C2M5L4_Errors_and_Exceptions-V2

July 9, 2020

1 Practice Notebook - Errors and Exceptions

Below we have a function that removes an item from an input list. Run it to see what it does.

```
In [1]: my_list = [27, 5, 9, 6, 8]

        def RemoveValue(myVal):
            my_list.remove(myVal)
            return my_list

        print(RemoveValue(27))
```

```
[5, 9, 6, 8]
```

We used the `RemoveValue()` function to remove the number, 27 from the given list. Great! The function seems to be working fine. However, there is a problem when we try to call the function on the number 27 again. Run the following cell to see what happens.

```
In [2]: print(RemoveValue(27))
```

```
        ---------------------------------------------------------------------------

        ValueError                                Traceback (most recent call last)

        <ipython-input-2-3f8c37f416f6> in <module>
    ----> 1 print(RemoveValue(27))


        <ipython-input-1-597531fc6dcc> in RemoveValue(myVal)
          2
          3 def RemoveValue(myVal):
    ----> 4     my_list.remove(myVal)
          5     return my_list
          6
```

```
ValueError: list.remove(x): x not in list
```

From the above output we see that our function now raises a **ValueError**. This is because we are trying to remove a number from a list that is not in the list. When we removed 27 from the list the first time, it was no longer available in the list to be removed a second time. Python is letting us know that the number 27 no longer makes sense for our `RemoveValue()` function. We'd like to take control of the error messaging here and pre-empt this error. Fill in the blanks below to raise a ValueError in the `RemoveValue()` function if a value is not in the list. You can have the error message say something obvious like "Value must be in the given list".

```
In [3]: def RemoveValue(myVal):
            if myVal not in my_list:
                raise ValueError("Value not present in the list")
            else:
                my_list.remove(myVal)
            return my_list

        print(RemoveValue(27))
```

```
        ---------------------------------------------------------------------------

        ValueError                                Traceback (most recent call last)

        <ipython-input-3-7241e21365c9> in <module>
          6     return my_list
          7
    ----> 8 print(RemoveValue(27))


        <ipython-input-3-7241e21365c9> in RemoveValue(myVal)
          1 def RemoveValue(myVal):
          2     if myVal not in my_list:
    ----> 3         raise ValueError("Value not present in the list")
          4     else:
          5         my_list.remove(myVal)


        ValueError: Value not present in the list
```

Did your error message print correctly? Was the output something like: **ValueError: Value must be in the given list**? If not, go back to the previous cell and make sure you filled in the blanks correctly. If your error message did print correctly, great! You are on your way to mastering the basics of handling errors and exceptions. Now, let's look at a different function. Below we have a function that sorts an input list alphabetically. Run it to see what it does.

```
In [4]: my_word_list = ['east', 'after', 'up', 'over', 'inside']

        def OrganizeList(myList):
            myList.sort()
            return myList

        print(OrganizeList(my_word_list))
```

```
['after', 'east', 'inside', 'over', 'up']
```

We used the `OrganizeList()` function to sort a given list alphabetically. The function seems to be working fine. However, there is a problem when we try to call the function on a list containing number values. Run the following cell to see what happens.

```
In [5]: my_new_list = [6, 3, 8, "12", 42]
        print(OrganizeList(my_new_list))
```

```
        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-5-459d95f39adf> in <module>
          1 my_new_list = [6, 3, 8, "12", 42]
    ----> 2 print(OrganizeList(my_new_list))


        <ipython-input-4-252da45e411d> in OrganizeList(myList)
          2
          3 def OrganizeList(myList):
    ----> 4     myList.sort()
          5     return myList
          6


        TypeError: '<' not supported between instances of 'str' and 'int'
```

From the above output we see that our function now raises a **TypeError**. This is because the `OrganizeList()` function makes sense for lists that are filled with only strings. Take control of the error messaging here and pre-empt this error by filling in the blanks below to add an assert type argument that verifies whether the input list is filled with only strings. You can have the error message say something like "Word list must be a list of strings".

```
In [8]: def OrganizeList(myList):
            for item in myList:
                assert type(myList) == str, "Word list must be a list of strings"
            myList.sort()
```

```
        return myList

    print(OrganizeList(my_new_list))
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)

<ipython-input-8-f2396f128b6c> in <module>
  5     return myList
  6
----> 7 print(OrganizeList(my_new_list))


<ipython-input-8-f2396f128b6c> in OrganizeList(myList)
  1 def OrganizeList(myList):
  2     for item in myList:
----> 3         assert type(myList) == str, "Word list must be a list of strings"
  4     myList.sort()
  5     return myList


AssertionError: Word list must be a list of strings
```

Did your error message print correctly? Was the output something like: **AssertionError: Word list must be a list of strings**? If not, go back to the previous cell and make sure you filled in the blanks correctly. If your error message did print correctly, excellent! You are another step closer to mastering the basics of handling errors and exceptions. Let's look at one last code block. The `Guess()` function below takes a list of participants, assigns each a random number from 1 to 9, and stores this information in a dictionary with the participant name as the key. It then returns *True* if Larry was assigned the number 9 and *False* if this was not the case. Run it to see what it does.

In [11]:
```
import random

participants = ['Jack','Jill','Larry','Tom']

def Guess(participants):
    my_participant_dict = {}
    for participant in participants:
        my_participant_dict[participant] = random.randint(1, 9)
    if my_participant_dict['Larry'] == 9:
        return True
    else:
        return False

print(Guess(participants))
```

```
False
```

The code seems to be working fine. However, there are some things that could go wrong, so find the part that might throw an exception and wrap it in a try-except block to ensure that you get sensible behavior. Do this in the cell below. Code your function to return *None* if an exception occurs.

```
In [15]: # Revised Guess() function
         def Guess(participants):
             my_participant_dict = {}
             for participant in participants:
                 my_participant_dict[participant] = random.randint(1, 9)
             try:
                 if my_participant_dict['Larry'] == 9:
                     return True
                 else:
                     return False
             except KeyError:
                 return None
```

Call your revised `Guess()` function with the following participant list.

```
In [16]: participants = ['Cathy','Fred','Jack','Tom']
         print(Guess(participants))
```

```
None
```

Was the above output *None*? If not, go back to the code block containing your revised `Guess()` function and make edits so that the output is *None* for the previous code block. If the above output was indeed *None*, congratulations! You've mastered the basics of handling errors and exceptions in Python and you are all done with this notebook!