Weekend: Looping

Looping

A loop is a way to execute some piece of code over and over again. There are 2 kinds of loops in Python;

1. `for` loop
2. `while` loop

`for` Loops

A `for` loop is used when one wants to repeat something a number of times. Just like the `if` statements, blocks of code in a for loop are indented, otherwise they will not run.

Here is an example of a `for` loop in Python:

input_example.py

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

for number in numbers:
    print(number)
```

The `for` statement loops through the `numbers` list, and stores each individual number in a variable called `number`, then prints out this new variable we have created.

```
$ python3.6 input_demo.py
1
2
3
4
5
```

We can also use a Python function named `range()` in our `for` loops. The `range()`function returns a list for which we can use to loop through.

First let us see how we use the `range()` function by creating a list.

input_demo.py

```
list_a = list(range(0,5))
print(list_a)
```

Here's what this code does:

```
$ python3.6 input_demo.py
[0, 1, 2, 3, 4]
```

The `range()` function can take on two parameters. The first one is the number from where to start the range, and the second one is where to stop the range - this does **not** include that number. In our example here, we start the range at 0 and end it at 5. Then we use the `list()` method to convert it to a list and print it out.

Let's take a look at another example using `range()`.

input_demo.py

```
for i in range(0,7):
    print("I would love " + str(i) + " cookies")
```

Here we created a loop that prints out the string `I would love cookies` with the number of cookies incrementing with the `for` loop. We used the `str()` method to convert the number `i` to a string so that we could concatenate it with the rest of the string. Here's the output:

```
$ python3.6 input_demo.py
I would love 0 cookies
I would love 1 cookies
I would love 2 cookies
I would love 3 cookies
I would love 4 cookies
I would love 5 cookies
I would love 6 cookies
```

Let us create another example of a `for` loop with conditions.

input_demo.py

```
numbers = [1, 2, 3, 4, 5]
for i in numbers:
    if i % 2 == 0:
        print(i)
```

Here we loop through the numbers list and check if the number inside the list is fully divisible by 2; if it is, then we print out that number.

Here's the output of this `for` loop:

```
$ python3.6 input_demo.py
2
4
```

`while` Loops

Unlike `for` loops, `while` loops run until a certain condition is met. `while` loops are often used as counters. Let's see one in action:

input_demo.py

```
players = 11

while players >= 5 :
    print("The remaining players are",players)
    players -= 1
```

We've created a variable `players` that we set as 11, and then created a `while` loop with the condition that the value of `players` needs to be greater than or equal to 5. If this condition is true, then a string will print and the value of `players` will decrease by 1, and the loop will execute again until the condition is no longer true (that is to say, `players` is less than 5).

Here's what happens when we run it:

```
$ python3.6 input_demo.py
The remaining players are 11
The remaining players are 10
The remaining players are 9
The remaining players are 8
The remaining players are 7
The remaining players are 6
The remaining players are 5
```

Sure enough, the code only executes when `players` is greater than or equal to 5. After that, the `while` loop is exited and the code stops running.

Control Flow Exercises

Magic 8 ball

Create the popular Magic 8 Ball game. If you've never played it before, it is a toy that you can ask a yes-or-no question and it will give you a random answer (feel free to read more on the **Wikipedia page (https://en.wikipedia.org/wiki/Magic_8-Ball)**). In your program, allow a person to input a question, and then create a random number. Based on the random number, pick a response to the question using `if`, `elif`, and `else` statements. You can find the responses at the **Wikipedia page (https://en.wikipedia.org/wiki/Magic_8-Ball#Possible_answers)** .

`break Statement`

The `break` statement allows code to jump out of a loop whenever a certain condition has been met.

input_demo.py

```
number = 0
while True:
    print("I love candy "+ str(number))
    number +=1
    if number == 7 :
        break
```

When we run this code, see what prints to the console:

```
$ python3.6 input_demo.py
I love candy 0
I love candy 1
I love candy 2
I love candy 3
I love candy 4
I love candy 5
I love candy 6
```

Each time the loop runs, and prints out our statement, and checks if `number` is equal to 7. If not, it runs again. If it is, the loop stops.

`continue Statement`

The `continue` statement is also used in loops. It allows us to jump back up to the top of our loop. The `continue` keyword ignores all other statements under it and they are not run. Let's look at an example using `continue`.

input_demo.py

```
'''
in a team of members 20, some numbers are taken
and want to display the numbers that are not taken
so others don't pick the picked numbers
'''

# taken numbers
numTaken = [3,5,7,11,13]

print("Available numbers")

# loop
for i in range(1,21):
    if i in numTaken:
        continue
    print(i)
```

We want to print numbers from 1-20, but skip over the numbers in the `numTaken` list. We've written a standard `for` loop, with a twist. If the current number `i` is in the `numTaken` list, the `print()` statement directly under the `continue` is not executed. This is because every time `continue` is called, Python jumps back up to the top of the loop.

```
$ python3.6 input_demo.py
Available numbers
1
2
4
6
8
9
10
12
14
15
16
17
18
19
20
```

The numbers 1-20 have printed, but we've skipped over the numbers in our list.

Lists

Lists are another Python data type. They are the equivalent of arrays in other languages. A Python list can store any kind of value.

We can create a list using square brackets `[]` or using the `list()` function, like this. We can open our Python shell and practice the following commands.

```
my_list = []
my_other_list =list()
```

Now let's create some lists that actually hold data.

```
list_a = ["a","b","c","d"] # list of strings
list_b = [1,2,3,4,5,6] # list of numbers
list_c = [1,"west",34,"longitude"] # mixed list
```

As we can see with the code above, we can create a list that holds similar types of items, such as the first two which hold only strings or only numbers; or we can mix the types of data stored in our list, as we do with `list_c`, which holds both numbers and strings.

We can even nest our lists with other lists.

```
list_d = [ ["a","b","c","d"],[1,2,3,4,5,6],[1,"west",34,"longitude"]] # nested list
```

We can join two lists using the `extend()` method.

```
list_a = ["a","b","c","d"]
list_b = [1,2,3,4,5,6]

# Joining list_b to list_a

list_a.extend(list_b)

print(list_a) # this will print ["a","b","c","d",1,2,3,4,5,6]
print(list_b) # this will print [1,2,3,4,5,6]
```

Here we have joined `list_b` to `list_a`. This adds new values to `list_a`. Notice that `list_b` remains unchanged.

We can also append values to a list using the `append()` method.

```
list_a = ["a","b","c","d"]

print(list_a)  # ["a","b","c","d"]

list_a.append("e")

print(list_a) # ["a","b","c","d","e"]
```

Here we append a single value, `"e"`, to the list.

We can also arrange list items in place using the `reverse()` and `sort()` methods.

```
list_a = ["a","b","c","d"]
list_a.reverse()

print(list_a) # ['d', 'c', 'b', 'a']
```

Here we use the `reverse()` method to reverse the list.

```
list_a = [1,3,4,8,5,7,6,2]
list_a.sort()

print(list_a) # [1, 2, 3, 4, 5, 6, 7, 8]
```

The `sort()` method arranges items in order.

Note `reverse` and `sort` methods only work on the list they were called on. They have no return value.

```
list_a = [1,2,3,4,5]
list_a.reverse() # converts list_a to [5,4,3,2,1]
list_b = list_a.reverse()  # list_b is None
```

Tuples

Tuples are like lists. The main difference between tuples and lists is that tuples are **immutable** - their values cannot be changed once created.

```
tuple_a = ("a","b","c","d") # tuple of strings
tuple_b = (1,2,3,4,5,6) # tuple of numbers
tuple_c = (1,"west",34,"longitude") # mixed tuple
tuple_d = tuple() #empty tuple
```

We can create a tuple using parentheses `()` (as in `tuple_a`, `tuple_b`, and `tuple_c` in the above example) or we can use the `tuple()` function (as in `tuple_d`).

Dictionary

Another really powerful data structure in Python is the dictionary. Like lists, dictionaries store collections of many values of different types. Unlike indexes for lists - which are integers and start from 0, dictionary indexes don't have to be integers. They can be of different data types like strings. Indexes in dictionaries are called **keys**. The value here is appropriately called the **value**.

Dictionaries can be created in two ways:

```
# Creating empty dictionaries
my_dict = {}
my_dict = dict()

# Creating a dictionary with keys and values
my_cat = {'name':'Mr sniffles','age':18, 'color':'black'}
```

In the example above, we created a new dictionary that stores details about `my_cat`. We can fetch data from a dictionary by using our key as the index. Continuing from the above example, let's say that we want to print the name of `my_cat`, `'Mr sniffles'`. Here is how we would access that value from the dictionary:

```
cat_name = my_cat['name']
print(cat_name) # 'Mr sniffles'
```

Unlike lists, dictionaries are unordered since the indexing does not start at [0]. We will see what that means later on.

We can also add items to a dictionary by defining a new key and assigning it a value.

```
birthdays = {"John":"August 1","Marcus":"April 8"}
birthdays["mary"] = "September 14"
print(birthdays) # this prints {"John":"August 1","Marcus":"April 8","Mary":"September 14"}
```

Here we had a dictionary with the birthday of two people, `John` and `Marcus`. Then we added `Mary` to our birthday dictionary.

We can also get the individual keys in the dictionary using the `keys()` method.

```
birthday = {"John":"August 1","Marcus":"April 8","Mary":"September 14"}

print(birthday.keys()) # this prints dict_keys(['John', 'Marcus', 'Mary'])
```

The `keys()` method creates a `dict_keys` object that holds a list of all the keys.

We can convert them into a list using the `list()` function.

```
birthday = {"John":"August 1","Marcus":"April 8","Mary":"September 14"}

print(list(birthday.keys())) # this prints ['John', 'Marcus', 'Mary']the ne
```

`setdefault()` method

When dealing with a dictionary, sometimes it can be useful to set placeholder values to keys, before we do any operation on them, to prevent unnecessary errors.

For example, let's say we want to create an application that counts every character in an input string. Open up *import_demo.py* and add this code:

import_demo.py

```
print("Enter a string")

input_string = input()

characters = {}
```

We prompt the user for an input. Then we create an empty dictionary named `characters` that will store each individual character and how many times it has been used. Now let's use `setdefault()`:

import_demo.py

```
print("Enter a string")

input_string = input()

characters = {}

for character in input_string:
    characters.setdefault(character,0)
```

Here we use a `for` loop to iterate through each individual character in the string. Then we use the `setdefault()` method to create a key for each character if the character does **not** exist in the dictionary. Then we set the default value for each as `0`.

import_demo.py

```
print("Enter a string")
```

```
input_string = input()

characters = {}

for character in input_string:
characters.setdefault(character,0)
characters[character] = characters[character] + 1

print(characters)
```

Lastly we target the key that matches the character and add one to the value everytime the character is encountered.

When we run this, here's what happens:

```
$ python3.6 import_demo.py
Enter a string # I'm going to enter "James is awesome"
James is awesome

{' ': 3, 'J': 1, 'a': 2, 'e': 3, 'i': 1, 'm': 2, 'o': 2, 's': 4, 'w': 1}
```

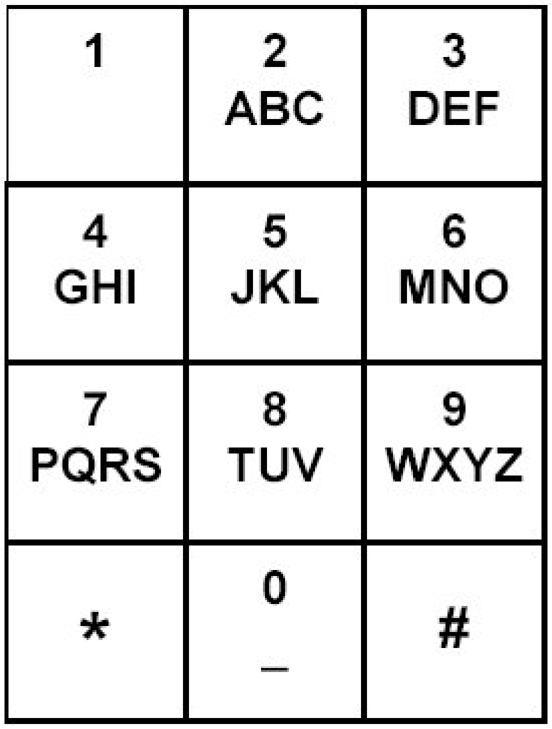
List and Dictionary Exercises

Magic 8 ball

Recreate the Magic 8 Ball exercise from the Control Flow lesson using lists instead of `if` statements.

Feature Phones

Remember the days before smartphones when text was entered in our phones using the keypad? If you need a refresher, see this image of a phone keypad:



Notice the letters on each number. To type a letter, the user needs to press the button on which the letter is printed the number of times that corresponds to its order on the button.

For example, to type "E", the user needs to press "3" two times. To type "Z", the user needs to press "9" four times. To type "DOG", they need to press "3" one time, "6" three times, and "4" one time. And so on.

Create a program that allows a user to enter text and then it calculates how many clicks it takes for a user to input the text. Ignore uppercase and special characters.

For example, if a user inputs ***"James"***, the output should be ***9*** because...

- "J": press "5" one time
- "A": press "2" one time
- "M": press "6" one time
- "E": press "3" two times
- "S": press "7" four times

Put this all together, and 1 + 1 + 1 + 2 + 4 = 9.