10. Lists

10.1. Lists

10.2. List Values

10.3. List Length

10.4. Accessing Elements

10.5. List Membership

10.6. Concatenation and Repetition

10.7. List Slices

10.8. Lists are Mutable

10.9. List Deletion

10.10. Objects and References

10.11. Aliasing

10.12. Cloning Lists

10.13. Repetition and References

10.14. List Methods

10.16. Append versus Concatenate

10.17. Lists and for loops

10.18. The Accumulator Pattern with Lists

10.18.1. Accumulating the Max Value

10.18.2. Accumulating a String Result

10.19. Using Lists as Parameters

10.20. Pure Functions

10.21. Which is Better?

10.22. Functions that Produce Lists

10.23. List Comprehensions

10.24. Nested Lists

10.25. Strings and Lists

10.26. list Type Conversion Function

10.27. Tuples and Mutability

10.28. Tuple Assignment

10.29. Tuples as Return Values

10.30. Glossary

10.31. Exercises

10.1. 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.

10.2. List Values

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 [].

As you would expect, we can also assign list values to variables and pass lists as parameters to functions.

1 vocabulary = ["iteration", "selection", "control"]

2 numbers = [17, 123]

3 empty = []

4 mixedlist = ["hello", 2.0, 5\*2, [10, 20]]

5

​6 print(numbers)

7 print(mixedlist)

8 newlist = [ numbers, vocabulary ]

9 print(newlist)

10

​

Check your understanding

list-2-2: A list can contain only integer items.

A. False

B. True

10.3. List Length

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 is 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.

1 alist = ["hello", 2.0, 5, [10, 20]]

2 print(len(alist))

3 print(len(['spam!', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]))

4

Check your understanding

list-3-2: What is printed by the following statements?

alist = [3, 67, "cat", 3.14, False]

print(len(alist))

A. 4

B. 5

list-3-3: What is printed by the following statements?

alist = [3, 67, "cat", [56, 57, "dog"], [ ], 3.14, False]

print(len(alist))

A. 7

B. 8

10.4. Accessing Elements

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.

1 numbers = [17, 123, 87, 34, 66, 8398, 44]

2 print(numbers[2])

3 print(numbers[9 - 8])

4 print(numbers[-2])

5 print(numbers[len(numbers) - 1])

6

​

Check your understanding

list-4-2: What is printed by the following statements?

alist = [3, 67, "cat", [56, 57, "dog"], [ ], 3.14, False]

print(alist[5])

A. [ ]

B. 3.14

C. False

list-4-3: What is printed by the following statements?

alist = [3, 67, "cat", [56, 57, "dog"], [ ], 3.14, False]

print(alist[2].upper())

A. Error, you cannot use the upper method on a list.

B. 2

C. CAT

list-4-4: What is printed by the following statements?

alist = [3, 67, "cat", [56, 57, "dog"], [ ], 3.14, False]

print(alist[2][0])

A. 56

B. c

C. cat

D. Error, you cannot have two index values unless you are using slicing.

10.5. List Membership

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

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

2

​3 print("apple" in fruit)

4 print("pear" in fruit)

5

Check your understanding

list-5-2: What is printed by the following statements?

alist = [3, 67, "cat", [56, 57, "dog"], [ ], 3.14, False]

print(3.14 in alist)

A. True

B. False

list-5-3: What is printed by the following statements?

alist = [3, 67, "cat", [56, 57, "dog"], [ ], 3.14, False]

print(57 in alist)

A. True

B. False

10.6. Concatenation and Repetition

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

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

2 print([1, 2] + [3, 4])

3 print(fruit + [6, 7, 8, 9])

4

​5 print([0] \* 4)

6 print([1, 2, ["hello", "goodbye"]] \* 2)

7

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.

One way for us to make this more clear is to run a part of this example in codelens. As you step through the code, you will see the variables being created and the lists that they refer to. Pay particular attention to the fact that when newlist is created by the statement newlist = fruit + numlist, it refers to a completely new list formed by making copies of the items from fruit and numlist.

In Python, every object has a unique identification tag. Likewise, there is a built-in function that can be called on any object to return its unique id. The function is appropriately called id and takes a single parameter, the object that you are interested in knowing about. You can see in the example below that a real id is usually a very large integer value (corresponding to an address in memory).

>>> alist = [4, 5, 6]

>>> id(alist)

4300840544

>>>

Check your understanding

list-6-3: What is printed by the following statements?

alist = [1, 3, 5]

blist = [2, 4, 6]

print(alist + blist)

A. 6

B. [1, 2, 3, 4, 5, 6]

C. [1, 3, 5, 2, 4, 6]

D. [3, 7, 11]

list-6-4: What is printed by the following statements?

alist = [1, 3, 5]

print(alist \* 3)

A. 9

B. [1, 1, 1, 3, 3, 3, 5, 5, 5]

C. [1, 3, 5, 1, 3, 5, 1, 3, 5]

D. [3, 9, 15]

10.7. List Slices

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.

1 a\_list = ['a', 'b', 'c', 'd', 'e', 'f']

2 print(a\_list[1:3])

3 print(a\_list[:4])

4 print(a\_list[3:])

5 print(a\_list[:])

6

Check your understanding

list-7-2: What is printed by the following statements?

alist = [3, 67, "cat", [56, 57, "dog"], [ ], 3.14, False]

print(alist[4:])

A. [ [ ], 3.14, False]

B. [ [ ], 3.14]

C. [ [56, 57, "dog"], [ ], 3.14, False]

10.8. Lists are Mutable

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.

1 fruit = ["banana", "apple", "cherry"]

2 print(fruit)

3

​4 fruit[0] = "pear"

5 fruit[-1] = "orange"

6 print(fruit)

7

An assignment to an element of a list is called item assignment. Item assignment does not work for strings. Recall that strings are immutable.

By combining assignment with the slice operator we can update several elements at once.

1 alist = ['a', 'b', 'c', 'd', 'e', 'f']

2 alist[1:3] = ['x', 'y']

3 print(alist)

4

We can also remove elements from a list by assigning the empty list to them.

1 alist = ['a', 'b', 'c', 'd', 'e', 'f']

2 alist[1:3] = []

3 print(alist)

4

​

We can even insert elements into a list by squeezing them into an empty slice at the desired location.

1 alist = ['a', 'd', 'f']

2 alist[1:1] = ['b', 'c']

3 print(alist)

4 alist[4:4] = ['e']

5 print(alist)

6

​

Check your understanding

list-8-6: What is printed by the following statements?

alist = [4, 2, 8, 6, 5]

alist[2] = True

print(alist)

A. [4, 2, True, 8, 6, 5]

B. [4, 2, True, 6, 5]

C. Error, it is illegal to assign

10.9. List Deletion

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.

1

a = ['one', 'two', 'three']

2 del a[1]

3 print(a)

4

​5 alist = ['a', 'b', 'c', 'd', 'e', 'f']

6 del alist[1:5]

7 print(alist)

8

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.

10.10. Objects and References

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:

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.

1 a = "banana"

2 b = "banana"

3

​4 print(a is b)

5

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.

1

a = [81, 82, 83]

2

b = [81, 82, 83]

3

​4 print(a is b)

5

​6 print(a == b)

7

​

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.

10.11. Aliasing

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

1 a = [81, 82, 83]

2 b = a

3 print(a is b)

4

Because the same list has two different names, a and b, we say that it is aliased. Changes made with one alias affect the other. In the example below, you can see that a and b refer to the same list after executing the assignment statement b = a.

1 a = [81, 82, 83]

2 b = [81, 82, 83]

3

4 print(a == b)

5 print(a is b)

6

7 b = a

8 print(a == b)

9 print(a is b)

10

11 b[0] = 5

12 print(a)

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.

Check your understanding

list-11-3: What is printed by the following statements?

alist = [4, 2, 8, 6, 5]

blist = alist

blist[3] = 999

print(alist)

A. [4, 2, 8, 6, 5]

B. [4, 2, 8, 999, 5]

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.

1 a = [81, 82, 83]

2

3 b = a[:] # make a clone using slice

4 print(a == b)

5 print(a is b)

6

7 b[0] = 5

8

9 print(a)

10 print(b)

Now we are free to make changes to b without worrying about a. Again, we can clearly see in codelens that a and b are entirely different list objects.

SavvyPython

10.13. Repetition and References

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

1 origlist = [45, 76, 34, 55]

2 print(origlist \* 3)

3

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.

Consider the following extension on the previous example.

1 origlist = [45, 76, 34, 55]

2 print(origlist \* 3)

3

​4newlist = [origlist] \* 3

5

​6 print(newlist)

7

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

Repetition of a nested list

Now, what happens if we modify a value in origlist.

1 origlist = [45, 76, 34, 55]

2

​3 newlist = [origlist] \* 3

4

​5 print(newlist)

6

​7 origlist[1] = 99

8

​9 print(newlist)

10

newlist shows the change in three places. This can easily be seen by noting that in the reference diagram, there is only one origlist, so any changes to it appear in all three references from newlist.

Check your understanding

list-13-5: What is printed by the following statements?

alist = [4, 2, 8, 6, 5]

blist = alist \* 2

blist[3] = 999

print(alist)

A. [4, 2, 8, 999, 5, 4, 2, 8, 6, 5]

B. [4, 2, 8, 999, 5]

C. [4, 2, 8, 6, 5]

list-13-6: What is printed by the following statements?

alist = [4, 2, 8, 6, 5]

blist = [alist] \* 2

alist[3] = 999

print(blist)

A. [4, 2, 8, 999, 5, 4, 2, 8, 999, 5]

B. [[4, 2, 8, 999, 5], [4, 2, 8, 999, 5]]

C. [4, 2, 8, 6, 5]

D. [[4, 2, 8, 999, 5], [4, 2, 8, 6, 5]]

10.14. List Methods

The dot operator can also be used to access built-in methods of list objects. append is a list method which adds the argument passed to it to the end of the list. Continuing with this example, we show several other list methods. Many of them are easy to understand.

1 mylist = []

2 mylist.append(5)

3 mylist.append(27)

4 mylist.append(3)

5 mylist.append(12)

6 print(mylist)

7

​8 mylist.insert(1, 12)

9 print(mylist)

10 print(mylist.count(12))

11

​12 print(mylist.index(3))

13 print(mylist.count(5))

14

​15 mylist.reverse()

16 print(mylist)

17

​18 mylist.sort()

19 print(mylist)

20

​21 mylist.remove(5)

22 print(mylist)

23

​

24

lastitem = mylist.pop()

There are two ways to use the pop method. The first, with no parameter, will remove and return the last item of the list. If you provide a parameter for the position, pop will remove and return the item at that position. Either way the list is changed.

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

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 will likely never appear as part of an assignment statement (see line 8 below).

1 mylist = []

2 mylist.append(5)

3 mylist.append(27)

4 mylist.append(3)

5 mylist.append(12)

6 print(mylist)

7

​8 mylist = mylist.sort() #probably an error

9 print(mylist)

10

​

Check your understanding

list-14-3: What is printed by the following statements?

alist = [4, 2, 8, 6, 5]

alist.append(True)

alist.append(False)

print(alist)

A. [4, 2, 8, 6, 5, False, True]

B. [4, 2, 8, 6, 5, True, False]

C. [True, False, 4, 2, 8, 6, 5]

list-14-4: What is printed by the following statements?

alist = [4, 2, 8, 6, 5]

alist.insert(2, True)

alist.insert(0, False)

print(alist)

A. [False, 4, 2, True, 8, 6, 5]

B. [4, False, True, 2, 8, 6, 5]

C. [False, 2, True, 6, 5]

list-14-5: What is printed by the following statements?

alist = [4, 2, 8, 6, 5]

temp = alist.pop(2)

temp = alist.pop()

print(alist)

A. [4, 8, 6]

B. [2, 6, 5]

C. [4, 2, 6]

list-14-6: What is printed by the following statements?

alist = [4, 2, 8, 6, 5]

alist = alist.pop(0)

print(alist)

A. [2, 8, 6, 5]

B. [4, 2, 8, 6, 5]

C. 4

D. None

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.

1 origlist = [45, 32, 88]

2

3 origlist.append("cat")

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.

1 origlist = [45, 32, 88]

2

3 origlist = origlist + ["cat"]

Check you understanding

list-16-4: What is printed by the following statements?

alist = [4, 2, 8, 6, 5]

alist = alist + 999

print(alist)

A. [4, 2, 8, 6, 5, 999]

B. Error, you cannot concatenate a list with an integer.

SavvyPython

View:

Textbook

Social

Search

User

Scratch Activecode

Help

This Chapter

10.17. Lists and for loops¶

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

1

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

2

​

3

for afruit in fruits: # by item

4

print(afruit)

5

​

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.

1

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

2

​

3

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

4

print(fruits[position])

5

​

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.

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

1

for number in range(20):

2

if number % 3 == 0:

3

print(number)

4

​

Activity: 10.17.3 ActiveCode (chp09\_for3)

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

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.

1

numbers = [1, 2, 3, 4, 5]

2

print(numbers)

3

​

4

for i in range(len(numbers)):

5

numbers[i] = numbers[i] \*\* 2

6

​

7

print(numbers)

8

​

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.

Check your understanding

list-17-5: What is printed by the following statements?

alist = [4, 2, 8, 6, 5]

blist = [ ]

for item in alist:

blist.append(item+5)

print(blist)

A. [4, 2, 8, 6, 5]

B. [4, 2, 8, 6, 5, 5]

C. [9, 7, 13, 11, 10]

D. Error, you cannot concatenate inside an append.

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

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.

Sometimes when we’re accumulating, we don’t want to add to our accumulator every time we iterate. Consider, for example, the following program which counts the number of names with more than 3 letters.

1 long\_names = 0

2 for name in ["Joe", "Sally", "Amy", "Brad"]:

3 if len(name) > 3:

4 long\_names += 1

5 print(long\_names)

6

Here, we initialize the accumulator variable to be zero on line 1.

We iterate through the sequence (line 2).

The update step happens in two parts. First, we check to see if the name is longer than 3 letters. If so, then we increment the accumulator variable long\_names (on line 4) by adding one to it.

At the end, we have accumulated the total number of long names.

We can use conditionals to also count if particular items are in a string or list. The following code finds all occurrences of vowels in a string.

1 s = "what if we went to the zoo"

2 num\_vowels = 0

3 for i in s:

4 f i in ['a', 'e', 'i', 'o', 'u']:

5 num\_vowels += 1

6 print(num\_vowels)

7

We can also use == to execute a similar operation. Here, we’ll check to see if the character we are iterating over is an “o”. If it is an “o” then we will update our counter.

a gif that shows code to check that "o" is in the phrase "onomatopoeia".

10.18.1. Accumulating the Max Value

We can also use the accumulation pattern with conditionals to find the maximum or minimum value. Instead of continuing to build up the accumulator value like we have when counting or finding a sum, we can reassign the accumulator variable to a different value.

The following example shows how we can get the maximum value from a list of integers.

1 nums = [9, 3, 8, 11, 5, 29, 2]

2 best\_num = 0

3 for n in nums:

4 if n > best\_num:

5 best\_num = n

6 print(best\_num)

7

Here, we initialize best\_num to zero, assuming that there are no negative numbers in the list.

In the for loop, we check to see if the current value of n is greater than the current value of best\_num. If it is, then we want to update best\_num so that it now is assigned the higher number. Otherwise, we do nothing and continue the for loop.

You may notice that the current structure could be a problem. If the numbers were all negative what would happen to our code? What if we were looking for the smallest number but we initialized best\_num with zero? To get around this issue, we can initialize the accumulator variable using one of the numbers in the list.

1 nums = [9, 3, 8, 11, 5, 29, 2]

2 best\_num = nums[0]

3 for n in nums:

4 if n > best\_num:

5 best\_num = n

6 print(best\_num)

7

The only thing we changed was the value of best\_num on line 2 so that the value of best\_num is the first element in nums, but the result is still the same!

10.18.2. Accumulating a String Result

The accumulator pattern can be used to convert a list of items to a string.

Consider the following program:

1 scores = [85, 95, 70]

2 result = ''

3 for score in scores:

4 result = result + str(score) + ','

5

​6 print("The scores are " + result)

7

Here, the accumulator variable is a result. Each time through the loop, the program concatenates the current contents of result with the comma separator and a score from the list, and updates the result with the new value. Use CodeLens to step through this example to see it in action.

The output of the program has some undesirable formatting problems: there is a trailing comma instead of a period, and there are no spaces between the items. The next activity lets you work to correct those problems.

Question

Tip

Answer

Let’s work to improve the formatting of the sentence produced by the program above. Revise the following code so that it outputs the sentence:

The scores are 85, 95, and 70.

1 scores = [85, 95, 70]

2 result = ''

3 for score in scores:

4 result = result + str(score) + ','

5

6 print("The scores are " + result)

7

​8​

Check your understanding

list-29-9: What is printed by the following statements?

s = "We are learning!"

x = 0

for i in s:

if i in ['a', 'b', 'c', 'd', 'e']:

x += 1

print(x)

A. 2

B. 5

C. 0

D. There is an error in the code so it cannot run.

list-29-10: What is printed by the following statements?

list= [5, 2, 1, 4, 9, 10]

min\_value = 0

for item in list:

if item < min\_value:

min\_value = item

print(min\_value)

A. 10

B. 1

C. 0

D. There is an error in the code so it cannot run.

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.

1 words = ["adopt", "bake", "beam", "confide", "grill", "plant", "time", "wave", "wish"]

2

​3

​

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:

1 def doubleStuff(aList):

2 """ Overwrite each element in aList with double its value. """

3 for position in range(len(aList)):

4 aList[position] = 2 \* aList[position]

5

​6 things = [2, 5, 9]

7 print(things)

8 doubleStuff(things)

9 print(things)

10

​

Activity: 10.19.1 ActiveCode (chp09\_parm1)

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

Since the list object is shared by two references, there is only one copy. If a function modifies the elements of a list parameter, the caller sees the change since the change is occurring to the original.

SavvyPython

View:

Textbook

Social

Search

User

Scratch Activecode

Help

This Chapter

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.

1 def doubleStuff(a\_list):

2 """ Return a new list in which contains doubles of the elements in a\_list. """

3 new\_list = []

4 for value in a\_list:

5 new\_elem = 2 \* value

6 new\_list.append(new\_elem)

7 return new\_list

8

​9 things = [2, 5, 9]

10 print(things)

11 things = doubleStuff(things)

12 print(things)

13

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.

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:

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

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,

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

2

​3 yourlist = [item \*\* 2 for item in mylist]

4

​5 print(yourlist)

6

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.

To write the primes\_upto function we will use the is\_prime function to filter the sequence of integers coming from the range function. In other words, for every integer from 2 up to but not including n, if the integer is prime, keep it in the list.

def primes\_upto(n):

""" Return a list of all prime numbers less than n using a list comprehension. """

result = [num for num in range(2,n) if is\_prime(num)]

return result

Check your understanding

list-22-3: What is printed by the following statements?

alist = [4,2,8,6,5]

blist = [num\*2 for num in alist if num%2==1]

print(blist)

A. [4,2,8,6,5]

B. [8,4,16,12,10]

C. 10

D. [10].

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.

1 nested = ["hello", 2.0, 5, [10, 20]]

2 innerlist = nested[3]

3 print(innerlist)

4 item = innerlist[1]

5 print(item)

6

​7 print(nested[3][1])

8

Check your understanding

list-23-2: What is printed by the following statements?

alist = [ [4, [True, False], 6, 8], [888, 999] ]

if alist[0][1][0]:

print(alist[1][0])

else:

print(alist[1][1])

A. 6

B. 8

C. 888

D. 999

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.

1 song = "The rain in Spain..."

2 wds = song.split()

3 print(wds)

4

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:

1 song = "The rain in Spain..."

2 wds = song.split('ai')

3 print(wds)

4

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

The inverse of the split method is join. You choose a desired separator string, (often called the glue) and join the list with the glue between each of the elements.

1 wds = ["red", "blue", "green"]

2glue = ';'

3 s = glue.join(wds)

4 print(s)

5 print(wds)

6

​7 print("\*\*\*".join(wds))

8 print("".join(wds))

9

​

The list that you glue together (wds in this example) is not modified. Also, you can use empty glue or multi-character strings as glue.

Check your understanding

list-24-4: What is printed by the following statements?

myname = "Edgar Allan Poe"

namelist = myname.split()

init = ""

for aname in namelist:

init = init + aname[0]

print(init)

A. Poe

B. EdgarAllanPoe

C. EAP

D. William Shakespeare

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:

1 xs = list("Crunchy Frog")

2 print(xs)

3

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.

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.

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

2 print(julia[2])

3 print(julia[2:6])

4

​5 print(len(julia))

6

​7 for field in julia:

8 print(field)

9

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

11 print(julia)

12

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:

1 tup = (5,)

2 print(type(tup))

3

​4 x = (5)

5 print(type(x))

6

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

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.

1 def circleInfo(r):

2 """ Return (circumference, area) of a circle of radius r """

3 c = 2 \* 3.14159 \* r

4 a = 3.14159 \* r \* r

5 return (c, a)

6

​7 print(circleInfo(10))

8

​

10.30. Glossary

aliases

Multiple variables that contain references to the same object.

clone

To create a new object that has the same value as an existing object. Copying a reference to an object creates an alias but doesn’t clone the object.

delimiter

A character or string used to indicate where a string should be split.

element

One of the values in a list (or other sequence). The bracket operator selects elements of a list.

index

An integer variable or value that indicates an element of a list.

list

A collection of objects, where each object is identified by an index. Like other types str, int, float, etc. there is also a list type-converter function that tries to turn its argument into a list.

list traversal

The sequential accessing of each element in a list.

modifier

A function which changes its arguments inside the function body. Only mutable types can be changed by modifiers.

mutable data type

A data type in which the elements can be modified. All mutable types are compound types. Lists are mutable data types; strings are not.

nested list

A list that is an element of another list.

object

A thing to which a variable can refer.

pattern

A sequence of statements, or a style of coding something that has general applicability in a number of different situations. Part of becoming a mature Computer Scientist is to learn and establish the patterns and algorithms that form your toolkit. Patterns often correspond to your “mental chunking”.

pure function

A function which has no side effects. Pure functions only make changes to the calling program through their return values.

sequence

Any of the data types that consist of an ordered collection of elements, with each element identified by an index.

side effect

A change in the state of a program made by calling a function that is not a result of reading the return value from the function. Side effects can only be produced by modifiers.

tuple

A sequential collection of items, similar to a list. Any python object can be an element of a tuple. However, unlike a list, tuples are immutable.

10.31. Exercises

a = [1, 2, 3]

b = a[:]

b[0] = 5

Create a list called myList with the following six items: 76, 92.3, “hello”, True, 4, 76. Begin with the empty list shown below, and add 6 statements to add each item, one per item. The first three statements should use the append method to append the item to the list, and the last three statements should use concatenation.

Starting with the list of the previous exercise, write Python statements to do the following:

Append “apple” and 76 to the list.

Insert the value “cat” at position 3.

Insert the value 99 at the start of the list.

Find the index of “hello”.

Count the number of 76s in the list.

Remove the first occurrence of 76 from the list.

Remove True from the list using pop and index.

Write a function called average that takes a list of numbers as a parameter and returns the average of the numbers.

Write a Python function named max that takes a parameter containing a nonempty list of integers and returns the maximum value. (Note: there is a builtin function named max but pretend you cannot use it.)

Write a function sum\_of\_squares(xs) that computes the sum of the squares of the numbers in the list xs. For example, sum\_of\_squares([2, 3, 4]) should return 4+9+16 which is 29:

Write a function to count how many odd numbers are in a list.

Sum up all the even numbers in a list.

Sum up all the negative numbers in a list.

Count how many words in a list have length 5.

Sum all the elements in a list up to but not including the first even number.

Count how many words occur in a list up to and including the first occurrence of the word “sam”.

Although Python provides us with many list methods, it is good practice and very instructive to think about how they are implemented. Implement a Python function that works like the following:

count

in

reverse

index

insert

Write a function replace(s, old, new) that replaces all occurences of old with new in a string s:

test(replace('Mississippi', 'i', 'I'), 'MIssIssIppI')

s = 'I love spom! Spom is my favorite food. Spom, spom, spom, yum!'

test(replace(s, 'om', 'am'),

'I love spam! Spam is my favorite food. Spam, spam, spam, yum!')

test(replace(s, 'o', 'a'),

'I lave spam! Spam is my favarite faad. Spam, spam, spam, yum!')

Hint: use the split and join methods.

Create a list named randlist containing 100 random integers between 0 and 1000 (use iteration, append, and the random module).