|  |  |
| --- | --- |
| Lab 9: for-loops | Name: Lex Baker  Date: 10/18/22 |

A **for-loop** is a construct that makes it slightly easier and more intuitive to loop a definite number of times. For example, you often want to do a particular operation on every element of a tuple or string, so the number of passes is equal to the number of elements. It turns out that for-loops are not strictly required (anything they can do could also be done with a while-loop), but they are convenient enough to be part of every programming language.

**1.** A **range object** is just a set of ordered numbers. It is similar to the arrange function you used with NumPy but it can only contain integers. We will look at them because they are useful with for-loops. A range object is a sequence because...

* you use one variable name for multiple values
* the values are ordered
* you access the elements by indexing

**a.** Run the following commands and write down any output.

>>> index = range(10)

>>> index[0]

>>> index[1]

>>> index[9]

>>> index[10]

>>> print(index)

>>> print(tuple(index))

**0**

**1**

**9**

**IndexError: range object index out of range**

**range(0, 10)**

**(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)**

**b.** The first command calls the function range and passes it an argument of 10. A range object is created and assigned to the variable index. What does the argument 10 specify about the range object that is created?

**It will have 10 integers inside of it, starting with 0 and going to one less than the argument provided, in this case 10. Here, it goes from 0 to 9.**

**c.** Why do you think the command index[10] failed?

**Because the range function created 10 elements, and [10] calls the eleventh element, which does not exist.**

**d.** Describe what is happening with the command print(tuple(index))?

**Python turns the range(0, 10) into a tuple, which is then printed to console. Turning it into a tuple makes it (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)**

**2.** **Details of the range() function**

**a.** Enter the following commands and write down the results

>>> even\_nums = range(0,10,2)

>>> even\_nums[0]

>>> even\_nums[1]

>>> even\_nums[2]

>>> len(even\_nums)

>>> tuple(even\_nums)

**0**

**2**

**4**

**5**

**(0, 2, 4, 6, 8)**

**b.** Create a range object named my\_range that starts as 3, goes up by four between elements, and stops at 15. Write the relevant statement below, and run some commands to see that it worked.

**my\_range = range(3, 15, 4) # 15 exclusive**

**my\_range = range(3, 16, 4) # 15 inclusive**

**c.** See if you can create a range object (again called my\_range) that has 6 elements, starts at 5, and goes down by 1 between elements. Write the relevant statement below and run commands to see that it worked.

**my\_range = range(5, -1, -1)**

**d.** Test to see if it is possible to have a range object where the step between adjacent elements is non-integer. Briefly describe what you did and what you found

**It is not possible, as all three arguments of range() are interpreted as integers, and so the program throws an error when a non-integer is used, such as with:**

**range(0, 5, .5)**

**3.** **Range objects are like tuples, but they are NOT tuples**

**a.** Try to concatenate two of the range objects you have created. Write down the relevant part of the resulting error message.

**TypeError: unsupported operand type(s) for +: 'range' and 'range'**

Long lists of evenly spaced integers are often useful, but there is no point in occupying a bunch of memory for something like all the even integers from 0 to 1,000,000. Instead, a range object “knows” how to use the formula T*n*= 2*n* to generate on-the-fly whichever element is needed, and it “knows” that the smallest *n* is 0 and the largest *n* is 500,000. For long ranges, remembering the instructions is vastly more efficient than remembering all the individual elements.

**4.** **for-loops**

Create a new program file and save it in your sandbox as for\_fun.py.

**a.** Type in and run the following code to create your first for-loop

evens\_2\_to\_20 = range(2,22,2)

for ii in evens\_2\_to\_20 :

print(ii)

The first line creates a range object and assigns it to the variable evens\_2\_to\_20. The second line is the key. The keyword for tells Python that you want a for-loop. The variable ii is referred to as the **loop variable** or loop counter. In Python, the loop variable sequentially takes on all the values present in the sequence that comes right after the keyword in. In our case, that sequence is the range object evens\_2\_to\_20.

This first time through the loop, ii has a value 2 because that is the first element of evens\_2\_to\_20. The second time through the loop, ii has a value of 4 because that is the second element of evens\_2\_to\_20 .... and so on until all the elements of evens\_2\_to\_20 have been used.

In this example the only thing in the loop is a call to print to display the value of ii.

**b.** I just used the variable evens\_2\_to\_20 for illustrative purposes. Make the changes shown below and re-run the program to see how this bit of code can (and should) be streamlined.

for ii in range(2,22,2) :

print(ii)

Modify this code so that it counts by 5’s from 0 to 10,000. Note what you did.

**for ii in range(0,10000,5): # 10,000 exclusive**

**print(ii)**

**for ii in range(0,10001,5): # 10,000 inclusive**

**print(ii)**

Now, just because you can, make it count backwards by 6’s from 5000 to -200. Note what you did.

**for ii in range(5000,-200,-6):**

**print(ii)**

**5.** Try to predict what the following code will do. Then enter and run it.

squares = ()

for ii in range(1,11) :

squares = squares + (ii\*ii,)

print(squares)

**a.** What does the first line do and why is it needed?

**Creates the accumulator variable. It’s needed because you can’t add to a variable that doesn’t exist, and if you create it within the loop it will be reset each time the loop is run.**

**b.** Why do you think there are parentheses around ii\*ii in line 3? Remove them, run the code, and see what happens. What about the comma after ii\*ii?

**Removing the parentheses returns “TypeError: can only concatenate tuple (not "int") to tuple”**

**Removing the comma, or removing both, returns the same error.**

**6.** **Example - Calculating the mean** Enter, run and explore the following code

runs = (2, 3, 5, 6)

total = 0

for ii in range(0,len(runs)) :

total = total + runs[ii]

average = total/len(runs)

print(average)

**7.** **Example – Looping through a string** Enter, run and explore the following code.

text = "Here is some text. It is boring. Replace it with your own text."

special\_letter = 'e'

ocurrences = 0

for ii in range(len(text)) :

print(ii, text[ii])

if (text[ii] == special\_letter) :

ocurrences += 1

print("The letter '",special\_letter,"' ocurrs ",ocurrences," times in the text:", sep="")

print(text)

**8. The somewhat unique aspect of for-loops in Python.**

In the previous example note that the loop variable, ii, is an integer that is used as an index to access each element of the string text. Using the loop-counter as an index is the standard approach used in most languages to loop through arrays.

For-loops in Python are a little different because the loop variable takes on all the values of whatever sequence follows the keyword in.

**a.** **Example with a tuple.** Enter, run and explore the following.

crazy\_list = ('hi', 42, True, (1, 3, 5), 1/4)

for element in crazy\_list :

print(element)

**b.** **Example with a string.** Modify the code from exercise 7 in the following way, then run and explore it.

text = "Here is some text. It is boring. Replace it with your own text."

special\_letter = 'e';

ocurrences = 0;

for letter in text :

print(letter)

if (letter == special\_letter) :

ocurrences += 1;

print("The letter",special\_letter,"ocurrs",ocurrences,"times in the text:");

print(text);

**9.** **Example – Nested for-loops** Enter and run the following code and try to figure out how it works.

MAX = 10

for ii in range(1, MAX+1):

for jj in range(1, ii+1) :

print(ii,'\t',end = "");

print();

**This code creates MAX rows of code, each with a MAX number of columns, where each number on the same row is the row’s position, starting at 1.**

**10.** Modify your code from exercise 2 d in Lab 8, replacing the while-loop with a for-loop.

**Mariners\_runs = (10, 8, 8, 2, 3, 3, 5, 0, 6, 1)**

**Opponents\_runs = (3, 5, 2, 3, 1, 6, 3, 2, 4, 3)**

**Mariners\_record = 0**

**for i in range(10):**

**if Mariners\_runs[i] > Opponents\_runs[i]:**

**Mariners\_record += 1**

**print(Mariners\_record)**

**Quick Check**

Create a tuple that has the first 100 Fibonacci numbers.

**# Better demonstration of for loop example**

**nums = (1,)**

**new = 1**

**old = 0**

**for i in range(1, 100):**

**nums += (new + old,)**

**old = new**

**new = nums[i]**

**# Minimal coding example**

**nums = (1, 1)**

**for i in range(1, 99):**

**nums += (nums[i] + nums[i-1],)**