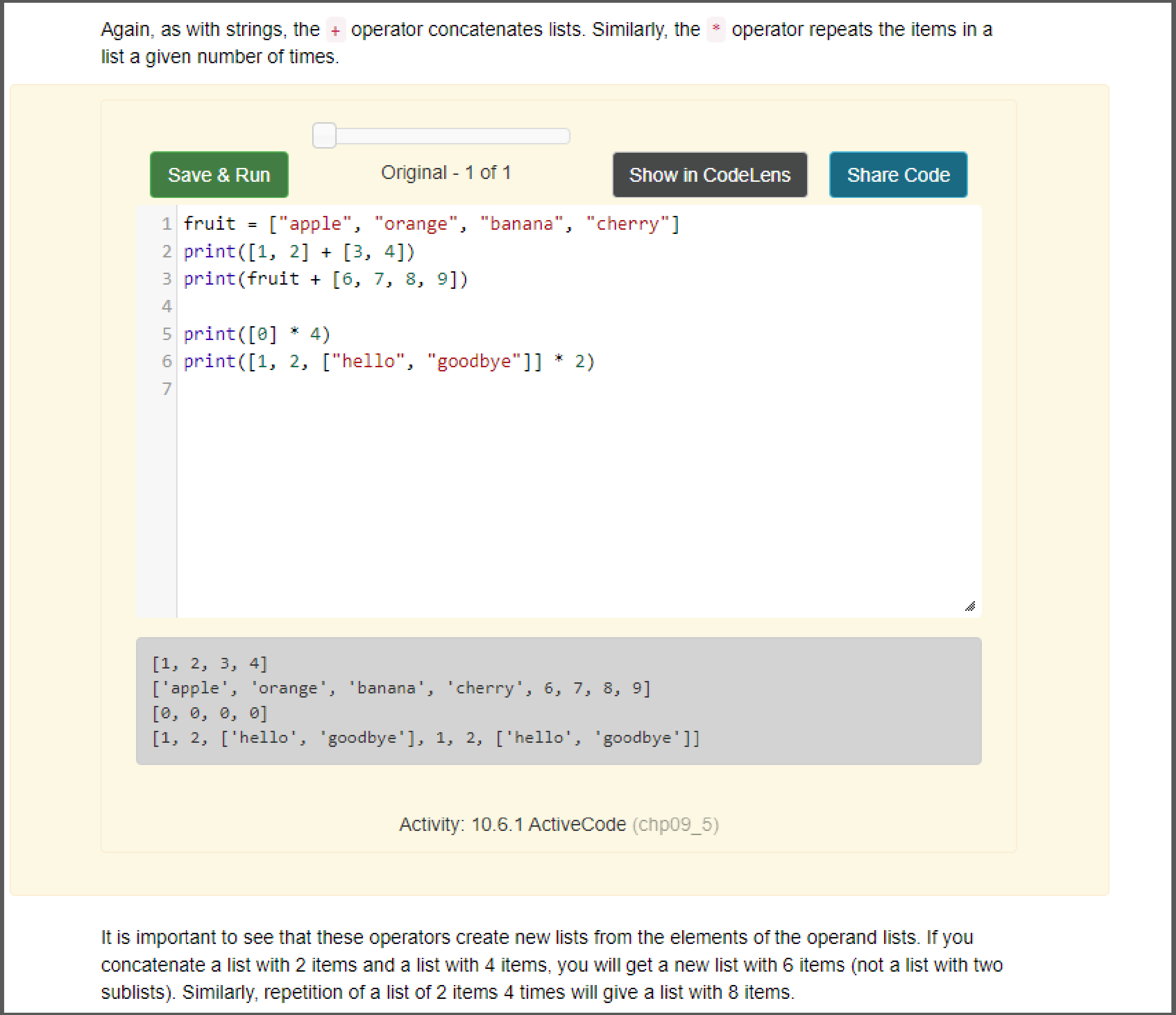
in and not in are boolean operators that test membership in a sequence. We used them previously with strings and they also work here.

```
fruit = ["apple", "orange", "banana", "cherry"]

print("apple" in fruit)
print("pear" in fruit)
```

```
True
False
```

Activity: 10.5.1 ActiveCode (chp09_4)

Lists



Again, as with strings, the `+` operator concatenates lists. Similarly, the `*` operator repeats the items in a list a given number of times.

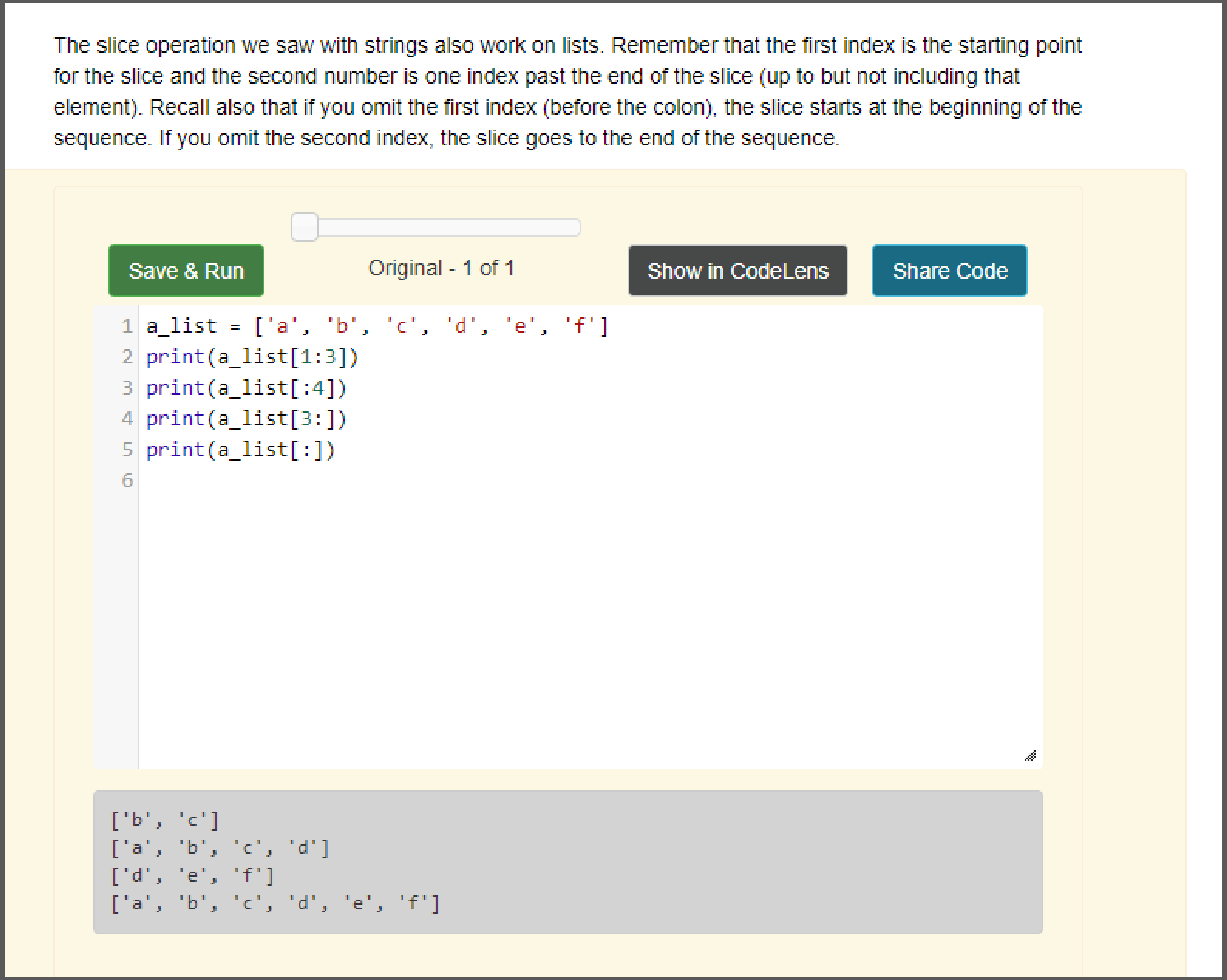
```python
fruit = ["apple", "orange", "banana", "cherry"]
print([1, 2] + [3, 4])
print(fruit + [6, 7, 8, 9])

print([0] * 4)
print([1, 2, ["hello", "goodbye"]] * 2)
```

```
[1, 2, 3, 4]
['apple', 'orange', 'banana', 'cherry', 6, 7, 8, 9]
[0, 0, 0, 0]
[1, 2, ['hello', 'goodbye'], 1, 2, ['hello', 'goodbye']]
```

Activity: 10.6.1 ActiveCode (chp09_5)

It is important to see that these operators create new lists from the elements of the operand lists. If you concatenate a list with 2 items and a list with 4 items, you will get a new list with 6 items (not a list with two sublists). Similarly, repetition of a list of 2 items 4 times will give a list with 8 items.

Lists

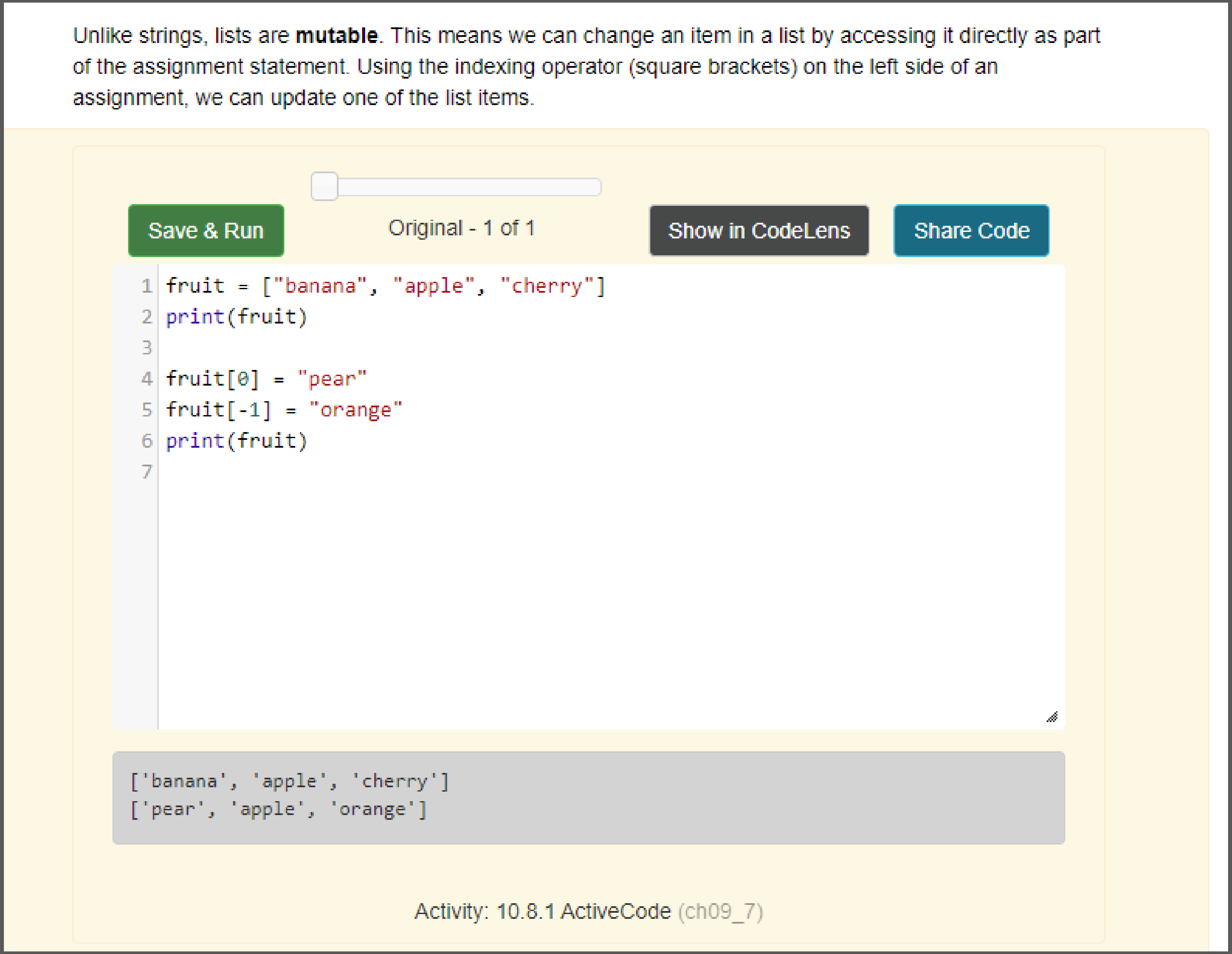


The slice operation we saw with strings also work on lists. Remember that the first index is the starting point for the slice and the second number is one index past the end of the slice (up to but not including that element). Recall also that if you omit the first index (before the colon), the slice starts at the beginning of the sequence. If you omit the second index, the slice goes to the end of the sequence.

```python
a_list = ['a', 'b', 'c', 'd', 'e', 'f']
print(a_list[1:3])
print(a_list[:4])
print(a_list[3:])
print(a_list[:])
```

```
['b', 'c']
['a', 'b', 'c', 'd']
['d', 'e', 'f']
['a', 'b', 'c', 'd', 'e', 'f']
```

Lists

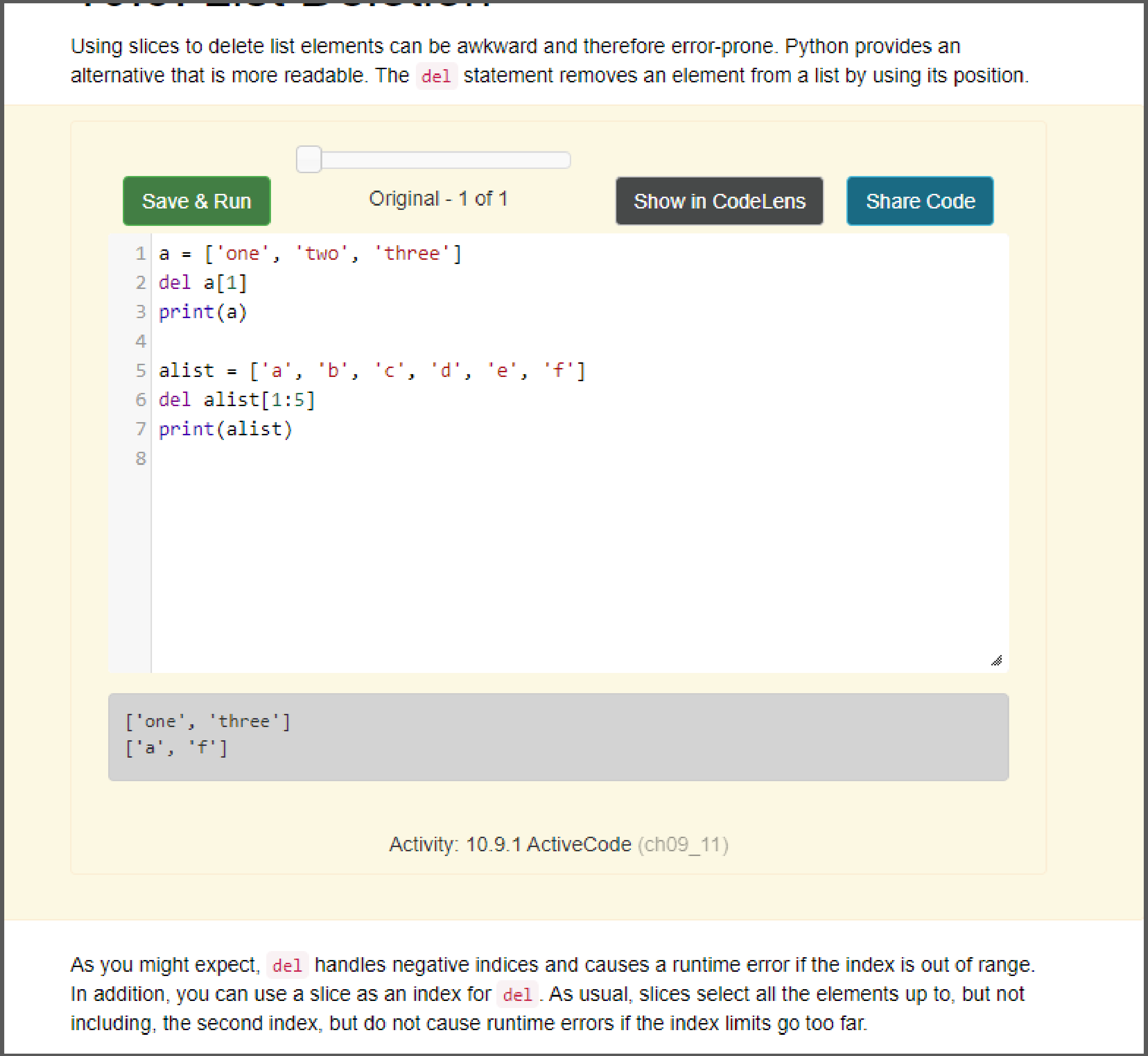

Unlike strings, lists are **mutable**. This means we can change an item in a list by accessing it directly as part of the assignment statement. Using the indexing operator (square brackets) on the left side of an assignment, we can update one of the list items.

```
fruit = ["banana", "apple", "cherry"]
print(fruit)

fruit[0] = "pear"
fruit[-1] = "orange"
print(fruit)

```

```
['banana', 'apple', 'cherry']
['pear', 'apple', 'orange']
```

Activity: 10.8.1 ActiveCode (ch09_7)

Lists



Using slices to delete list elements can be awkward and therefore error-prone. Python provides an alternative that is more readable. The `del` statement removes an element from a list by using its position.

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

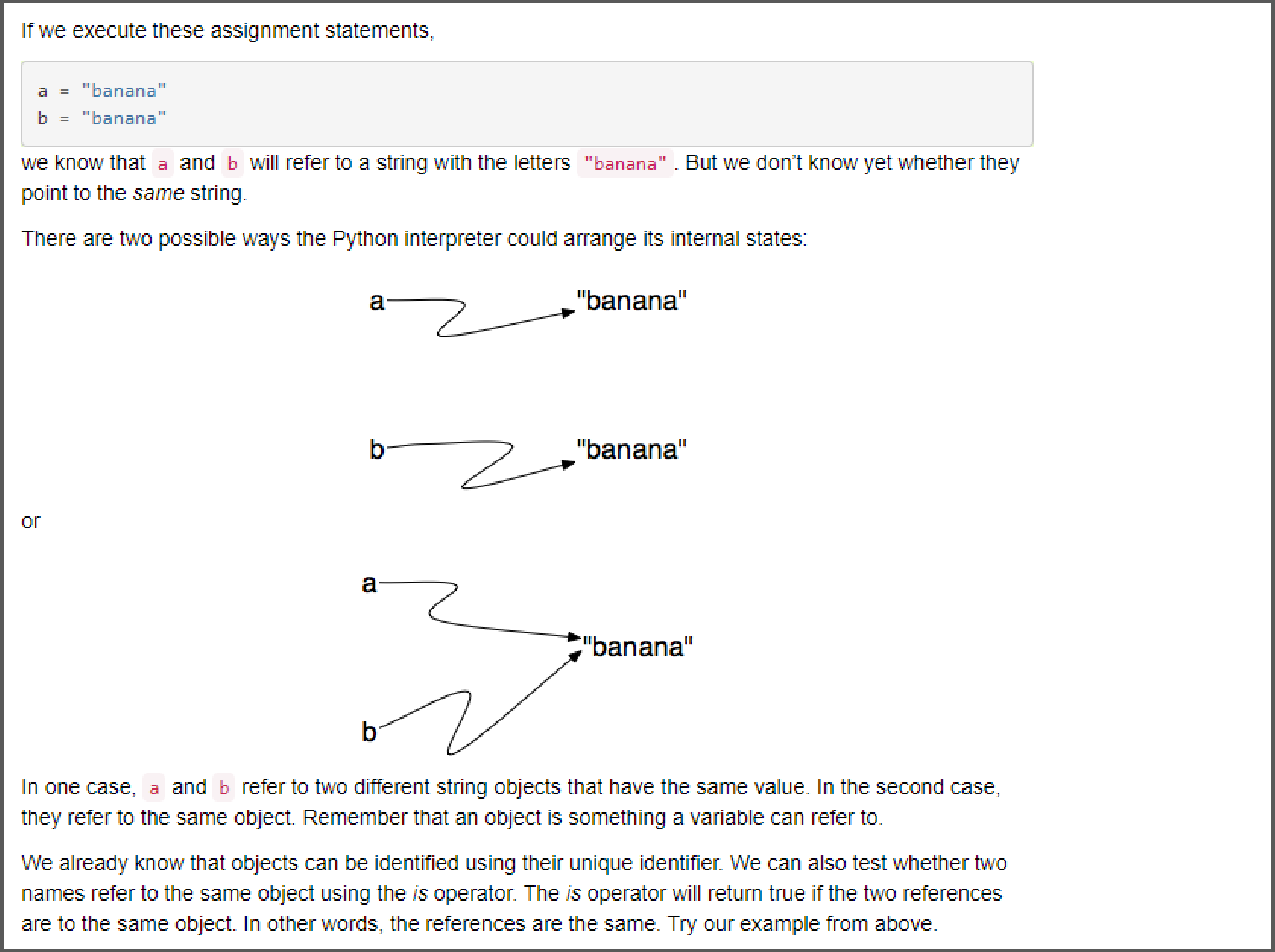
```python
a = ['one', 'two', 'three']
del a[1]
print(a)

alist = ['a', 'b', 'c', 'd', 'e', 'f']
del alist[1:5]
print(alist)
```

```
['one', 'three']
['a', 'f']
```

Activity: 10.9.1 ActiveCode (ch09_11)

As you might expect, `del` handles negative indices and causes a runtime error if the index is out of range. In addition, you can use a slice as an index for `del`. As usual, slices select all the elements up to, but not including, the second index, but do not cause runtime errors if the index limits go too far.

Lists

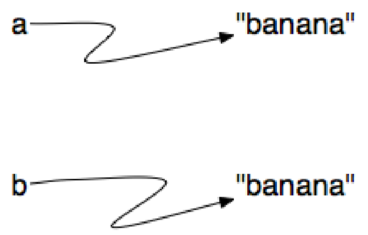


If we execute these assignment statements,

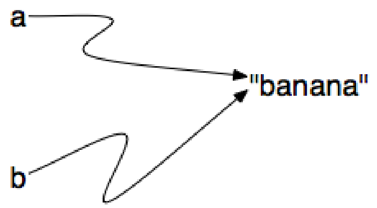
```
a = "banana"
b = "banana"
```

we know that `a` and `b` will refer to a string with the letters `"banana"`. But we don't know yet whether they point to the *same* string.

There are two possible ways the Python interpreter could arrange its internal states:



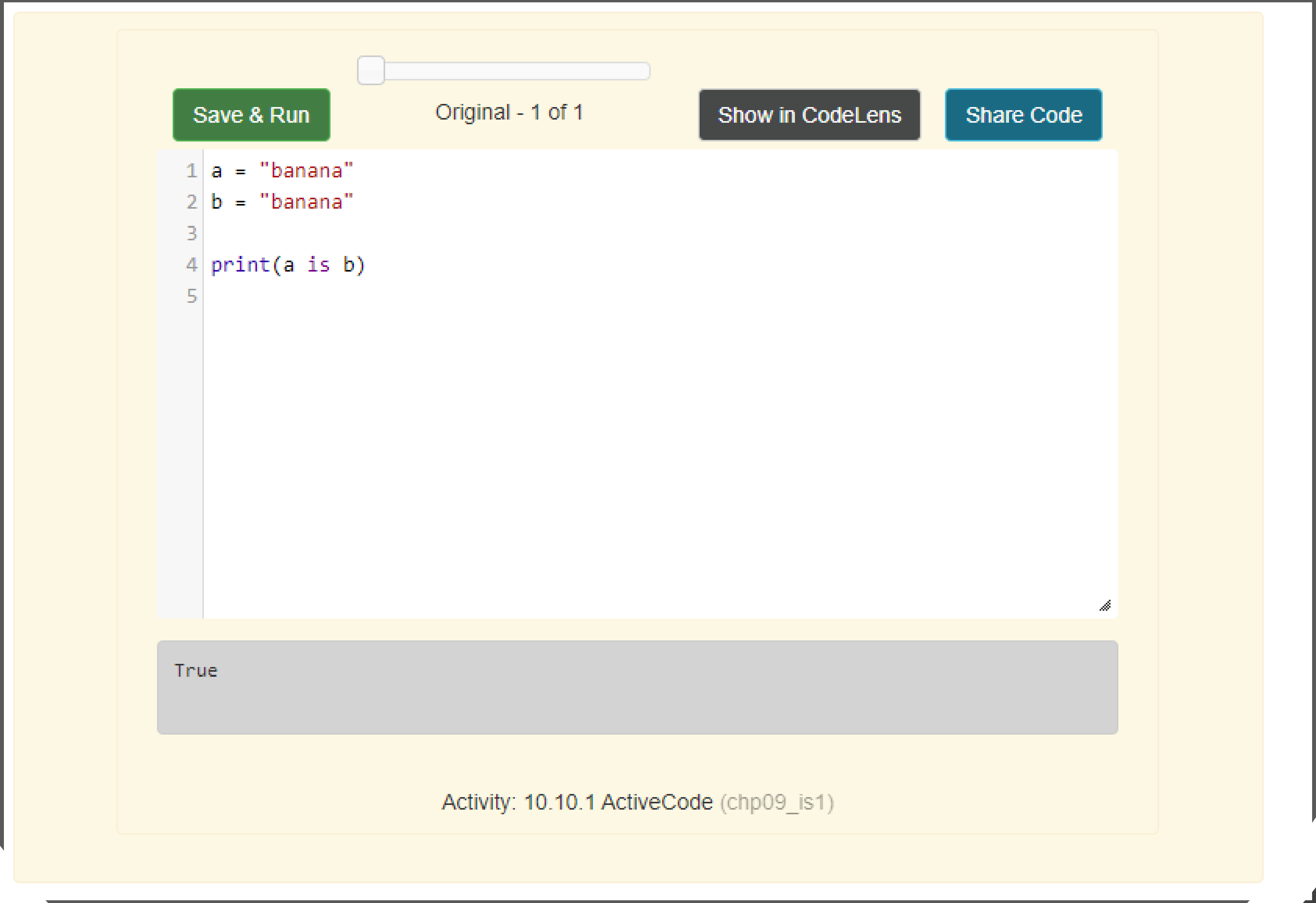
or



In one case, `a` and `b` refer to two different string objects that have the same value. In the second case, they refer to the same object. Remember that an object is something a variable can refer to.

We already know that objects can be identified using their unique identifier. We can also test whether two names refer to the same object using the *is* operator. The *is* operator will return true if the two references are to the same object. In other words, the references are the same. Try our example from above.

Lists

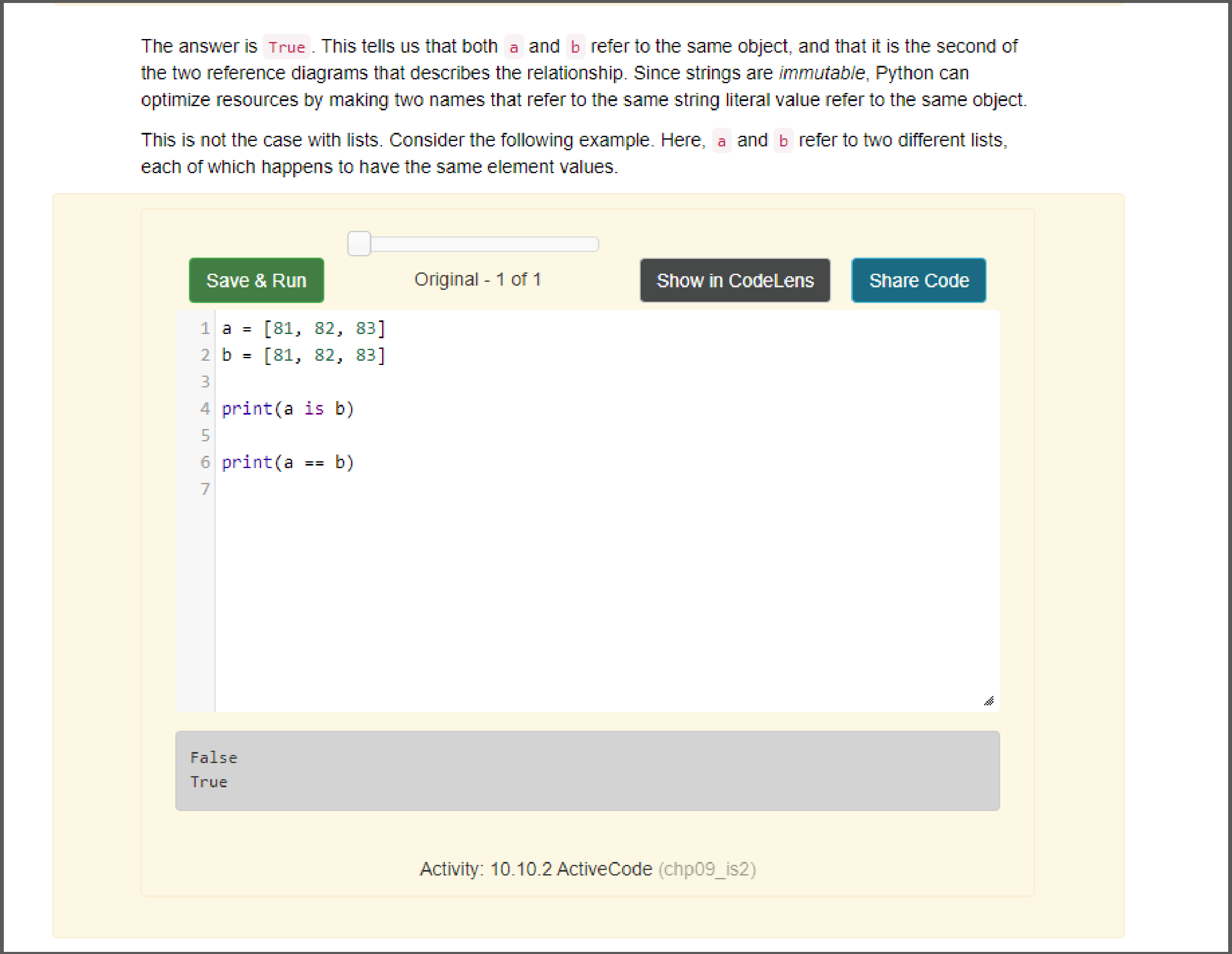


```python
a = "banana"
b = "banana"

print(a is b)

```

```
True
```

Activity: 10.10.1 ActiveCode (chp09_is1)

Lists



The answer is `True`. This tells us that both `a` and `b` refer to the same object, and that it is the second of the two reference diagrams that describes the relationship. Since strings are *immutable*, Python can optimize resources by making two names that refer to the same string literal value refer to the same object.

This is not the case with lists. Consider the following example. Here, `a` and `b` refer to two different lists, each of which happens to have the same element values.

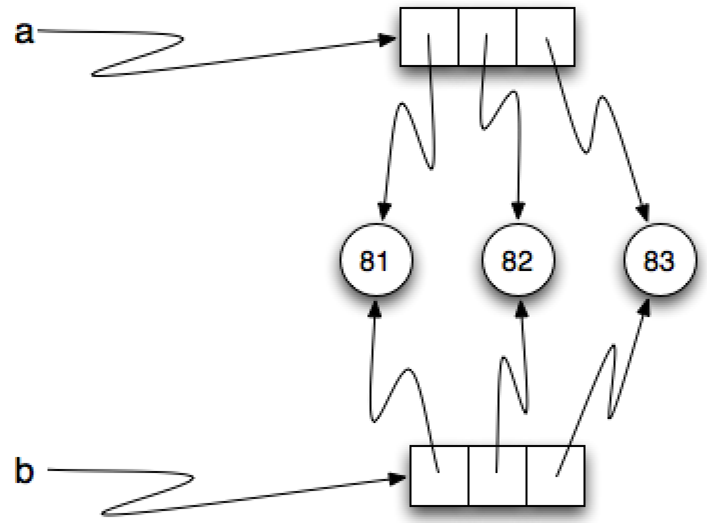
```python
a = [81, 82, 83]
b = [81, 82, 83]

print(a is b)

print(a == b)

```

```
False
True
```

Activity: 10.10.2 ActiveCode (chp09_is2)

Lists

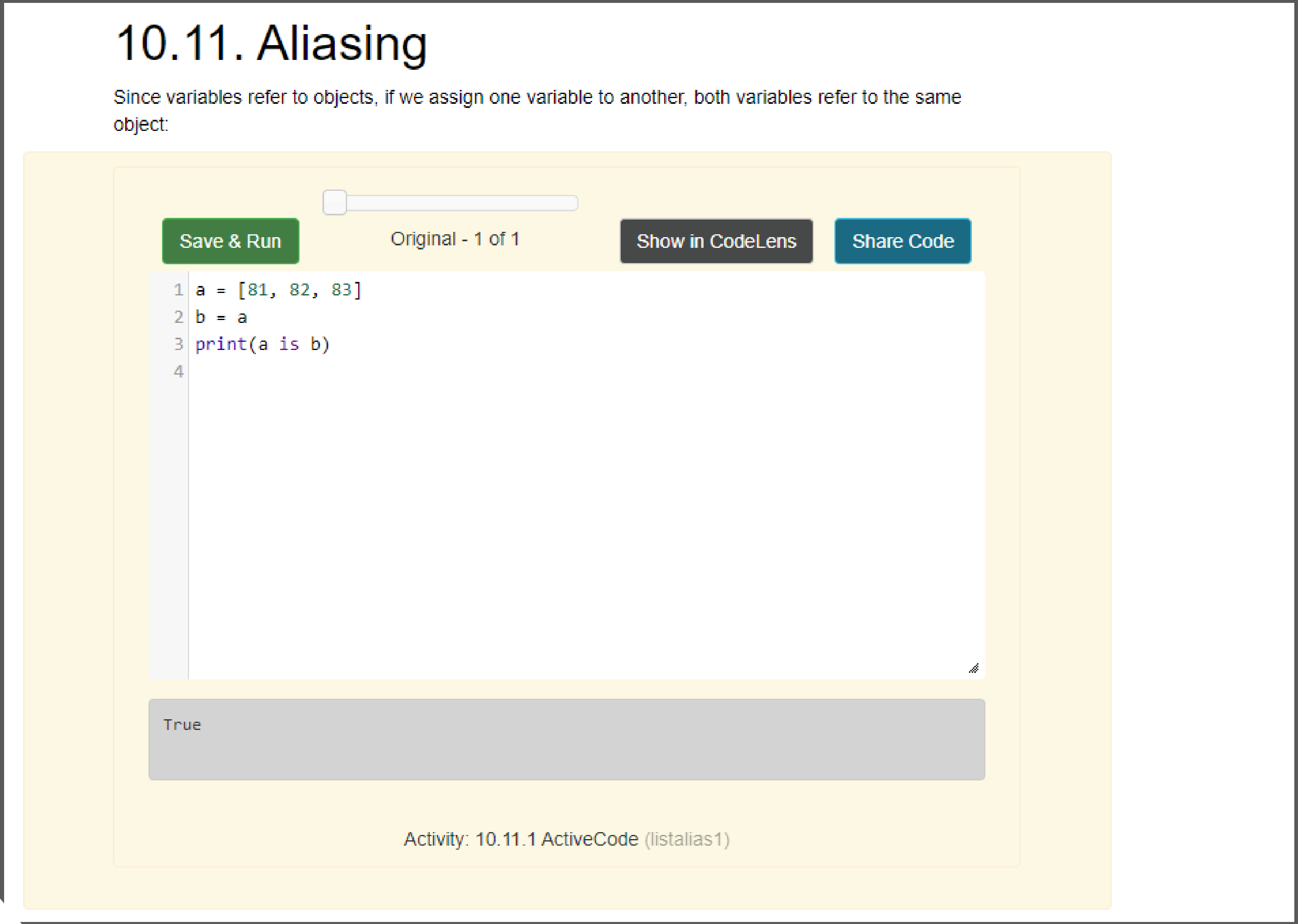
The reference diagram for this example looks like this:



`a` and `b` have the same value but do not refer to the same object.

There is one other important thing to notice about this reference diagram. The variable `a` is a reference to a **collection of references**. Those references actually refer to the integer values in the list. In other words, a list is a collection of references to objects. Interestingly, even though `a` and `b` are two different lists (two different collections of references), the integer object `81` is shared by both. Like strings, integers are also immutable so Python optimizes and lets everyone share the same object for some commonly used small integers.

Lists



10.11. Aliasing

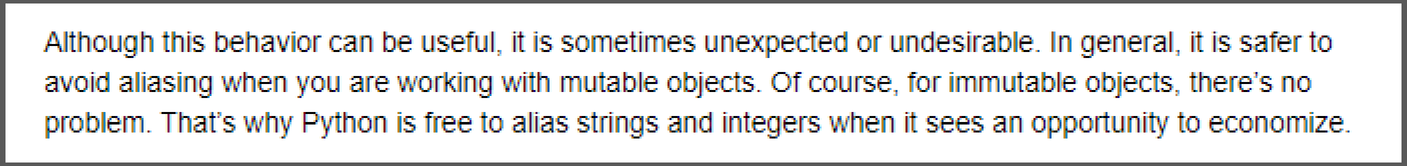
Since variables refer to objects, if we assign one variable to another, both variables refer to the same object:

```
a = [81, 82, 83]
b = a
print(a is b)
```

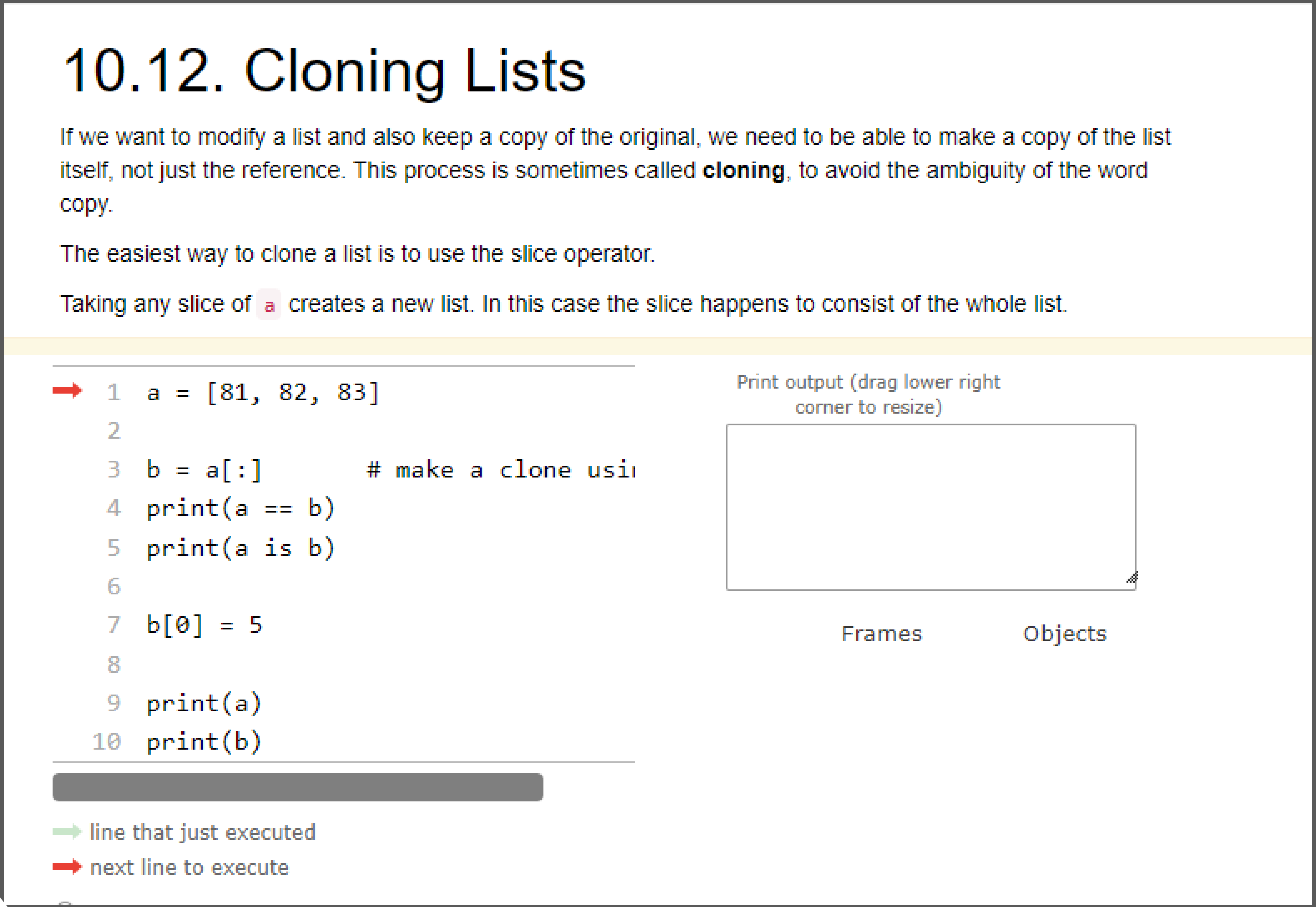
```
True
```

Activity: 10.11.1 ActiveCode (listalias1)

Lists


Although this behavior can be useful, it is sometimes unexpected or undesirable. In general, it is safer to avoid aliasing when you are working with mutable objects. Of course, for immutable objects, there's no problem. That's why Python is free to alias strings and integers when it sees an opportunity to economize.

Lists



10.12. Cloning Lists

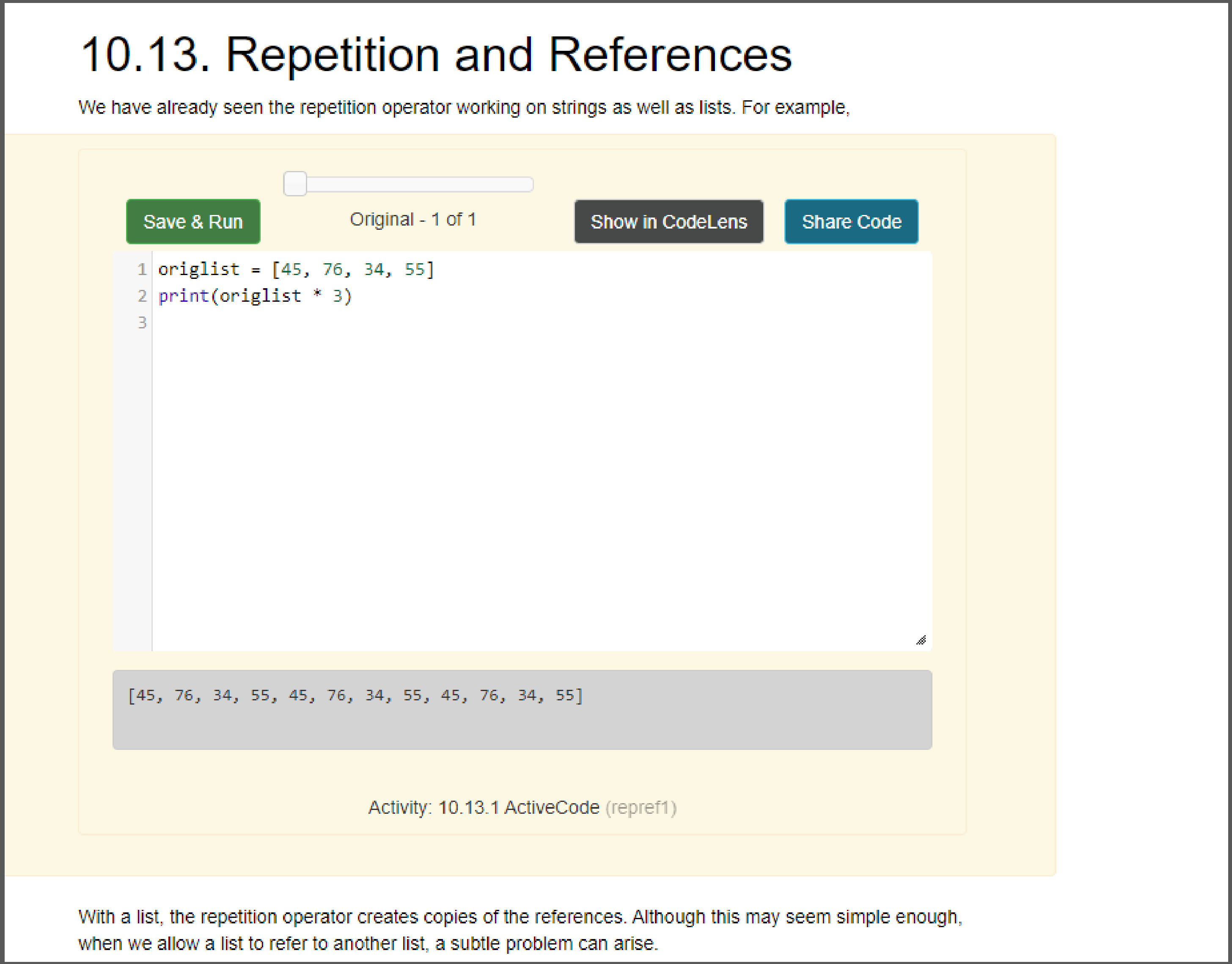
If we want to modify a list and also keep a copy of the original, we need to be able to make a copy of the list itself, not just the reference. This process is sometimes called **cloning**, to avoid the ambiguity of the word copy.

The easiest way to clone a list is to use the slice operator.

Taking any slice of `a` creates a new list. In this case the slice happens to consist of the whole list.

```
a = [81, 82, 83]

b = a[:]       # make a clone usi
print(a == b)
print(a is b)

b[0] = 5

print(a)
print(b)
```

Lists



10.13. Repetition and References

We have already seen the repetition operator working on strings as well as lists. For example,

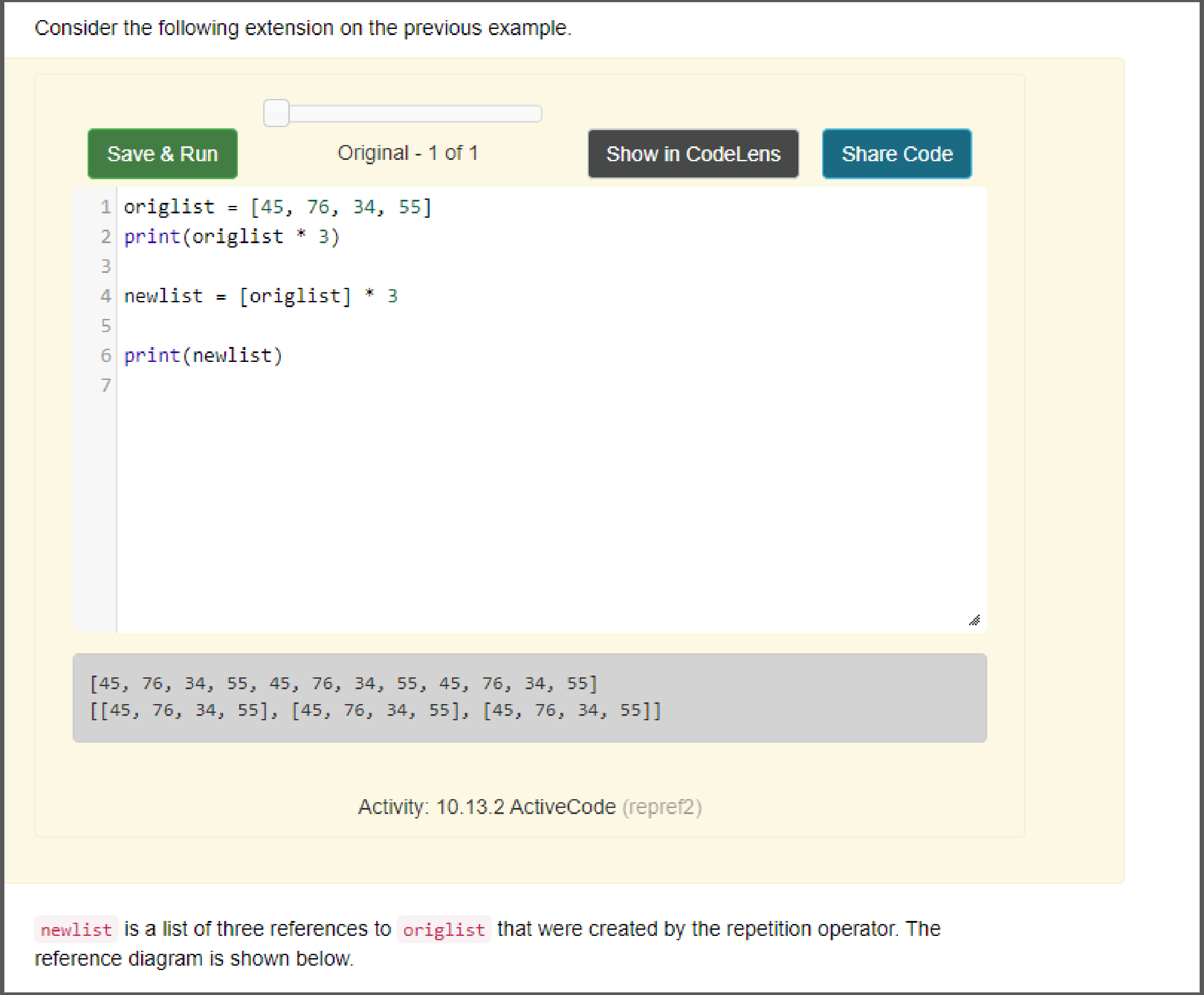
```python
origlist = [45, 76, 34, 55]
print(origlist * 3)
```

```
[45, 76, 34, 55, 45, 76, 34, 55, 45, 76, 34, 55]
```

Activity: 10.13.1 ActiveCode (repref1)

With a list, the repetition operator creates copies of the references. Although this may seem simple enough, when we allow a list to refer to another list, a subtle problem can arise.

Lists



Consider the following extension on the previous example.

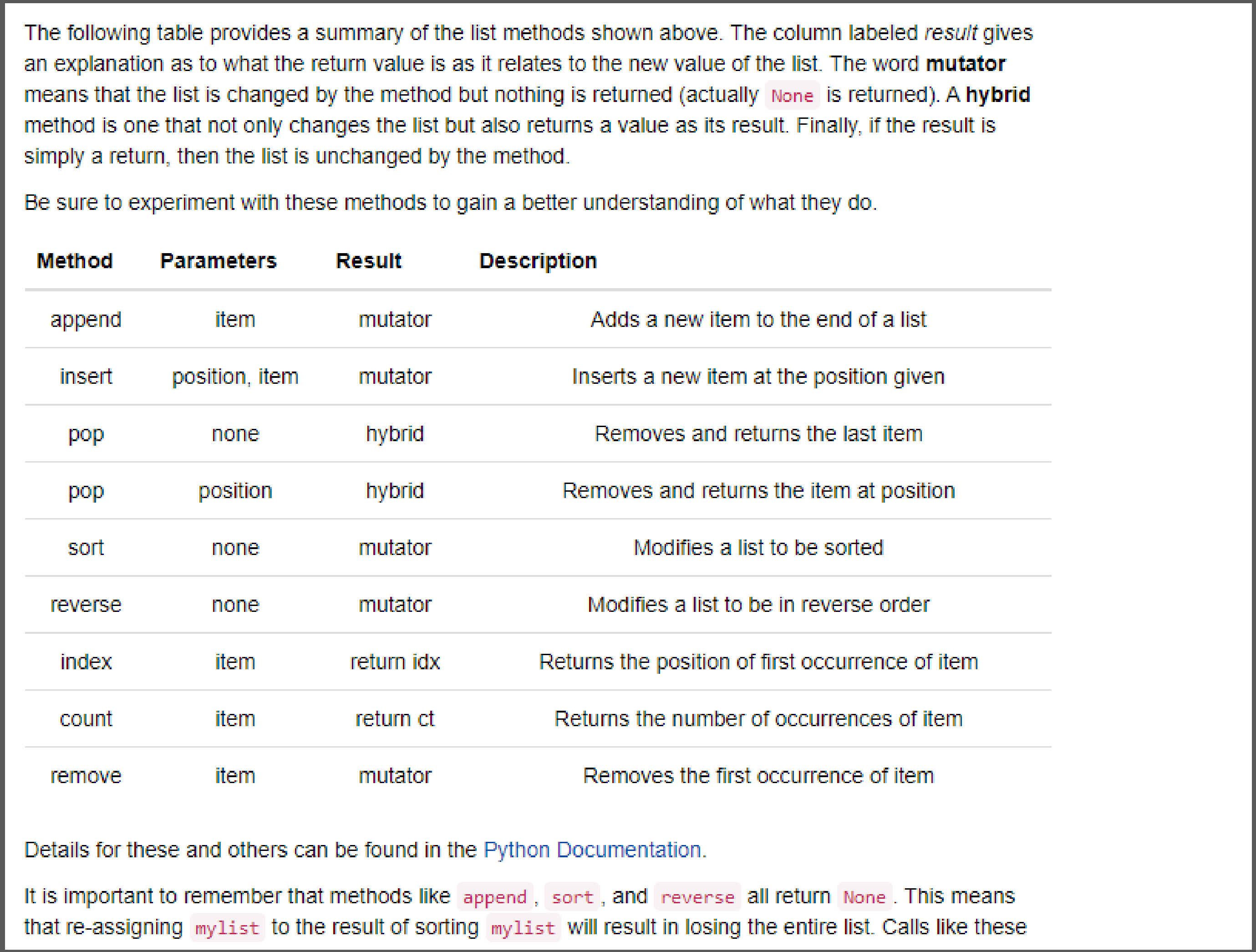
```python
origlist = [45, 76, 34, 55]
print(origlist * 3)

newlist = [origlist] * 3

print(newlist)

```

```
[45, 76, 34, 55, 45, 76, 34, 55, 45, 76, 34, 55]
[[45, 76, 34, 55], [45, 76, 34, 55], [45, 76, 34, 55]]
```

Activity: 10.13.2 ActiveCode (repref2)

`newlist` is a list of three references to `origlist` that were created by the repetition operator. The reference diagram is shown below.

Lists



The following table provides a summary of the list methods shown above. The column labeled *result* gives an explanation as to what the return value is as it relates to the new value of the list. The word **mutator** means that the list is changed by the method but nothing is returned (actually `None` is returned). A **hybrid** method is one that not only changes the list but also returns a value as its result. Finally, if the result is simply a return, then the list is unchanged by the method.

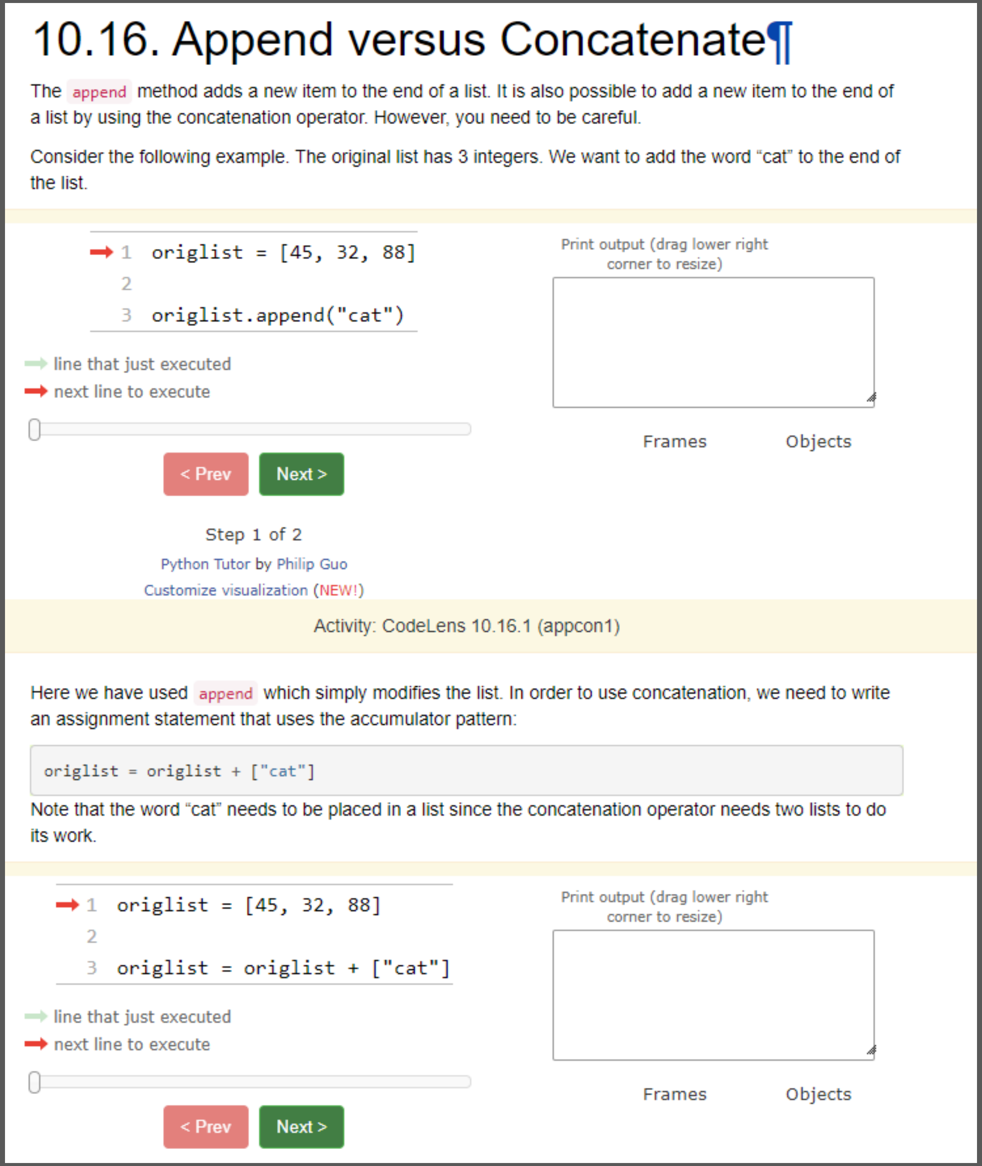
Be sure to experiment with these methods to gain a better understanding of what they do.

Method	Parameters	Result	Description
append	item	mutator	Adds a new item to the end of a list
insert	position, item	mutator	Inserts a new item at the position given
pop	none	hybrid	Removes and returns the last item
pop	position	hybrid	Removes and returns the item at position
sort	none	mutator	Modifies a list to be sorted
reverse	none	mutator	Modifies a list to be in reverse order
index	item	return idx	Returns the position of first occurrence of item
count	item	return ct	Returns the number of occurrences of item
remove	item	mutator	Removes the first occurrence of item

Details for these and others can be found in the Python Documentation.

It is important to remember that methods like `append`, `sort`, and `reverse` all return `None`. This means that re-assigning `mylist` to the result of sorting `mylist` will result in losing the entire list. Calls like these

Lists



10.16. Append versus Concatenate¶

The `append` method adds a new item to the end of a list. It is also possible to add a new item to the end of a list by using the concatenation operator. However, you need to be careful.

Consider the following example. The original list has 3 integers. We want to add the word “cat” to the end of the list.

```
origlist = [45, 32, 88]

origlist.append("cat")
```

line that just executed
next line to execute
< Prev
Next >
Step 1 of 2
Python Tutor by Philip Guo
Customize visualization (NEW!)
Print output (drag lower right corner to resize)
Frames
Objects

Activity: CodeLens 10.16.1 (appcon1)

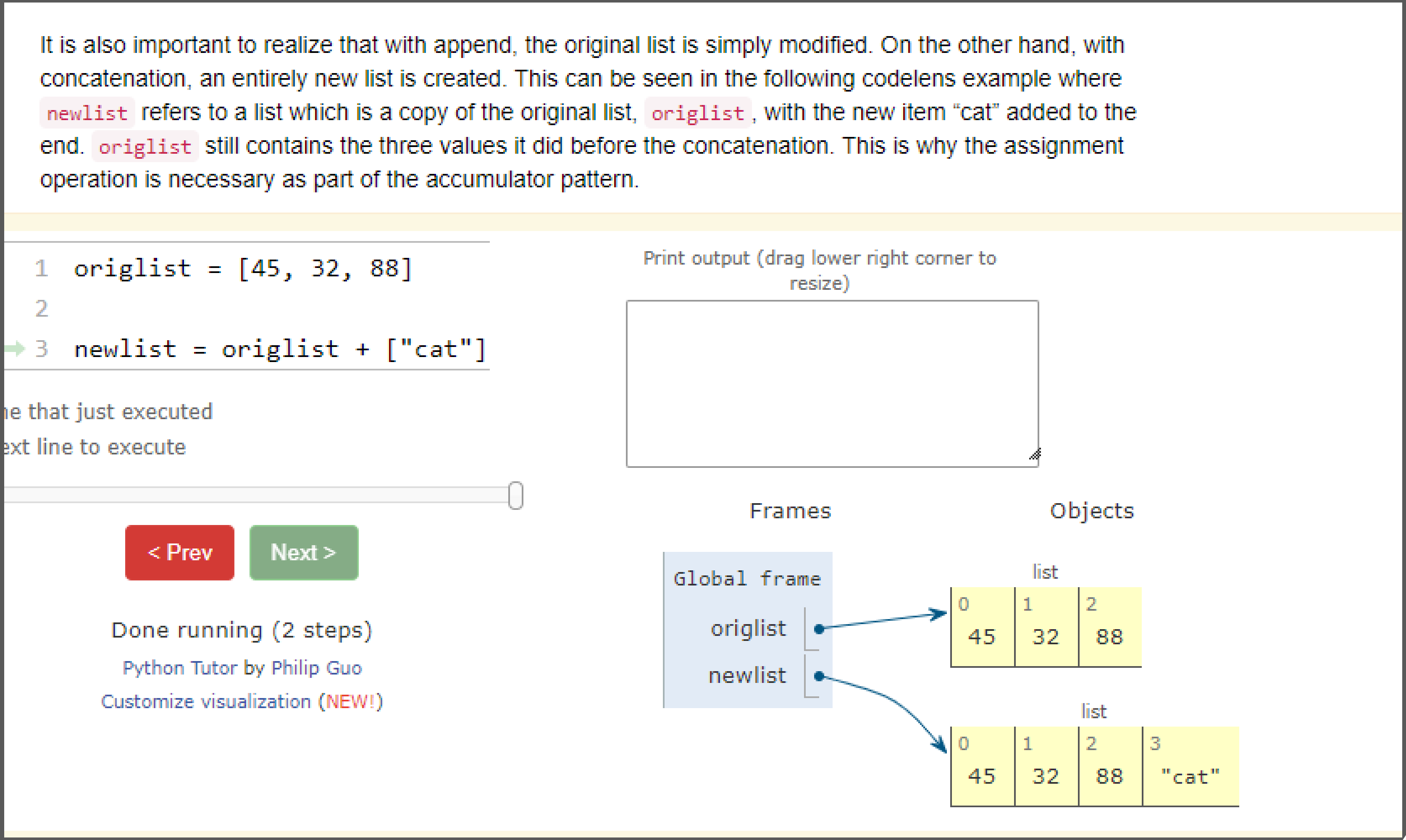
Here we have used `append` which simply modifies the list. In order to use concatenation, we need to write an assignment statement that uses the accumulator pattern:

```
origlist = origlist + ["cat"]
```

Note that the word “cat” needs to be placed in a list since the concatenation operator needs two lists to do its work.

```
origlist = [45, 32, 88]

origlist = origlist + ["cat"]
```

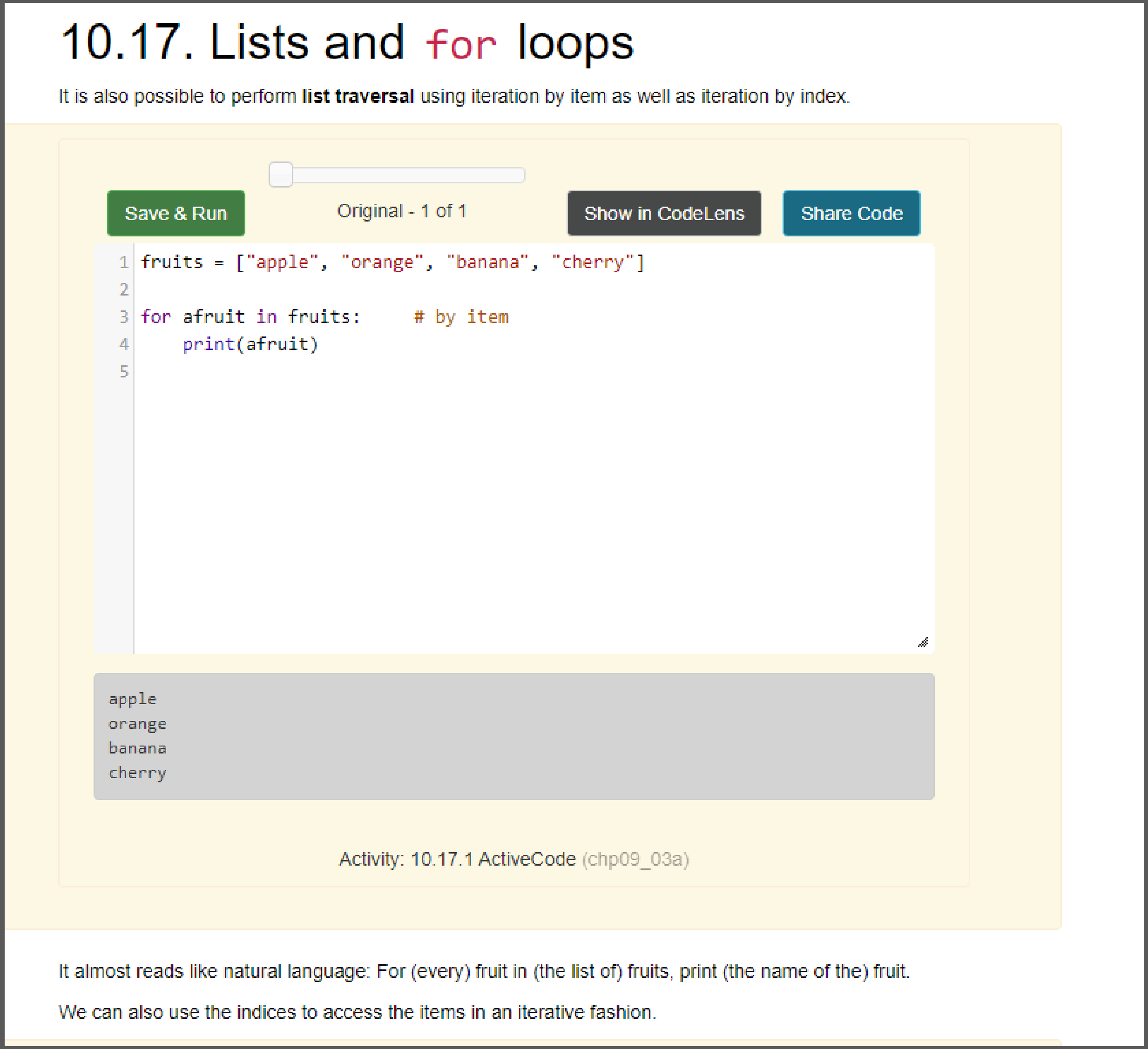
Lists



It is also important to realize that with append, the original list is simply modified. On the other hand, with concatenation, an entirely new list is created. This can be seen in the following codelens example where `newlist` refers to a list which is a copy of the original list, `origlist`, with the new item “cat” added to the end. `origlist` still contains the three values it did before the concatenation. This is why the assignment operation is necessary as part of the accumulator pattern.

```
origlist = [45, 32, 88]

newlist = origlist + ["cat"]
```

Lists



10.17. Lists and `for` loops

It is also possible to perform **list traversal** using iteration by item as well as iteration by index.

```python
fruits = ["apple", "orange", "banana", "cherry"]

for afruit in fruits:     # by item
    print(afruit)

```

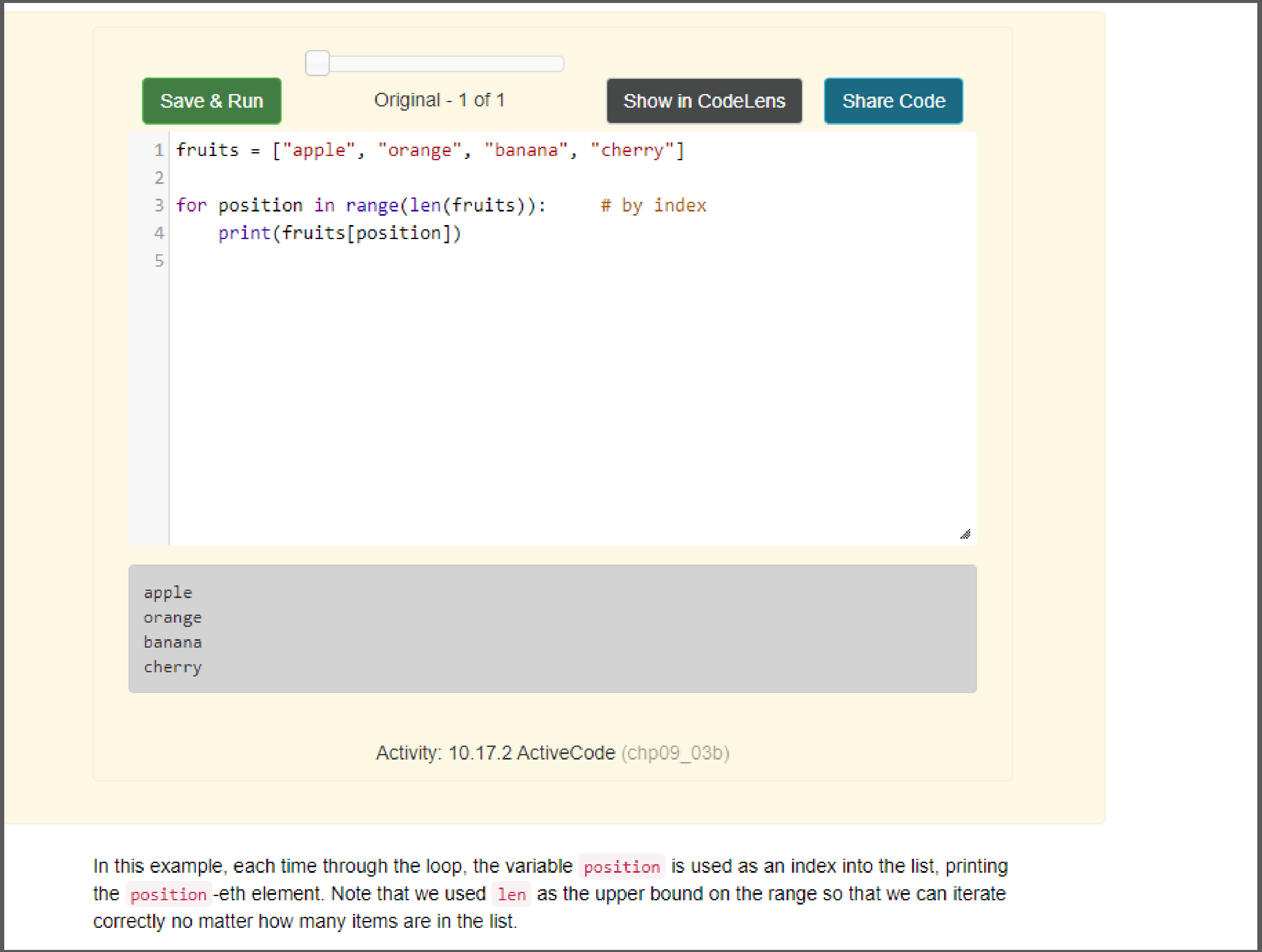
```
apple
orange
banana
cherry
```

Activity: 10.17.1 ActiveCode (chp09_03a)

It almost reads like natural language: For (every) fruit in (the list of) fruits, print (the name of the) fruit.

We can also use the indices to access the items in an iterative fashion.

Lists



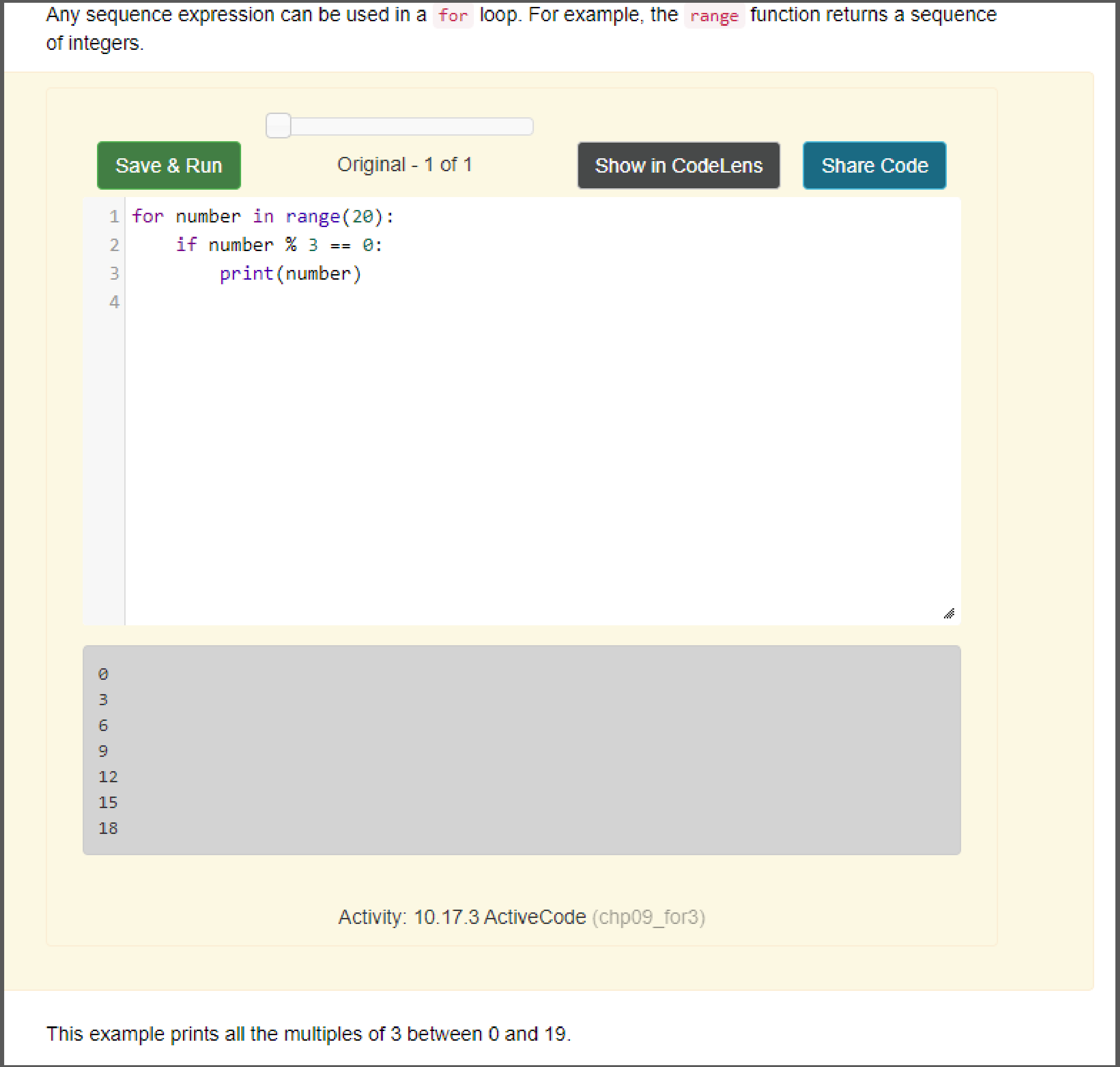
```python
fruits = ["apple", "orange", "banana", "cherry"]

for position in range(len(fruits)):     # by index
    print(fruits[position])

```

```
apple
orange
banana
cherry
```

Activity: 10.17.2 ActiveCode (chp09_03b)

In this example, each time through the loop, the variable `position` is used as an index into the list, printing the `position`-eth element. Note that we used `len` as the upper bound on the range so that we can iterate correctly no matter how many items are in the list.

Lists



Any sequence expression can be used in a `for` loop. For example, the `range` function returns a sequence of integers.

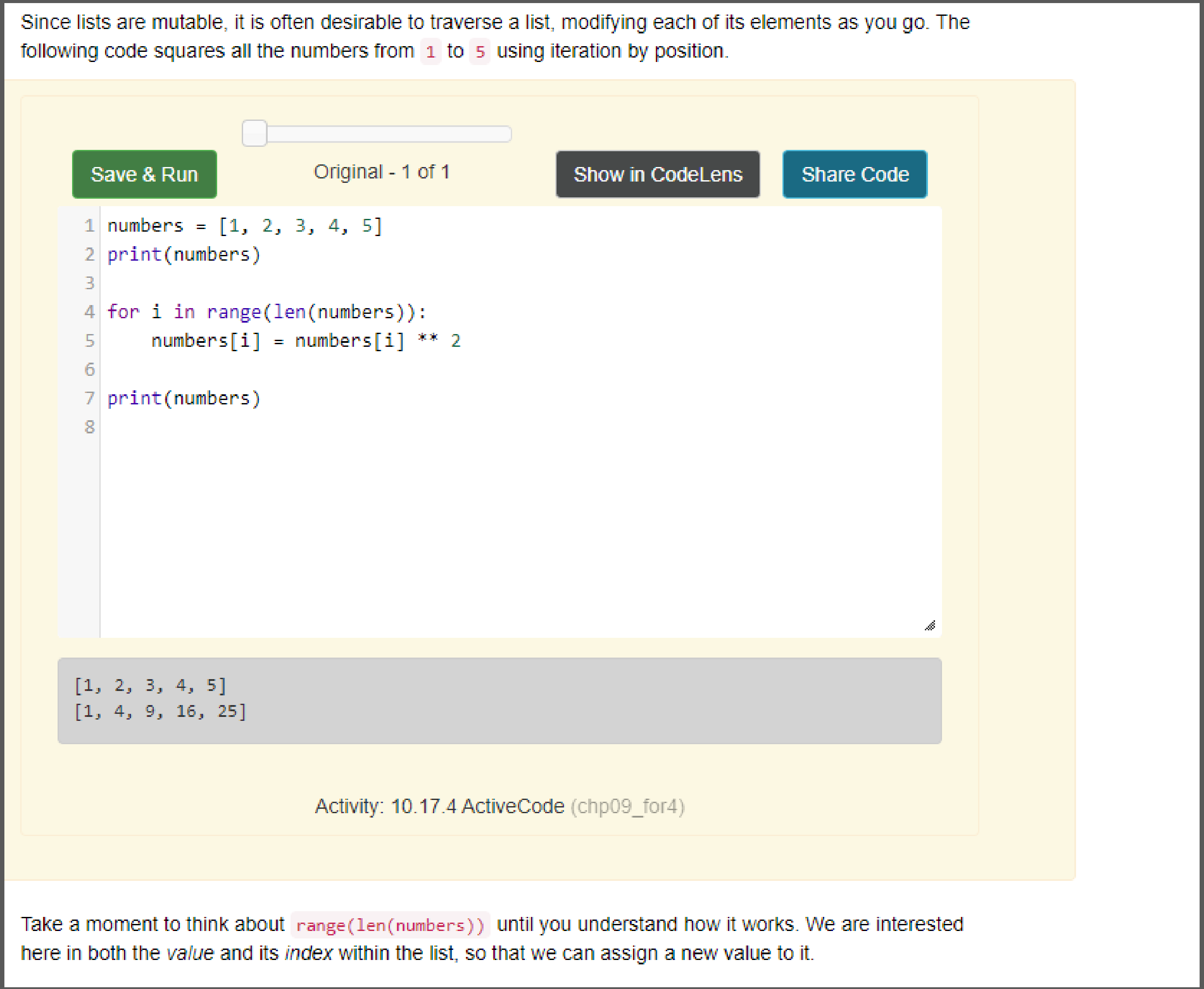
```python
for number in range(20):
    if number % 3 == 0:
        print(number)
```

```
0
3
6
9
12
15
18
```

Activity: 10.17.3 ActiveCode (chp09_for3)

This example prints all the multiples of 3 between 0 and 19.

Lists



Since lists are mutable, it is often desirable to traverse a list, modifying each of its elements as you go. The following code squares all the numbers from `1` to `5` using iteration by position.

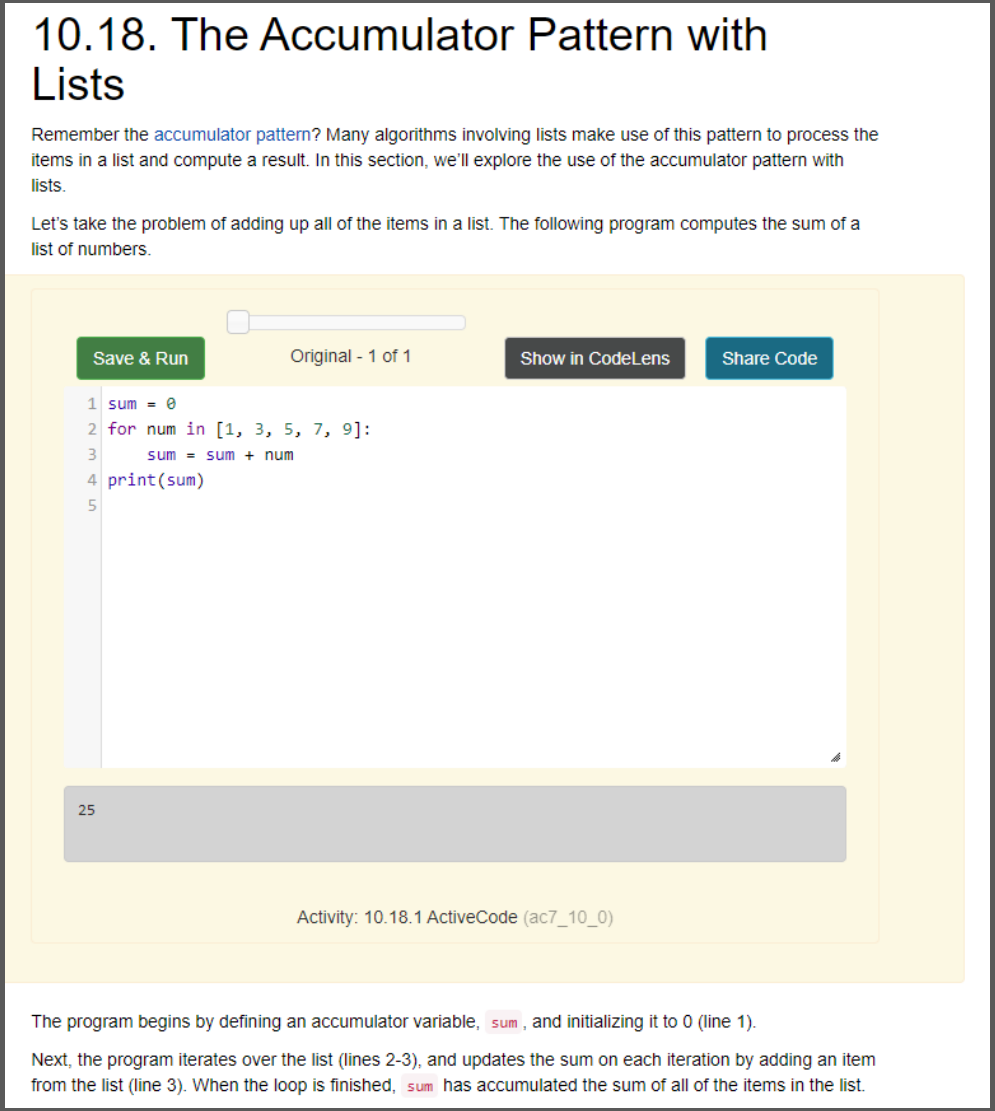
```python
numbers = [1, 2, 3, 4, 5]
print(numbers)

for i in range(len(numbers)):
    numbers[i] = numbers[i] ** 2

print(numbers)
```

```
[1, 2, 3, 4, 5]
[1, 4, 9, 16, 25]
```

Activity: 10.17.4 ActiveCode (chp09_for4)

Take a moment to think about `range(len(numbers))` until you understand how it works. We are interested here in both the *value* and its *index* within the list, so that we can assign a new value to it.

Lists



10.18. The Accumulator Pattern with Lists

Remember the accumulator pattern? Many algorithms involving lists make use of this pattern to process the items in a list and compute a result. In this section, we'll explore the use of the accumulator pattern with lists.

Let's take the problem of adding up all of the items in a list. The following program computes the sum of a list of numbers.

```
1 sum = 0
2 for num in [1, 3, 5, 7, 9]:
3     sum = sum + num
4 print(sum)
5
```

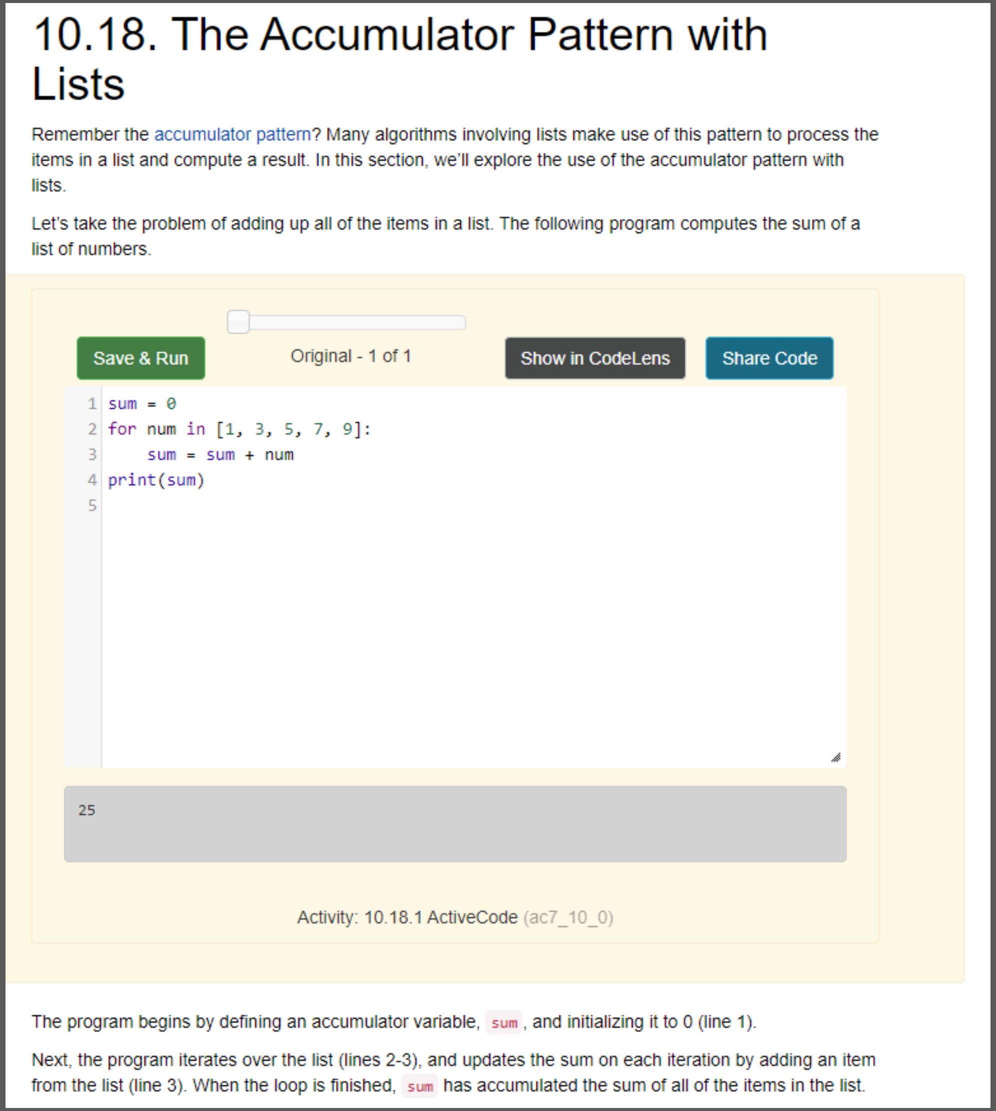
```
25
```

Activity: 10.18.1 ActiveCode (ac7_10_0)

The program begins by defining an accumulator variable, `sum`, and initializing it to 0 (line 1).

Next, the program iterates over the list (lines 2-3), and updates the sum on each iteration by adding an item from the list (line 3). When the loop is finished, `sum` has accumulated the sum of all of the items in the list.

Lists



10.18. The Accumulator Pattern with Lists

Remember the accumulator pattern? Many algorithms involving lists make use of this pattern to process the items in a list and compute a result. In this section, we'll explore the use of the accumulator pattern with lists.

Let's take the problem of adding up all of the items in a list. The following program computes the sum of a list of numbers.

```
1 sum = 0
2 for num in [1, 3, 5, 7, 9]:
3     sum = sum + num
4 print(sum)
5
```

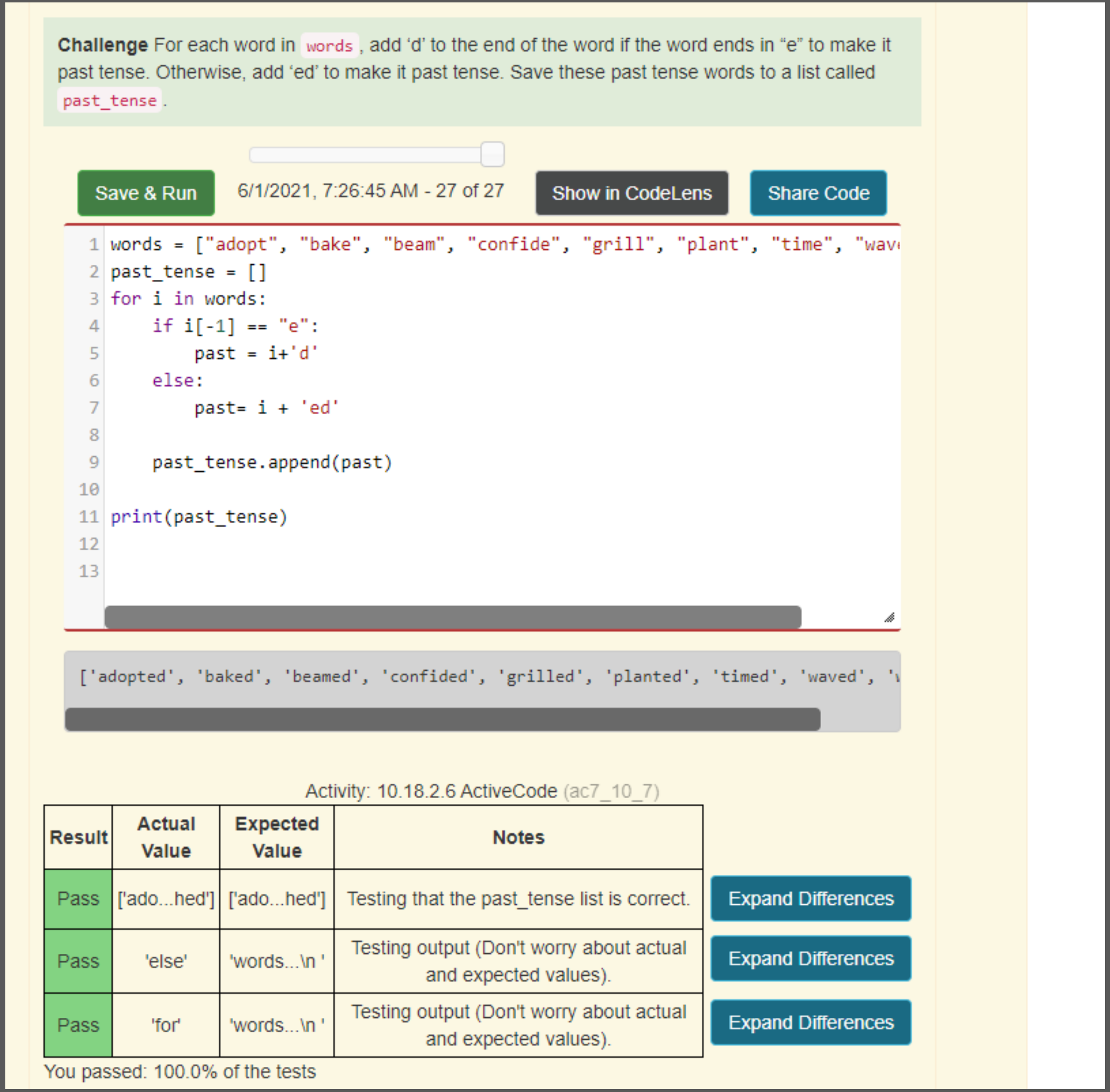
```
25
```

Activity: 10.18.1 ActiveCode (ac7_10_0)

The program begins by defining an accumulator variable, `sum`, and initializing it to 0 (line 1).

Next, the program iterates over the list (lines 2-3), and updates the sum on each iteration by adding an item from the list (line 3). When the loop is finished, `sum` has accumulated the sum of all of the items in the list.

Lists



Challenge For each word in `words`, add 'd' to the end of the word if the word ends in "e" to make it past tense. Otherwise, add 'ed' to make it past tense. Save these past tense words to a list called `past_tense`.

```
words = ["adopt", "bake", "beam", "confide", "grill", "plant", "time", "wav
past_tense = []
for i in words:
    if i[-1] == "e":
        past = i+'d'
    else:
        past= i + 'ed'

    past_tense.append(past)

print(past_tense)


```

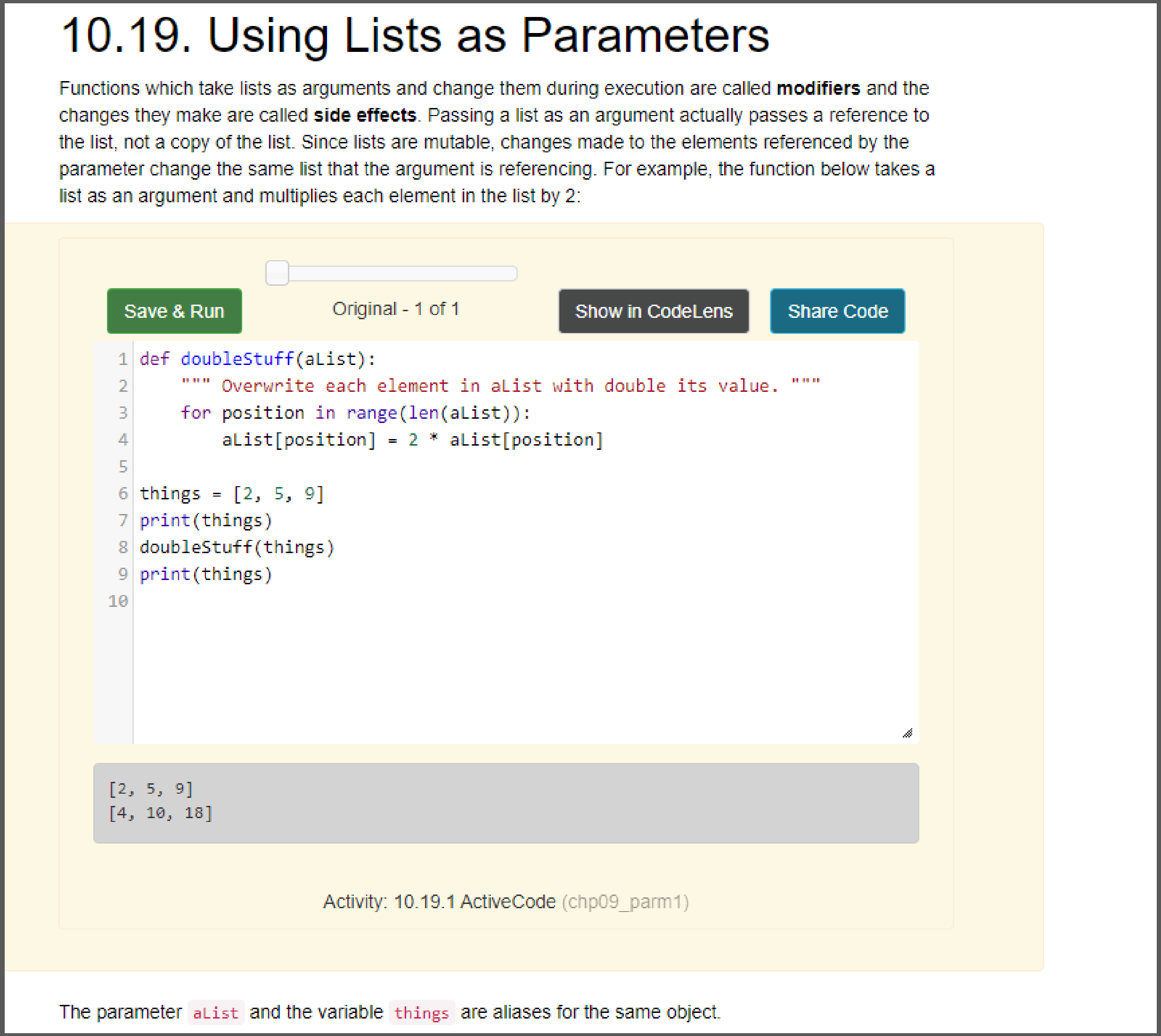
```
['adopted', 'baked', 'beamed', 'confided', 'grilled', 'planted', 'timed', 'waved', '
```

Activity: 10.18.2.6 ActiveCode (ac7_10_7)

Result	Actual Value	Expected Value	Notes
Pass	['ado...hed']	['ado...hed']	Testing that the past_tense list is correct.
Pass	'else'	'words...\n '	Testing output (Don't worry about actual and expected values).
Pass	'for'	'words...\n '	Testing output (Don't worry about actual and expected values).

You passed: 100.0% of the tests

Lists



10.19. Using Lists as Parameters

Functions which take lists as arguments and change them during execution are called **modifiers** and the changes they make are called **side effects**. Passing a list as an argument actually passes a reference to the list, not a copy of the list. Since lists are mutable, changes made to the elements referenced by the parameter change the same list that the argument is referencing. For example, the function below takes a list as an argument and multiplies each element in the list by 2:

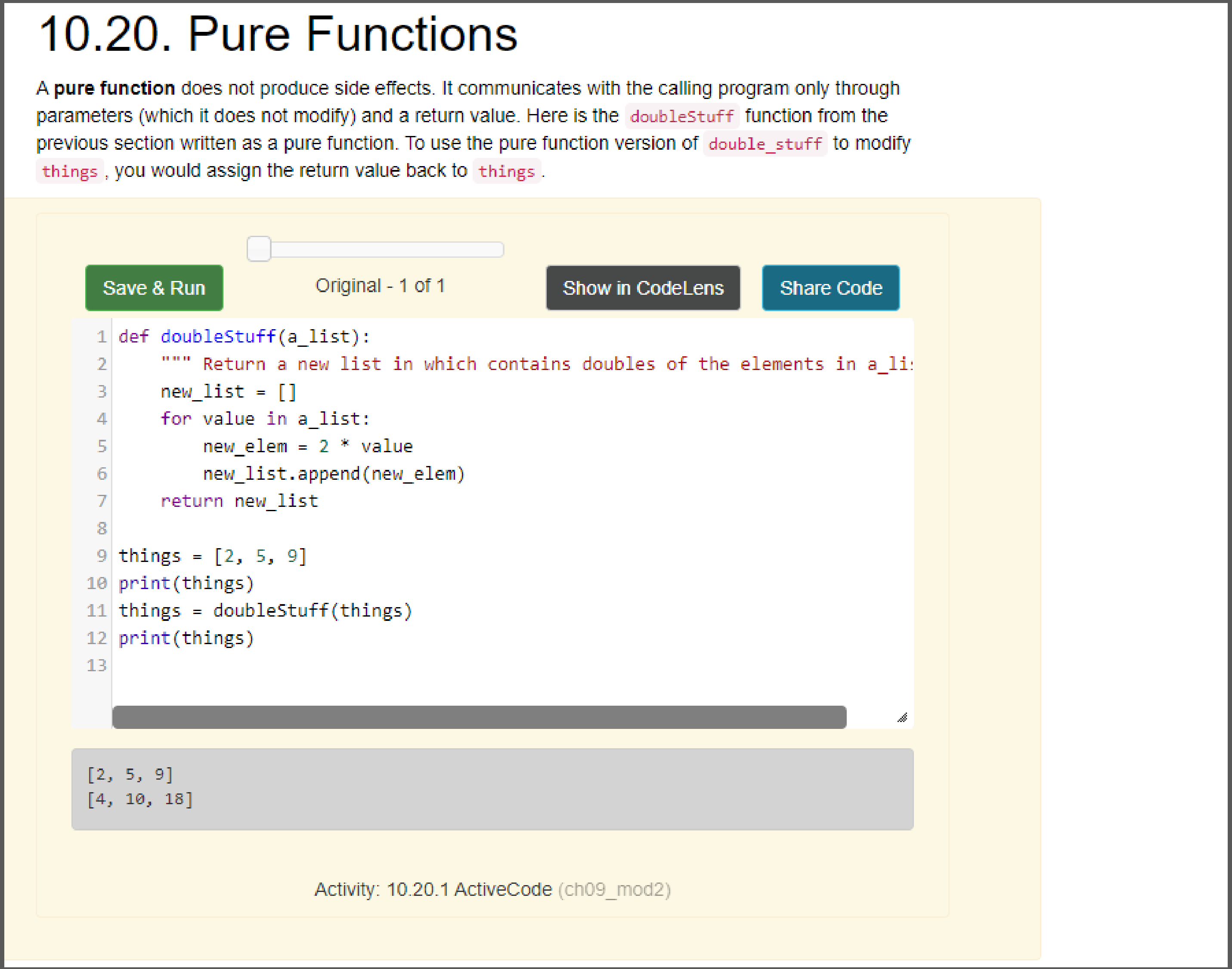
```python
def doubleStuff(aList):
    """ Overwrite each element in aList with double its value. """
    for position in range(len(aList)):
        aList[position] = 2 * aList[position]

things = [2, 5, 9]
print(things)
doubleStuff(things)
print(things)

```

```
[2, 5, 9]
[4, 10, 18]
```

Activity: 10.19.1 ActiveCode (chp09_parm1)

The parameter `aList` and the variable `things` are aliases for the same object.

Lists



10.20. Pure Functions

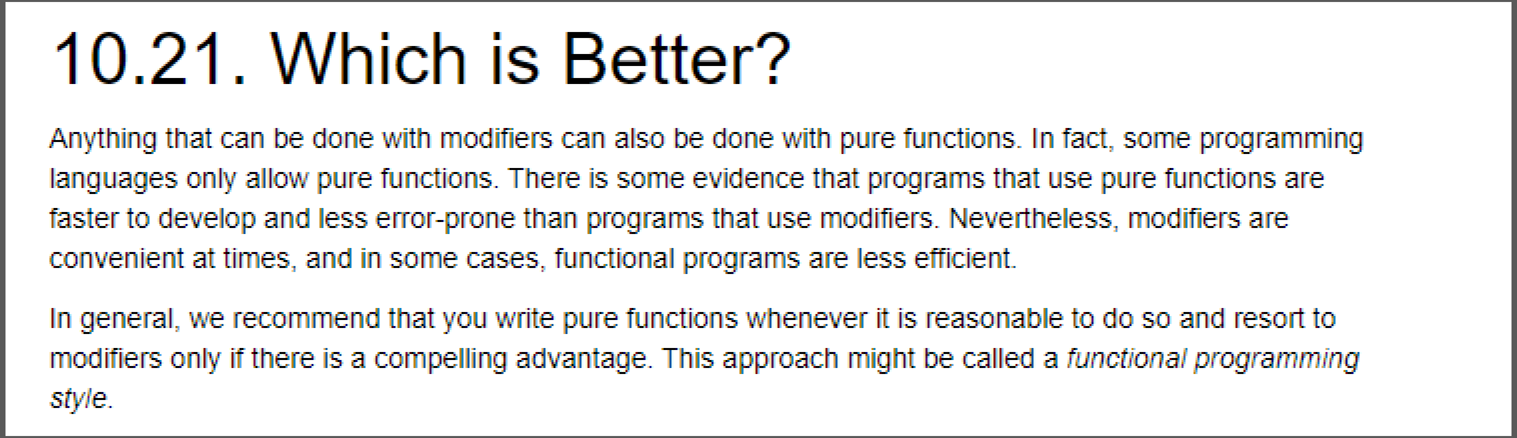
A **pure function** does not produce side effects. It communicates with the calling program only through parameters (which it does not modify) and a return value. Here is the `doubleStuff` function from the previous section written as a pure function. To use the pure function version of `double_stuff` to modify `things`, you would assign the return value back to `things`.

```
def doubleStuff(a_list):
    """ Return a new list in which contains doubles of the elements in a_li
    new_list = []
    for value in a_list:
        new_elem = 2 * value
        new_list.append(new_elem)
    return new_list

things = [2, 5, 9]
print(things)
things = doubleStuff(things)
print(things)
```

```
[2, 5, 9]
[4, 10, 18]
```

Activity: 10.20.1 ActiveCode (ch09_mod2)

Lists

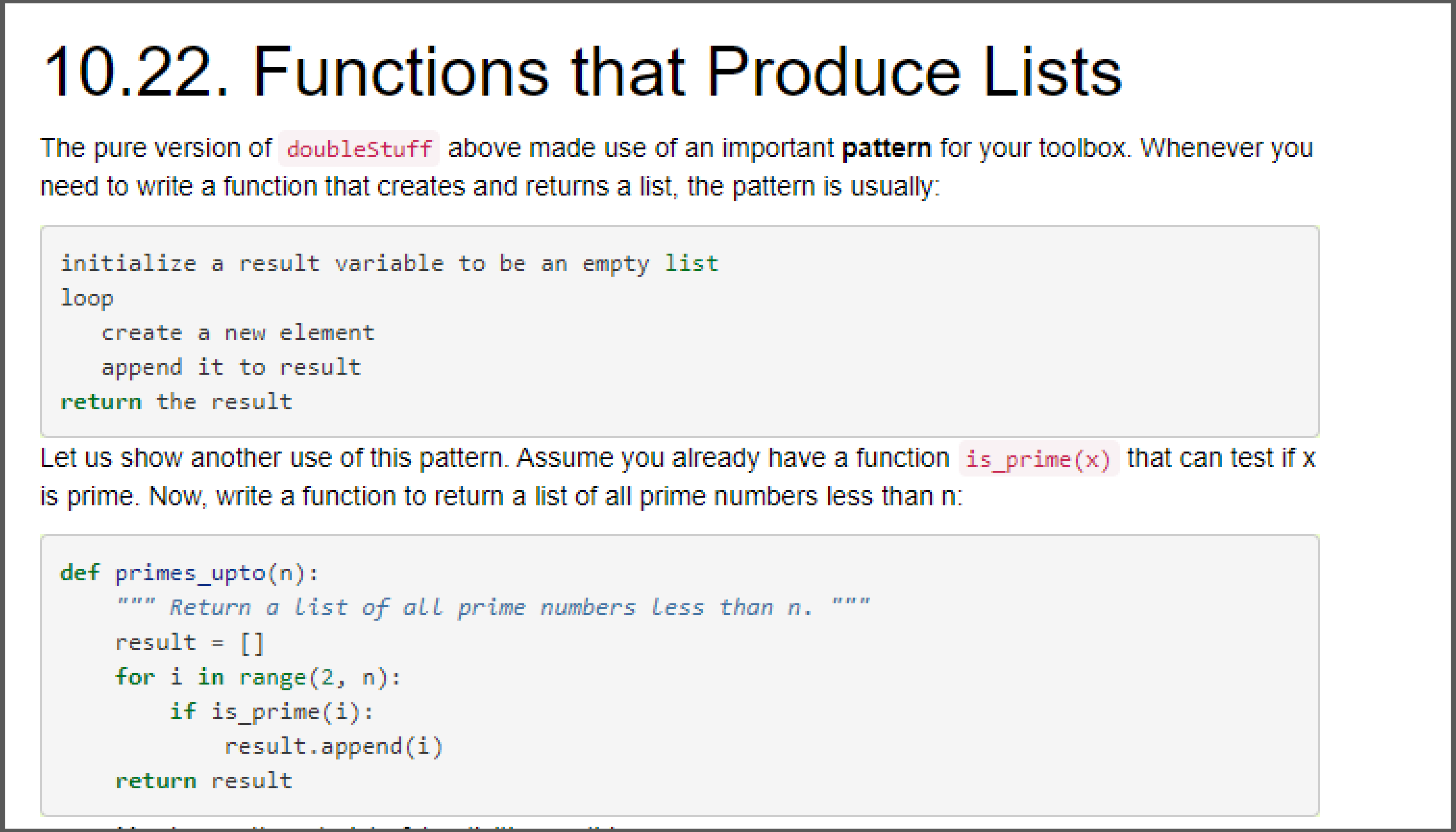


10.21. Which is Better?

Anything that can be done with modifiers can also be done with pure functions. In fact, some programming languages only allow pure functions. There is some evidence that programs that use pure functions are faster to develop and less error-prone than programs that use modifiers. Nevertheless, modifiers are convenient at times, and in some cases, functional programs are less efficient.

In general, we recommend that you write pure functions whenever it is reasonable to do so and resort to modifiers only if there is a compelling advantage. This approach might be called a *functional programming style*.

Lists



10.22. Functions that Produce Lists

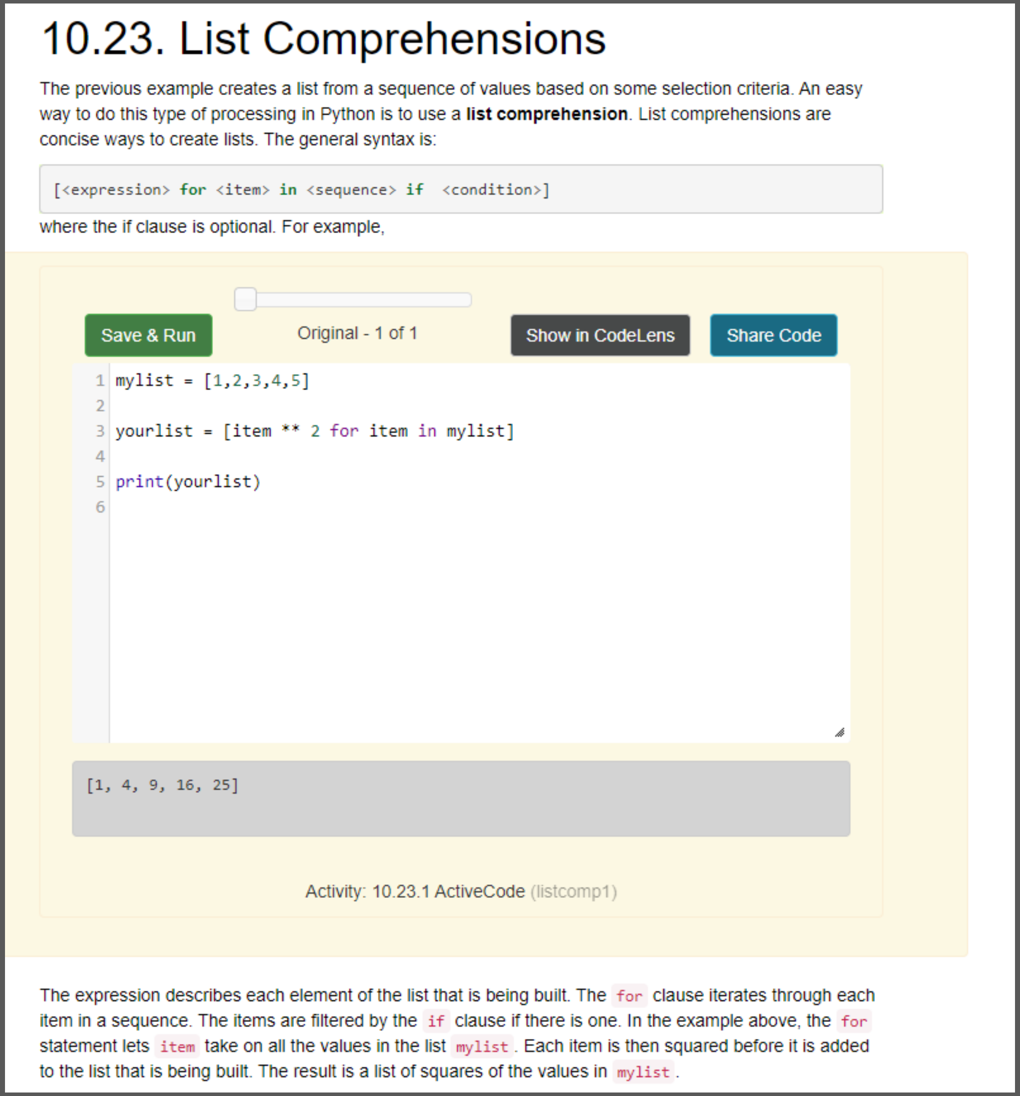
The pure version of `doubleStuff` above made use of an important **pattern** for your toolbox. Whenever you need to write a function that creates and returns a list, the pattern is usually:

```
initialize a result variable to be an empty list
loop
   create a new element
   append it to result
return the result
```

Let us show another use of this pattern. Assume you already have a function `is_prime(x)` that can test if x is prime. Now, write a function to return a list of all prime numbers less than n:

```python
def primes_upto(n):
    """ Return a list of all prime numbers less than n. """
    result = []
    for i in range(2, n):
        if is_prime(i):
            result.append(i)
    return result
```

Lists



10.23. List Comprehensions

The previous example creates a list from a sequence of values based on some selection criteria. An easy way to do this type of processing in Python is to use a **list comprehension**. List comprehensions are concise ways to create lists. The general syntax is:

```
[<expression> for <item> in <sequence> if  <condition>]
```

where the if clause is optional. For example,

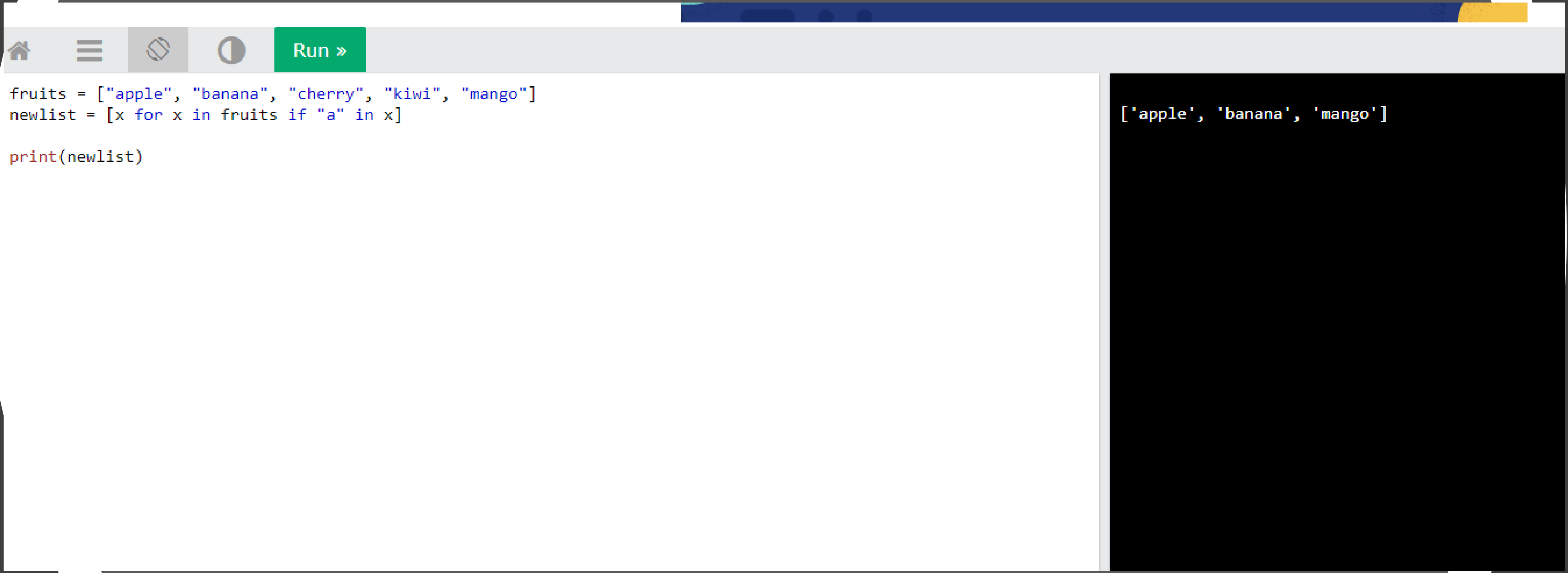
```
mylist = [1,2,3,4,5]

yourlist = [item ** 2 for item in mylist]

print(yourlist)

```

```
[1, 4, 9, 16, 25]
```

Activity: 10.23.1 ActiveCode (listcomp1)

The expression describes each element of the list that is being built. The `for` clause iterates through each item in a sequence. The items are filtered by the `if` clause if there is one. In the example above, the `for` statement lets `item` take on all the values in the list `mylist`. Each item is then squared before it is added to the list that is being built. The result is a list of squares of the values in `mylist`.

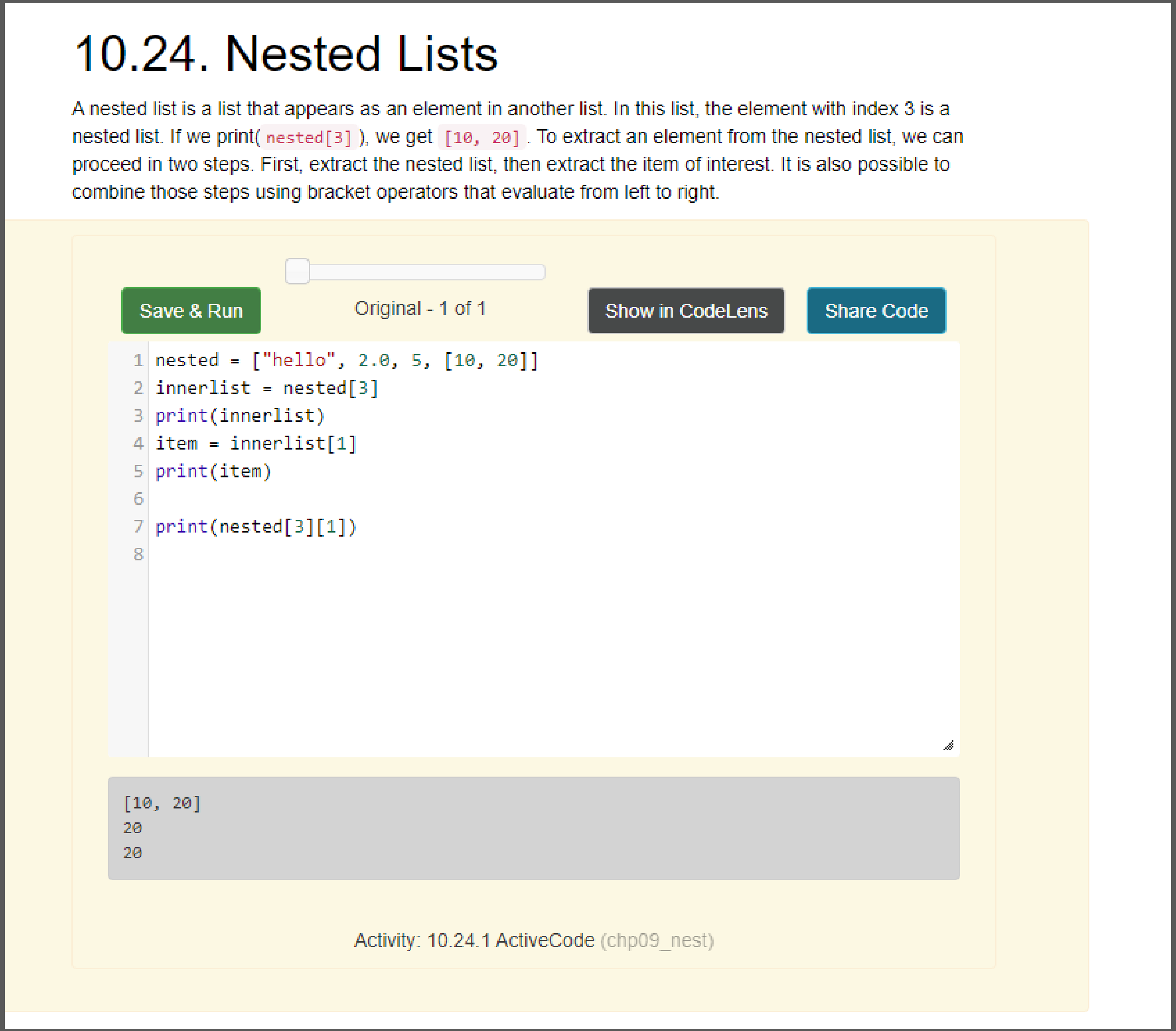
Lists



```python
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = [x for x in fruits if "a" in x]

print(newlist)
```

```
['apple', 'banana', 'mango']
```

Lists



10.24. Nested Lists

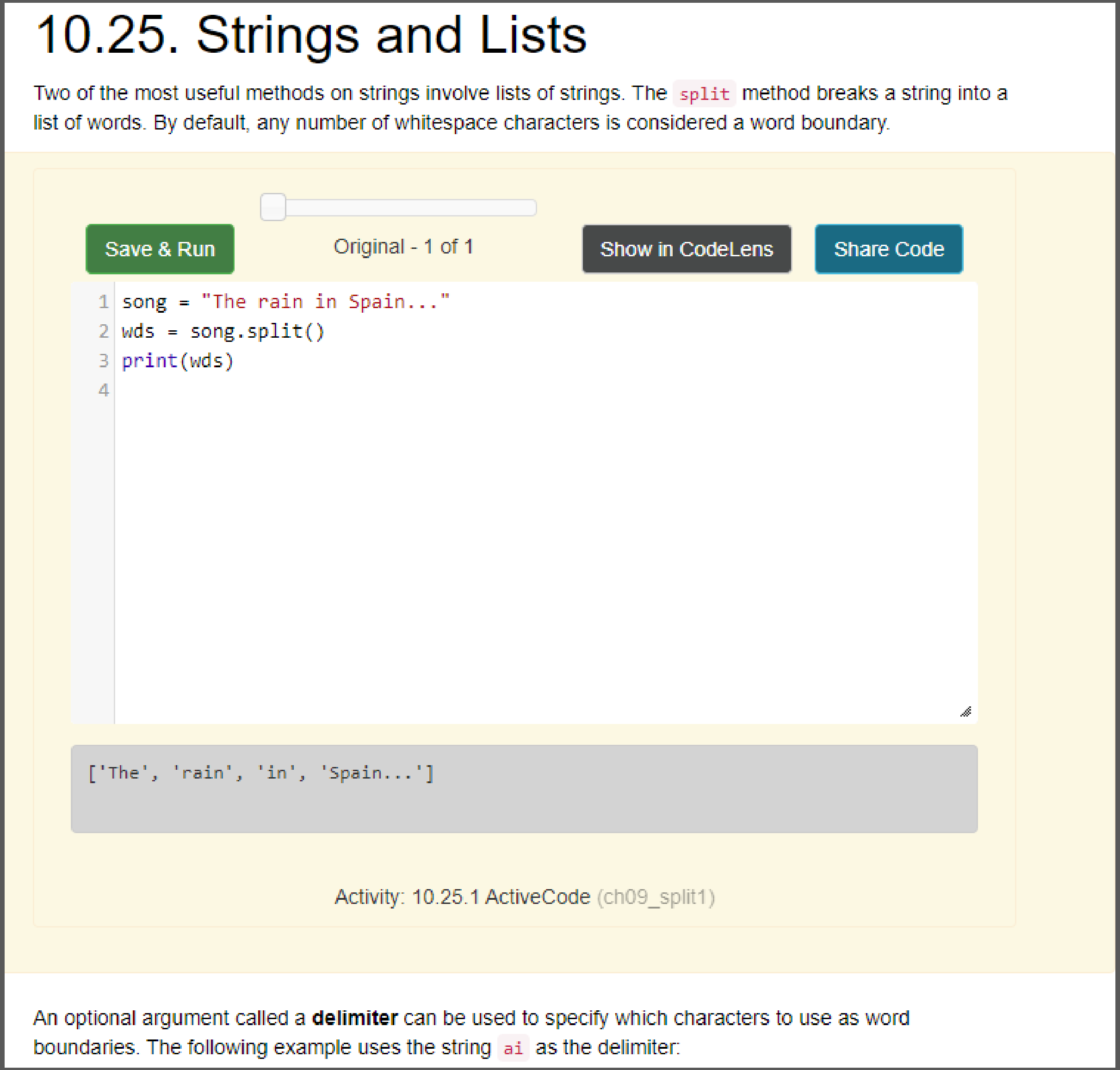
A nested list is a list that appears as an element in another list. In this list, the element with index 3 is a nested list. If we print(`nested[3]`), we get `[10, 20]`. To extract an element from the nested list, we can proceed in two steps. First, extract the nested list, then extract the item of interest. It is also possible to combine those steps using bracket operators that evaluate from left to right.

```python
nested = ["hello", 2.0, 5, [10, 20]]
innerlist = nested[3]
print(innerlist)
item = innerlist[1]
print(item)

print(nested[3][1])
```

```
[10, 20]
20
20
```

Activity: 10.24.1 ActiveCode (chp09_nest)

Lists



10.25. Strings and Lists

Two of the most useful methods on strings involve lists of strings. The `split` method breaks a string into a list of words. By default, any number of whitespace characters is considered a word boundary.

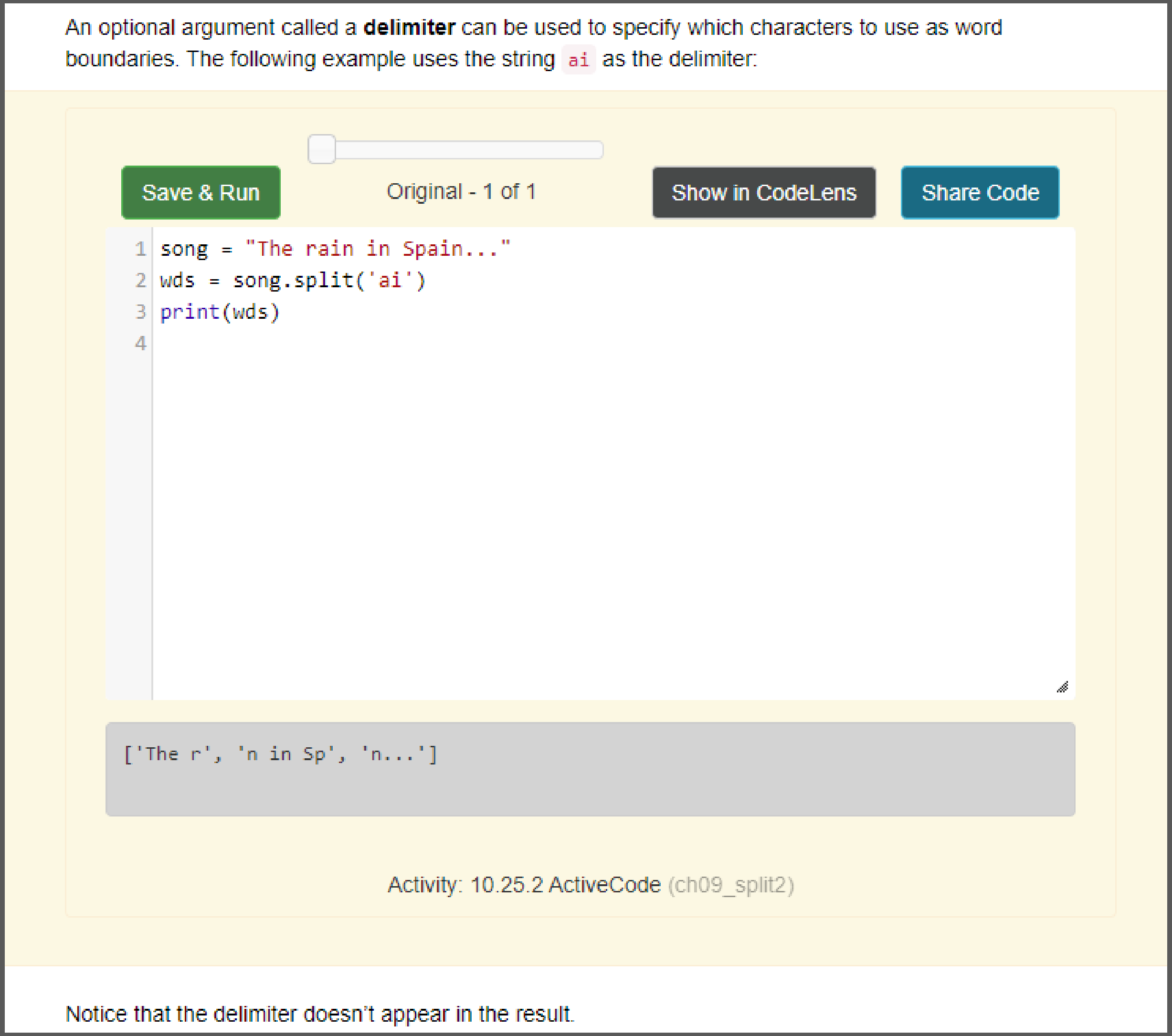
```python
song = "The rain in Spain..."
wds = song.split()
print(wds)

```

```
['The', 'rain', 'in', 'Spain...']
```

Activity: 10.25.1 ActiveCode (ch09_split1)

An optional argument called a **delimiter** can be used to specify which characters to use as word boundaries. The following example uses the string `ai` as the delimiter:

Lists


An optional argument called a **delimiter** can be used to specify which characters to use as word boundaries. The following example uses the string `ai` as the delimiter:

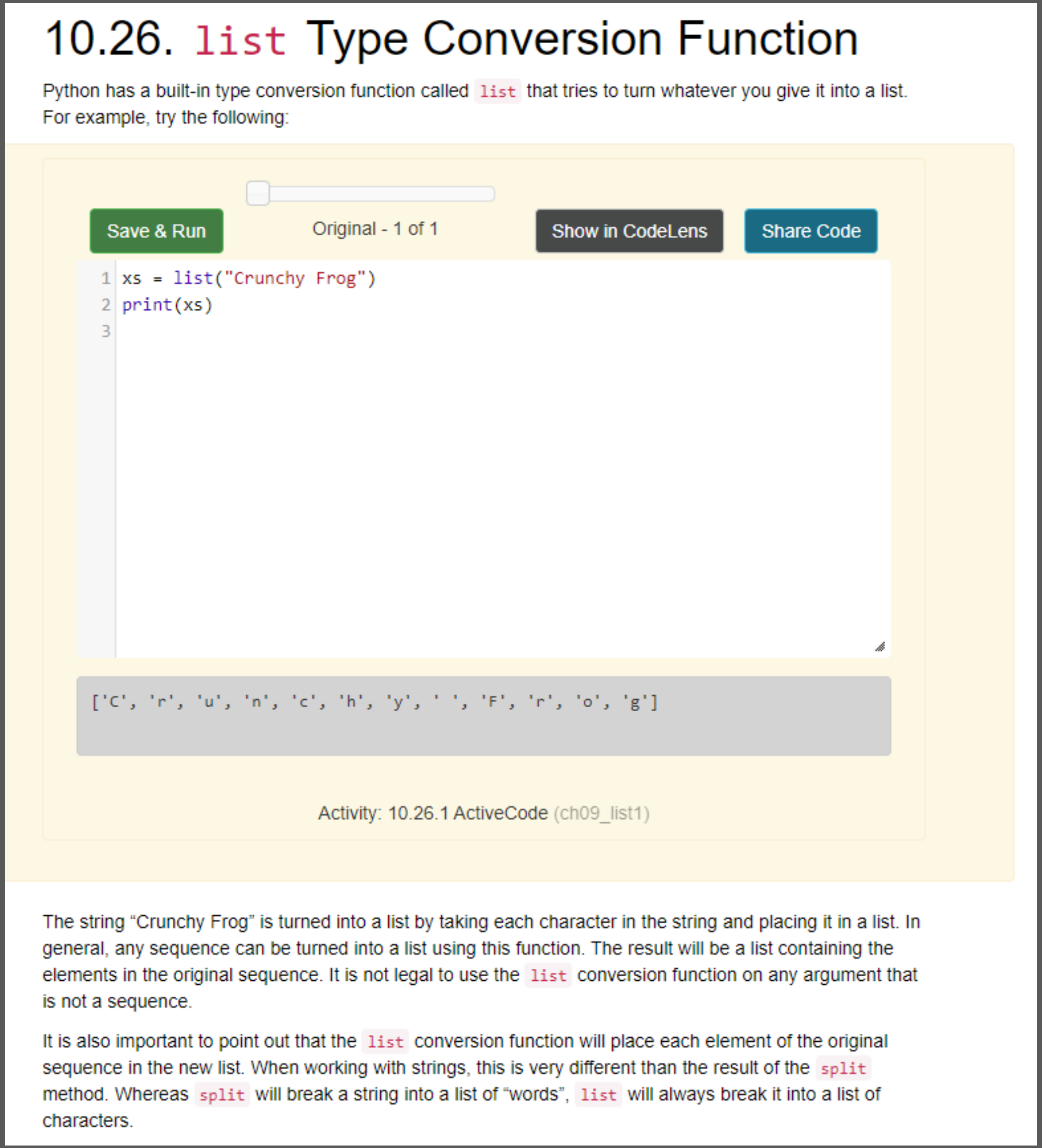
```
song = "The rain in Spain..."
wds = song.split('ai')
print(wds)
```

```
['The r', 'n in Sp', 'n...']
```

Activity: 10.25.2 ActiveCode (ch09_split2)

Notice that the delimiter doesn't appear in the result.

Lists



10.26. `list` Type Conversion Function

Python has a built-in type conversion function called `list` that tries to turn whatever you give it into a list. For example, try the following:

```python
xs = list("Crunchy Frog")
print(xs)
```

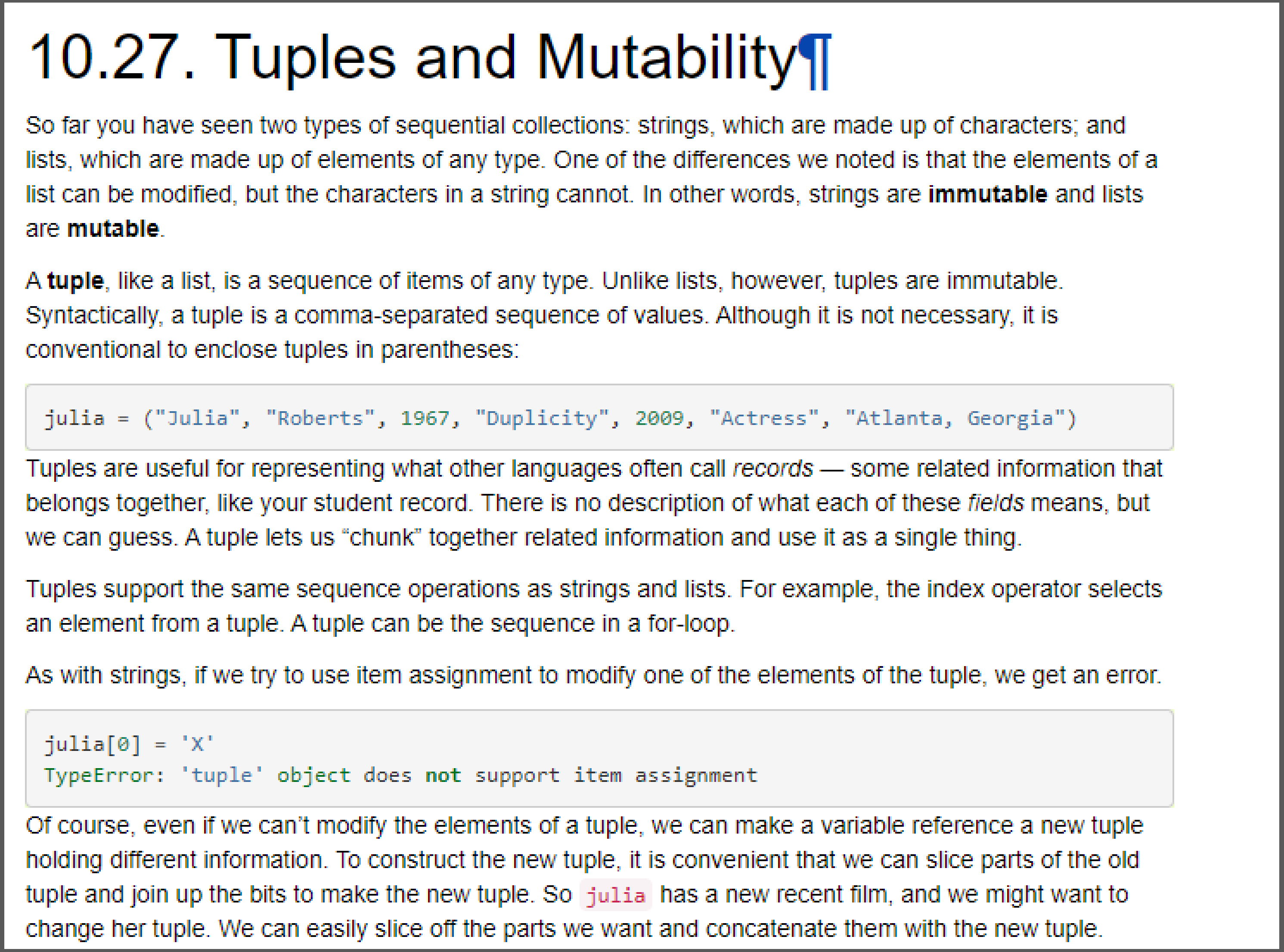
```
['C', 'r', 'u', 'n', 'c', 'h', 'y', ' ', 'F', 'r', 'o', 'g']
```

Activity: 10.26.1 ActiveCode (ch09_list1)

The string “Crunchy Frog” is turned into a list by taking each character in the string and placing it in a list. In general, any sequence can be turned into a list using this function. The result will be a list containing the elements in the original sequence. It is not legal to use the `list` conversion function on any argument that is not a sequence.

It is also important to point out that the `list` conversion function will place each element of the original sequence in the new list. When working with strings, this is very different than the result of the `split` method. Whereas `split` will break a string into a list of “words”, `list` will always break it into a list of characters.

Tuples



10.27. Tuples and Mutability¶

So far you have seen two types of sequential collections: strings, which are made up of characters; and lists, which are made up of elements of any type. One of the differences we noted is that the elements of a list can be modified, but the characters in a string cannot. In other words, strings are **immutable** and lists are **mutable**.

A **tuple**, like a list, is a sequence of items of any type. Unlike lists, however, tuples are immutable. Syntactically, a tuple is a comma-separated sequence of values. Although it is not necessary, it is conventional to enclose tuples in parentheses:

```
julia = ("Julia", "Roberts", 1967, "Duplicity", 2009, "Actress", "Atlanta, Georgia")
```

Tuples are useful for representing what other languages often call *records* — some related information that belongs together, like your student record. There is no description of what each of these *fields* means, but we can guess. A tuple lets us “chunk” together related information and use it as a single thing.

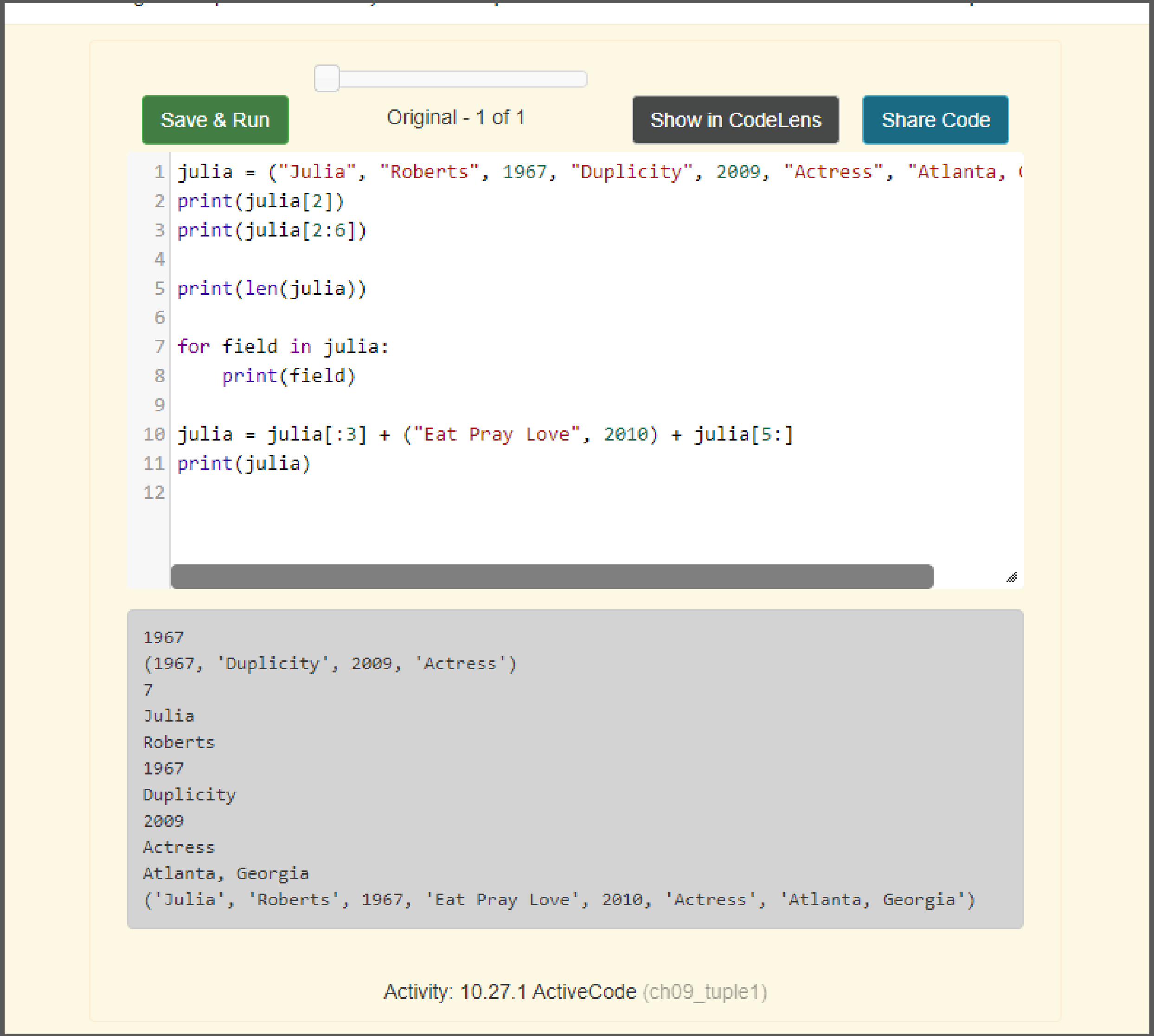
Tuples support the same sequence operations as strings and lists. For example, the index operator selects an element from a tuple. A tuple can be the sequence in a for-loop.

As with strings, if we try to use item assignment to modify one of the elements of the tuple, we get an error.

```
julia[0] = 'X'
TypeError: 'tuple' object does not support item assignment
```

Of course, even if we can’t modify the elements of a tuple, we can make a variable reference a new tuple holding different information. To construct the new tuple, it is convenient that we can slice parts of the old tuple and join up the bits to make the new tuple. So `julia` has a new recent film, and we might want to change her tuple. We can easily slice off the parts we want and concatenate them with the new tuple.

Tuples

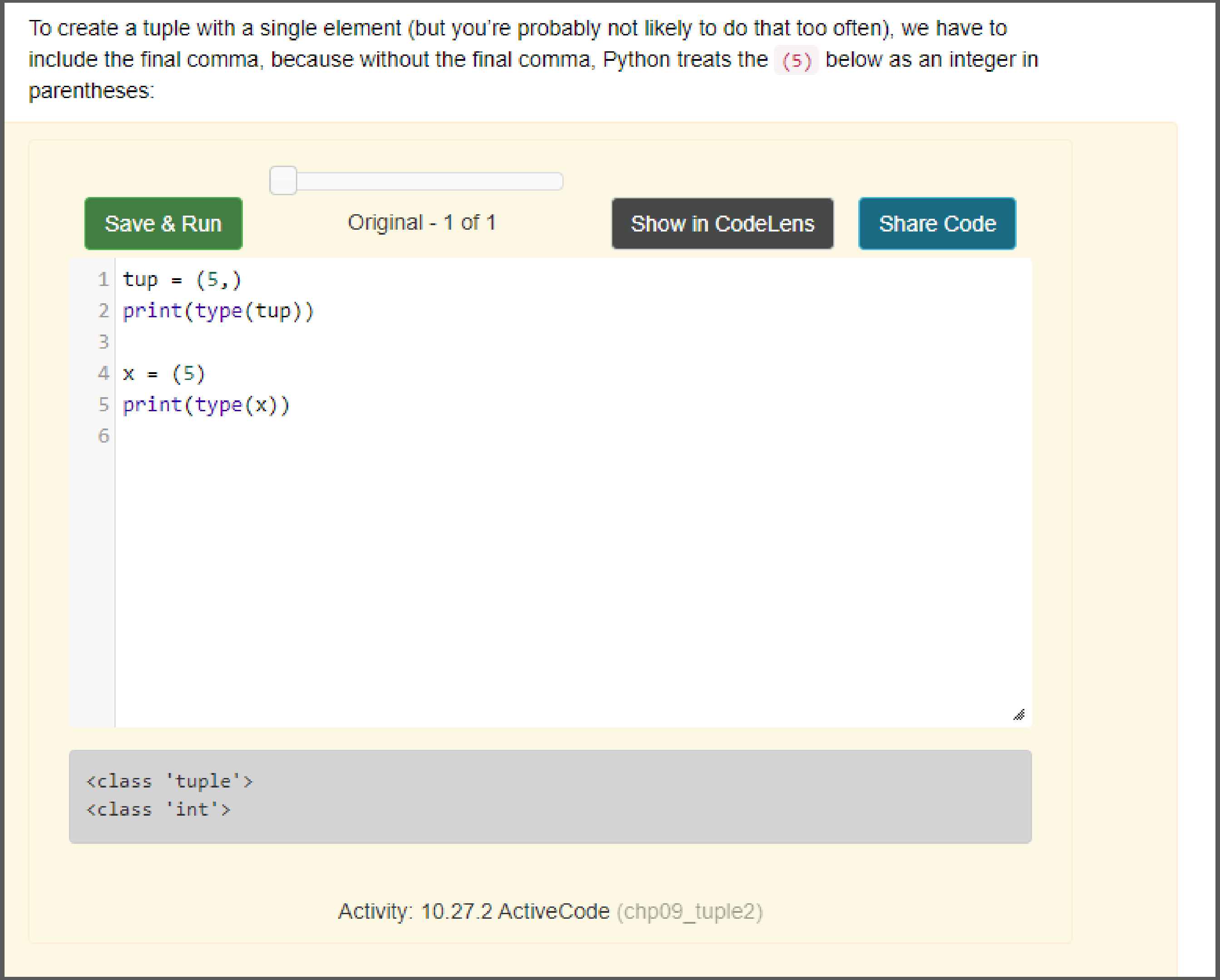


```
julia = ("Julia", "Roberts", 1967, "Duplicity", 2009, "Actress", "Atlanta, (
print(julia[2])
print(julia[2:6])

print(len(julia))

for field in julia:
    print(field)

julia = julia[:3] + ("Eat Pray Love", 2010) + julia[5:]
print(julia)

```

```
1967
(1967, 'Duplicity', 2009, 'Actress')
7
Julia
Roberts
1967
Duplicity
2009
Actress
Atlanta, Georgia
('Julia', 'Roberts', 1967, 'Eat Pray Love', 2010, 'Actress', 'Atlanta, Georgia')
```

Activity: 10.27.1 ActiveCode (ch09_tuple1)

Tuples



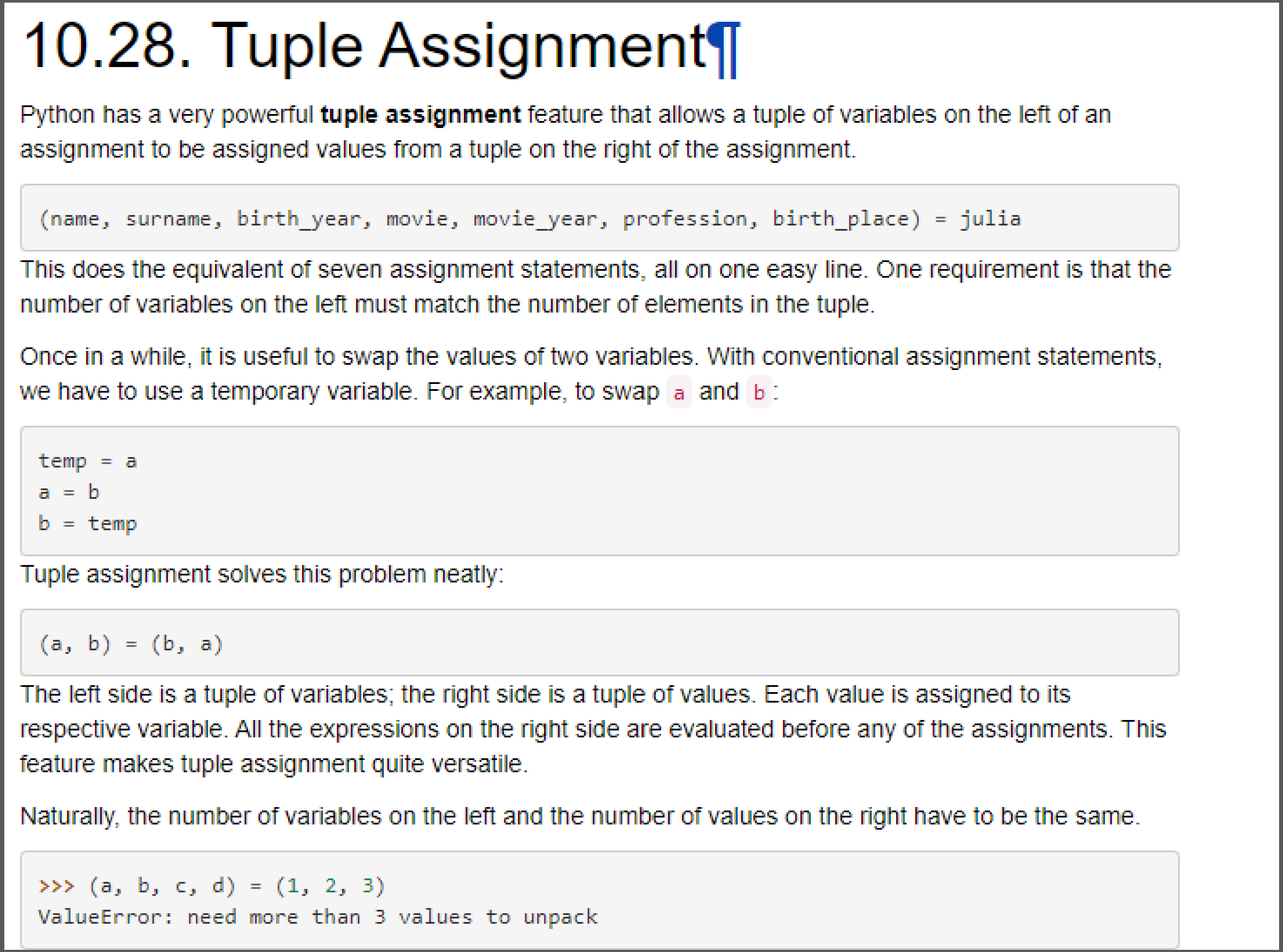
To create a tuple with a single element (but you're probably not likely to do that too often), we have to include the final comma, because without the final comma, Python treats the `(5)` below as an integer in parentheses:

```python
tup = (5,)
print(type(tup))

x = (5)
print(type(x))
```

```
<class 'tuple'>
<class 'int'>
```

Activity: 10.27.2 ActiveCode (chp09_tuple2)

Tuples



10.28. Tuple Assignment¶

Python has a very powerful **tuple assignment** feature that allows a tuple of variables on the left of an assignment to be assigned values from a tuple on the right of the assignment.

```
(name, surname, birth_year, movie, movie_year, profession, birth_place) = julia
```

This does the equivalent of seven assignment statements, all on one easy line. One requirement is that the number of variables on the left must match the number of elements in the tuple.

Once in a while, it is useful to swap the values of two variables. With conventional assignment statements, we have to use a temporary variable. For example, to swap `a` and `b`:

```
temp = a
a = b
b = temp
```

Tuple assignment solves this problem neatly:

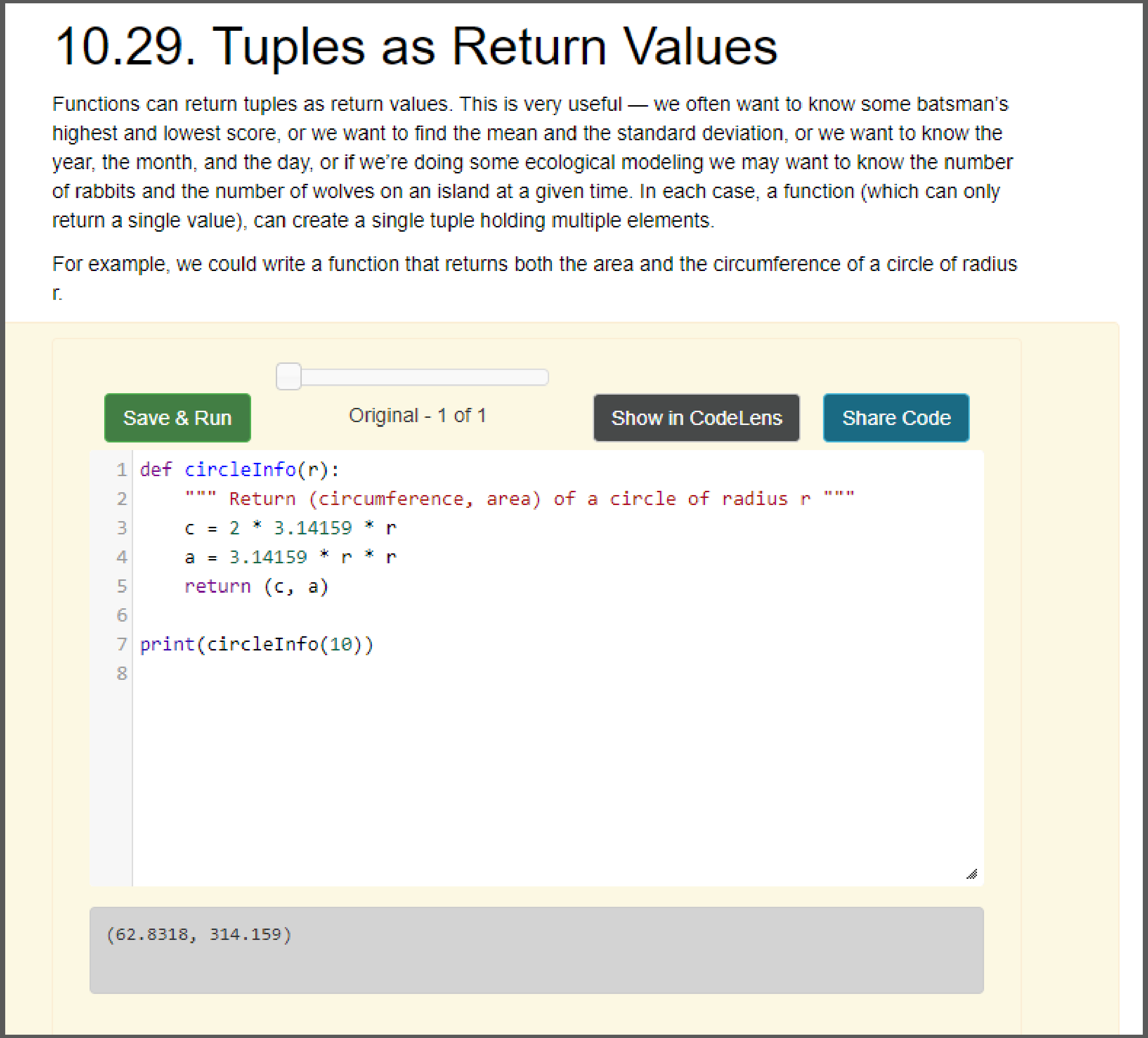
```
(a, b) = (b, a)
```

The left side is a tuple of variables; the right side is a tuple of values. Each value is assigned to its respective variable. All the expressions on the right side are evaluated before any of the assignments. This feature makes tuple assignment quite versatile.

Naturally, the number of variables on the left and the number of values on the right have to be the same.

```
>>> (a, b, c, d) = (1, 2, 3)
ValueError: need more than 3 values to unpack
```

Tuples



10.29. Tuples as Return Values

Functions can return tuples as return values. This is very useful — we often want to know some batsman's highest and lowest score, or we want to find the mean and the standard deviation, or we want to know the year, the month, and the day, or if we're doing some ecological modeling we may want to know the number of rabbits and the number of wolves on an island at a given time. In each case, a function (which can only return a single value), can create a single tuple holding multiple elements.

For example, we could write a function that returns both the area and the circumference of a circle of radius r.

```python
def circleInfo(r):
    """ Return (circumference, area) of a circle of radius r """
    c = 2 * 3.14159 * r
    a = 3.14159 * r * r
    return (c, a)

print(circleInfo(10))
```

```
(62.8318, 314.159)
```