Lists & For Loops

Ceren Kocaoğullar
ckocaogullar15@ku.edu.tr

Date: 18th of February

1 Overview

This handout is prepared for KOLT Python Certificate Program. It contains a brief review of this week's topics and exercise questions.
You can download the starter code from here.
You can find the solutions at here after all the sections are conducted.

2 Review

2.1 Strings

Textual data in Python is handled with `str` objects, or strings. Strings are immutable sequences of Unicode code points.

2.1.1 Indexing

Strings are collections of characters. We can access specific characters using **indexing**. Let's look at an example:

```
my_name = 'hasan'
```

How can we access **third** character of this string?

```
  0  1  2  3  4
' h  a  s  a  n '
 -5 -4 -3 -2 -1
```

Notice that **indexing starts at zero** instead of one. Therefore, we need to use `my_name[2]` instead of `my_name[3]`. Alternatively, we can use negative indices.

2.1.2 Slicing & Other String Operations

- We can **slice** strings by using `[start:stop:step]` *(You don't need to specify step. In that case, it will be 1 by default)*
- `str1 + str2` ⇒ **Concatenate** `str1` and `str2`
- `str1 *` n ⇒ Repeate `str1` n times.

2.2 While loops

We can use `while` loops when we want to repeat some `<expression>`s **as long as** a `<condition>` is `True`.

```
while <condition>:
    <expression>
    <expression>
    ...
```

Code snippet 1: `while` statement, code block will be executed only if the condition is **true**. After each execution of code block, Python jumps back to condition check stage.

2.3 Lists

Lists contain **multiple pieces of information** at once and **in order**.

```
empty_list = []
list_with_objects = [1, 1, 2, 3, 5, 8, 13]
list_with_mixed_types = [1.0, 'sen', 1, 'ben', 1, 'de', 'bebek']
```

Code snippet 2: You can create lists empty or with elements. Lists in Python can contain different `types` of objects at once.

2.3.1 Accessing & Updating Elements

Similar to what we did with `Strings`, we use indexing to access elements of lists.

$$\begin{array}{rcccccl} & 0 & 1 & 2 & 3 & 4 & \\ \text{word} = [& \text{'ş'}, & \text{'a'}, & \text{'r'}, & \text{'z'}, & 0 &] \\ & -5 & -4 & -3 & -2 & -1 & \end{array}$$

To update 'z' in this list as 'j', we should do one of the following:

```
word[3] = 'j'
word[-2] = 'j'
```

Code snippet 3: Accessing and replacing elements is done via indexing.

2.3.2 List Mutation & Some Other List Operations

To add, remove, and further mutate elements, the functions listed below are used.

```
list.append(x) # Append x to end of the sequence list.insert(i, x):
     Insert x to index i
list.pop(i=-1) # Remove and return element at index i
list.remove(x): # Remove first occurrence of x
list.extend(iterable) # Add all elements in iterable to end of list
list[i] = new value # Update value of index i with new value
list[basic slice] = iterable # Change elements in basic slice with
    elements in iterable, sizes can be different:
numbers[:] = []
list[extended slice] = iterable # Change elements in extended slice
     with elements in iterable 1-1, sizes must be equal.
```

Code snippet 4: Common list mutation functions.

We may need to perform other operations on strings, the two most common of which are:

- Looking for a specific element (we use `in` operator for this)
- Getting the length of a list (we use `len` operator for this)

Here is a full list of the most useful list operators:

```
in operator # Check whether an element is in list.
3 in numbers # => True
len(list) # Returns the length of list (and other collections).
list.index(value, start=0, stop=len(list)) # Return first index of
    value.
list.count(value) # Count number of occurrences of value.
list.reverse() # Reverse the list (in-place)
list.sort() # Sort list elements (in-place)
```

Code snippet 5: Common list operations.

2.4 `for` Loops

`for` loops help us repeat blocks of code.

`range(start, stop, step)` is a function to create ranges. We can use this function to create `for` loops.

The general schema for `for` loops is:

```
for <item> in <iterable>:
    <expression>
    <expression>
    ...
```

Code snippet 6: `for` loops general structure

What do we mean by **iterable**? An **iterable**? is an object which one can iterate over. We will commonly use the following kinds of iterables in `for` loops.

```
# To print every character in string 'Python'
for ch in 'Python':
    print(ch)

# To print every element in a list multiplied by three
```

```
for num in [4,23,12,0,50]:
    print(num * 3)

# To print 'hello' 8 times
for i in range(0,8):
    print('hello')
```

Code snippet 7: `for` loop examples

2.4.1 Break, Continue & Pass

- **`break` immediately terminates the closest loop.** You can use it when you want to end a loop when a specific condition holds.
- **`continue` skips to the next iteration of the loop.** You can use it when you want your loop not to execute some code under a specific condition.
- **`pass` does not have an effect.** Loops, conditional statements, functions, classes, etc. cannot be empty. `pass` comes in handy when you need an empty one.

3 Exercises

3.1 Nerdy Santa

Kids wait all year to sit in Santa's lap and tell him what presents they want. Sadly, Santa's gifts are about to run out, so he decides to ask the child their age and give them present if their age is a prime number.

1. You should ask the child their age.
2. If it is a prime number and give them a gift. If not, give them the bad news.
3. Continue asking for a new age input until the user enters a value smaller than or equal to zero.

Definition of prime number: *A whole number greater than 1 that can not be made by multiplying other whole numbers.*

```
# Program starts
# Case 1: Age is a prime number
'Ho ho hooo! What is your age?': 13
'I have a gift for you!'

# Case 2: Age is not a prime number
'Ho ho hooo! What is your age?': 12
'Sorry, I do not have a gift for you.'

# Case 3: Age is smaller than or equal to zero
'Ho ho hooo! What is your age?': -1
'Sorry, age is invalid.'

# Program finishes
```

Code snippet 8: Example run

3.2 Mocking Mimic Machine

When we were kids, when we wanted to mock someone, we would funnily mimic their sentences. Since now we are mature enough to not do that ourselves, let's create a machine that will make it for us.

Our Mocking Mimic Machine should:

1. Take a number from the user.
2. Use that number to take that many words from the user.
3. Put those words inside a list.
4. For every item in the list, convert all vowels to letter **'o'**.
5. Print the converted versions of the words.

```
# Program starts
'How many words do you want me to process? ': 3
'What is word #1? ': Ceren
'What is word #2? ': Eralp
'What is word #3? ': Haluk
'Here is the output '
['Coron', 'Orolp', 'Holok']
# Program finishes
```

Code snippet 9: Example run

3.3 Printing Craze

Eralp was printing the lecture slides at the library when the printer suddenly failed, shuffled and omitted some pages. She tried to print again and encountered the same failure. She decided to fix the order of the pages and throw the duplicate pages into the recycle bin. Let's help Eralp with this arrangement.

1. We have a list of two print results with page numbers inside.
2. The final list should be ordered and should not contain duplicates.
3. The pages we will throw in the recycle bin should only be pages with duplicates.