Lecture 10

Loop Review and Strings as Sequences

10 Ventôse, Year CCXXX

Song of the day: ***Had To Come Back Wet*** *by Rogér Fakhr (1978).*

Part 0 (Review): *Must be funny in the rich man's world*

Let's say that you are tasked with writing the final money counter program for a Monopoly-like video game. If you don't know what **Monopoly** is, don't worry; all you need to know is that at the end of each game of Monopoly game, every player will have a certain amount of money, and the person with the most amount of money wins (kind of a **terrible concept**, if you ask me. But I digress).

So, your task is to write a program called that:

1. Asks the user for an integer value called `number_of_players`, which represents the number of players who played that round. The minimum number of players that can play a game of Monopoly is **2**, and the maximum number of players is (apparently) **8**. If the user enters any number under 2 or over 8, keep asking for that input until they enter a valid number of players.
2. Once they do so, each player will enter the values of each of their properties/assets.
3. Once a player enters all the values, the game will print out the sum.
4. The program repeats steps 2 and 3 until all players have been accounted for.
5. Print out at the very end which player had the most money at the end.

Here's a sample execution:

```
Enter a valid number of players: 3
Enter the value of a property/asset, or DONE to finish: 100
Enter the value of a property/asset, or DONE to finish: 34
Enter the value of a property/asset, or DONE to finish: 54
Enter the value of a property/asset, or DONE to finish: DONE
Player 1 has 188.0 dollars.
Player 1 is in the lead!
Enter the value of a property/asset, or DONE to finish: 10000
Enter the value of a property/asset, or DONE to finish: DONE
Player 2 has 10000.0 dollars.
Player 2 is in the lead!
Enter the value of a property/asset, or DONE to finish: 43.34
Enter the value of a property/asset, or DONE to finish: DONE
Player 3 has 43.34 dollars.
2 wins with 10000.0 dollars!
(venv) sebastianromerocruz@Sebastians-MBP 10 % python3 monopoly_counter.py
Enter a valid number of players: 1
Enter a valid number of players: 10
Enter a valid number of players: 3
Enter the value of a property/asset, or DONE to finish: 23.54
Enter the value of a property/asset, or DONE to finish: 34.667
Enter the value of a property/asset, or DONE to finish: 123.3
Enter the value of a property/asset, or DONE to finish: DONE
Player 1 has 181.51 dollars.
Player 1 is in the lead!
Enter the value of a property/asset, or DONE to finish: 1969.00
Enter the value of a property/asset, or DONE to finish: DONE
Player 2 has 1969.0 dollars.
Player 2 is in the lead!
Enter the value of a property/asset, or DONE to finish: 12.0
Enter the value of a property/asset, or DONE to finish: 0.05
Enter the value of a property/asset, or DONE to finish: DONE
Player 3 has 12.05 dollars.
2 wins with 1969.0 dollars!
```

Here, I have added a `print()` statement to let us know when a player takes the lead. This is optional.

A few of things to watch out for:

- For this problem, you can round dollar values appropriately as soon as you have determined that they are "roundable" values. You may round in the traditional mathematical way.
- You may assume that, for the value of a property/asset, the user will only enter either a positive numerical value or the string `"DONE"`.
- If you're unsure of how to keep track of the largest number in any process, the following example, where I determine the largest of 10 random numbers from 1 to 100, may help:

```python
import random

current_largest = -1  # we need a starting value that any amount can surpass

for iteration in range(10):
    random_number = random.randrange(1, 11)
    print("Number generate:", random_number)
    # If the random number is larger than the current largest number, it becomes th
    if random_number > current_largest:
        current_largest = random_number

print("The largest number was:", current_largest)
```

Possible output:

```
Number generate: 2
Number generate: 5
Number generate: 1
Number generate: 1
Number generate: 2
Number generate: 1
Number generate: 8
Number generate: 5
Number generate: 2
Number generate: 4
The largest number was: 8
```

Part 1: *Strings as Sequences*

Remember that a `for` -loop iterates through every member of a sequence?

```
for number in range(0, 10):
    print("Sequence member '", number, "'", sep="")
```

Output:

```
Sequence member '0'
Sequence member '1'
Sequence member '2'
Sequence member '3'
Sequence member '4'
Sequence member '5'
Sequence member '6'
Sequence member '7'
Sequence member '8'
Sequence member '9'
```

So, in this case, our sequence is every whole number between and including 0 and 9:

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

It turns out that, in Python, sequences aren't limited to being numerical—they can also be alpha-numerical:

```
alpha_seq = "Adachi and Shimamura"

for character in alpha_seq:
    print(character)
```

Output:

```
A
d
a
c
h
i

a
n
d

S
h
i
m
a
m
u
r
a
```

So is a string a sequence? Technically. It can basically be used in sequence-like ways, which makes it very handy for programmers, since a good amount of user-input that we receive is in string form. The ability to be able to consider characters one-by-one is will prove to be indispensable.

What are other cool, sequence-like things we can do with strings?

Well, what if we wanted to access any individual letter in a string? Let's say we have the following activity log in a chat-room:

```
[A]: hey
[A]: whats up
[B]: Please leave me alone
```

In this case, it's pretty clear that the `"A"` and `"B"` characters represent two members of a chat-room. Moreover, we know that both `"A"` and `"B"` are always the second character in the whole line. We can use this information to our advantage by using something called ***indexing***.

Let's say we wanted to find out whether user A or user B sent the message:

```
ID_LOCATION = 1

message = "[B]: Please leave me alone"
user_id = message[ID_LOCATION]

print("User", user_id, "sent this message.")
```

Output:

```
User B sent this message.
```

So, what did we do? The key line here is:

```
user_id = message[ID_LOCATION]
```

Since `ID_LOCATION` is equal `1` , we could read this line as:

> Take the **first** character from the string variable `message` and store it inside a variable called `user_id` .

Why is it the first character if `A` and `B` always appear after the `[` ? General speaking, all programming languages start counting from `0` instead of `1` . That's actually why the `range()` function's default starting value is `0` —it's an inherent characteristic of most programming languages. So if you wanted to refer to the `[` in the strings above, you'd say:

> In the string `"[B]: Please leave me alone"` , the **0th** (zeroth) element is `[` .

So with this knowledge, we can now print each of the characters of a string in two ways:

```
BOOK_TITLE = "In Search of Lost Time"
LENGTH = 22

# using sequences
for letter in BOOK_TITLE:
    print(letter)

# using indices
for index in range(LENGTH):
    letter = BOOK_TITLE[index]
    print(letter)
```

One quick other (***very***) important thing about strings is that they are ***immutable***. That is, once you define the value of a string, you **cannot** change it. The only way to do something similar is to create a whole new string using the value of the old string:

```
first_name = "Élisabeth Louise"
last_name = "Vigée Le Brun"
full_name = first_name + " " + last_name

print(full_name)
```

Output:

```
Élisabeth Louise Vigée Le Brun
```

Here, it is very important to recognize that the string `full_name` is *not* `first_name` with the strings `" "` and `last_name` appended to it. Instead, it is a **completely** different string, existing in a completely different place in memory. This new string will simply happen to have the contents of `first_name` , `" "` , and `last_name` put together in that order—and only because we asked Python to create it in such a way.

Mutability / immutability is a huge topic in this class and computer science in general, so don't forget the words. We'll see them again soon enough.

Part 2: *String Comparison*

If we have a variable called `string` , and it has a value of `"abc"` . What does the following expression evaluate to?

```
>>> string == "abc"
```

Well, let's check the output:

```
>>> string == "abc"
True
```

Makes sense; the comparison operator `==` checks whether two Python objects are equal in value. Since the value of `string` is `"abc"`, it is indeed equal to the string `"abc"`.

So, if we can check for the equality of strings, can we check for inequality?

```
>>> string != "not abc lol"
True
```

Makes sense; these strings are not at all the same in value. These are both pretty intuitive operations, but what about comparing them using `>`, `<`, `>=`, and `<=`?

```
>>> string >= "not abc lol"
False

>>> string < "bcd"
True
```

Clearly, these operations are not causing errors, so how do they work? When comparing string, Python uses their **lexicographical order** in order to determine whether one is larger than the other. Lexicographical order simply means applying a value of, say, 1 to `"a"`, 2 to `"b"`, 3 to `"c"`, etc. (the numerical equivalent of `"a"` is actually `97`, but don't worry about that for now).

This means that:

```
>>> "a" > "b"
False
>>> "a" < "b"
True
>>> "A" < "b"
True
```

Notice that these operations are not case-sensitive—a very rare exception in programming.

What about comparisons using strings of different lengths? Python basically goes in order:

```
>>> "Paris" > "Parthenon"
False
```

1. Is the `"P"` from `"Paris"` equal to the "`P`" from `"Parthenon"` ? Yes, so move on to the next character.
2. Is the `"a"` from `"Paris"` equal to the "`a`" from `"Parthenon"` ? Yes, so move on to the next character.
3. Is the `"r"` from `"Paris"` equal to the "`r`" from `"Parthenon"` ? Yes, so move on to the next character.
4. Is the `"i"` from `"Paris"` equal to the "`t`" from `"Parthenon"` ? No, so apply the `>` operation.
5. Is `"i"` `>` `"t"` ? `"i"` has a lower lexicoogical value than `"t"` (i.e. it appears earlier in the alphabet), so this operation evaluates to `False` .
6. The whole operation evaluates to `False`

What about this?

```
>>> "Car" >= "Cartagena"
False
```

1. Is the `"C"` from `"Car"` equal to the "`C`" from `"Cartagena"` ? Yes, so move on to the next character.
2. Is the `"a"` from `"Car"` equal to the "`a`" from `"Cartagena"` ? Yes, so move on to the next character.
3. Is the `"r"` from `"Car"` equal to the "`r`" from `"Cartagena"` ? Yes, so move on to the next character.
4. Since `"Car"` has no more values to compare with `"Cartagena"` , `"Cartagena"` is by default of larger value.
5. The operation, thus, evaluates to `False` .

Here are more examples:

```
>>> "Sonny Boy" < "SONNY BOY"
False

>>> "Nozomi" >= "Mizuho"
True

>>> "Napoleon I" > "Napoleon III"
```

```
False

>>> "!!!" <= "   "
False
```

That last one is a bit of a toughie if you don't know about ASCII values and how to find them, but don't worry—it's not important for now. We'll get there eventually.

Part 3: *The* `in` *Operator*

Yay, a new operator! `in` is actually kind of heaven-sent if you come from a Java/C++ background. `in` is what is called a **membership** operator, and it evaluates to either `True` or `False` :

```
>>> "Car" in "Cartagena"
True

>>> "PARIS" in "Parisian"
False
```

The above operations can be described in English as follows:

> > The string `Car` exists as a sub-string of the exact same value somewhere in the string `Cartagena` .
>
> > The string `PARIS` does not exist as a sub-string of the exact same value somewhere in the string `Parisian` .

That's why it's called a membership operator: it checks for the membership of an object inside another object. You can also check for non-membership:

```
>>> "Prague" not in "Czechia"
True

>>> "1" in "onetwothree"
False

>>> "net" not in "onetwothree"
False
```

Part 4: *Special Characters and* `print()` *'s* `end` *Parameter.*

There are a few "special characters" in programming that we should be aware of. The two that we'll need in this course are `"\n"` and `"\t"` .

`\n` is the **newline operator**:

```
information = "NAME: Alice Sara Ott\nOCCUPATION: Pianist\nBIRTHPLACE: München, West
print(information)
```

Output:

```
NAME: Alice Sara Ott
OCCUPATION: Pianist
BIRTHPLACE: Munich, West Germany
```

And `\t` is the **tab, or indentation, operator**:

```
first_half = "Jan\tFeb\tMar\tApr\tMay\tJun\t"
second_half = "Jul\tAug\tSep\tOct\tNov\tDec\t"

print(first_half)
print(second_half)
```

Output:

```
Jan     Feb     Mar     Apr     May     Jun
Jul     Aug     Sep     Oct     Nov     Dec
```