Topic-6 Lists

In previous topics, we have discussed basics of lists:

- How to define a list
- How to access the elements in a list
- How to update the elements
- `range()` function gives us a range of numbers
- `len()` function to obtain length In this part, we will study more about list, which is a useful and powerful container in Python .

1. A list is a sequence

Define a list

- Like a string, a **list** is a sequence of values. In a string, the values are characters; in a list, the values can be any type. For example,

```
>>> list_1 = [10, 20, 30, 40]
>>> list_2 = ['Rutgers', 'Princeton', 'NYU']
```

- In `list_1`, all values are number, or specifically `int`. In `list_2` all values are strings.
- However, the elements in a list **are not necessarily the same type**. For example,

```
>>> list_3 = [1, 3.14, 'Rutgers']
```

- In the above list, we have `int`, `float`, and `string`
- We can even put a list into another list.

```
>>> list_4 = [[1, 3.14], 'Rutgers']
```

- In the above list, the first element is a list, which is `[1, 3.14]`; the second list is a string.
- So you can image that we can use a list to represent a matrix.

```
>>> m = [[1,2,3],[4,5,6]]
```

`len()` function

- Just like strings, `len()` also works for lists.

```
>>> list_1 = [10, 20, 30, 40]
>>> len(list_1)
4
```

List slices

- List strings, the indices in list starts at 0.

```
>>> l = ['a', 'b', 'c', 'd', 'e', 'f']
>>> l[1:3]
['b', 'c']
```

- `l[1:3]` returns the elements with indices 1 and 2. It means it starts from 1 and stops at the index smaller than 3.
- If you omit the first index, the slice starts at the beginning.

```
>>> l = ['a', 'b', 'c', 'd', 'e', 'f']
>>> l[:4]
['a', 'b', 'c', 'd']
```

- If you omit the second, the slice goes to the end.

```
>>> l = ['a', 'b', 'c', 'd', 'e', 'f']
>>> l[3:]
['d', 'e', 'f']
```

- If you omit both, the slice is a copy of the whole list.

```
>>> l = ['a', 'b', 'c', 'd', 'e', 'f']
>>> l[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

2. Lists are mutable

- Different from strings, lists are mutable.

```
>>> l = ['a', 'b', 'c', 'd', 'e', 'f']
>>> l[0] = 'w'
>>> l
['w', 'b', 'c', 'd', 'e', 'f']
```

- We can also use slices to change the elements in a list.

```
>>> l = ['a', 'b', 'c', 'd', 'e', 'f']
>>> l[1:3] = ['x', 'y']
>>> l
['a', 'x', 'y', 'd', 'e', 'f']
```

3. List methods

`.append()`

- `append()` method adds a new element to the end of a list.

```
>>> l = ['a', 'b', 'c']
>>> l.append('d')
>>> l
['a', 'b', 'c', 'd']
```

`.extend()`

- `.extend()` takes a list as an argument and appends all of the elements:

```
>>> l1 = ['a', 'b', 'c']
>>> l2 = ['d', 'e']
>>> l1.extend(l2)
>>> l1
['a', 'b', 'c', 'd', 'e']
```

`.sort()`

- `.sort()` sorts the list ascending by default.
- There is an optional parameter, `reverse`, in `sort` method. Default is `reverse=False`. `reverse=True` will sort the list descending.
- For strings:

```
>>> l = ['d', 'b', 'a', 'c']
>>> l.sort()
>>> l
['a', 'b', 'c', 'd']
```

- For numbers:

```
>>> l = [4, 10, 5, 2]
>>> l.sort()
>>> l
[2, 4, 5, 10]
```

- or

```
>>> l = [4, 10, 5, 2]
>>> l.sort(reverse=True)
>>> l
[10, 5, 4, 2]
```

4. Void and fruitful method/function

- Let us review methods in strings. Take `.upper()` for an example,

```
name = 'jin'
name = name.upper()
print(name)
```

- We will get the expected result, `JIN`.

- Let us compare the string methods with the list methods below. Take the string `.append()` for an example,

```
l = ['a', 'b', 'c']
l.append('d')
print(l)
```

- Can you see the difference?
- **In string**, the method `.upper()` return a string, which is capitalized letters.
- We call a function/method that returns values **fruitful function/method**, called a function that doesn't return values **void function/method**.
- **In list**, most methods are *void*; they modify the list and return None.

5. Traversing a list

- The most common way to traverse the elements of a list is with a for loop. The syntax is the same as for strings:

```
universities = ['Rutgers', 'NYU', 'Princeton']
for i in universities:
    print(i)
```

- We can also traverse a list through indices.

```
universities = ['Rutgers', 'NYU', 'Princeton']
for i in range(len(universities)):
    print(names[i])
```

6. List operations

+ operator

- The + operator concatenates lists:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> c
[1, 2, 3, 4, 5, 6]
```

* operator

- The * operator repeats a list a given number of times:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

`in` operator

- The `in` operator helps find out whether an element is in a list.

```
>>> universities = ['Rutgers', 'NYU', 'Princeton']
>>> 'Rutgers' in universities
True
>>> 'Harvard' in universities
False
```

7. Deleting elements

Method 1: `.pop()`

- If you know the index of the element you want, you can use pop:

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> t
['a', 'c']
>>> x
'b'
```

- `pop` modifies the list and returns the element that was removed.

Method 2: `del`

- If you don't need the removed value, you can use the `del` operator:

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> t
['a', 'c']
```

- To remove more than one element, you can use `del` with a slice index:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
```

```
>>> t
['a', 'f']
```

Method 3: `.remove()`

- If you know the element you want to remove (but not the index), you can use remove:

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> t
['a', 'c']
```

- The return value from remove is None.

8. List operation 1: Map

Map

- Sometimes you want to traverse one list while building another. For example, the following function takes a list of numbers and returns a list of square values.

```
def square_values(x):
    squared = []
    for n in x:
        squared.append(n ** 2)
    return squared

numbers = [1, 2, 3, 4, 5]
print(square_values(numbers))
```

- An operation like *square* is sometimes called a **map** because it "maps" a function (in this case the function square) onto each of the elements in a sequence.

- We can achieve the same result without using an explicit loop by using `map()`.

```
def square(x)
    return number ** 2

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

squared = map(square, numbers)

print(list(squared))
```

- The call to `map()` applies `square()` to all the values in `numbers` and returns an iterator that yields square values. Then we call `list()` on `map()` to create a list object containing the square values.

- Since `map()` is written in C and is highly optimized, its internal implied loop can be more efficient than a regular Python `for` loop. This is one advantage of using `map()`.
- Format of `map()`

```
map(function, iterable)
```

Lambda function

- Functions can (1) perform action(s), or/and (2) return output(s)
- Lambda function is mainly used to take inputs and return outputs.

```
def divide(x, y):
    return x/y

# lambda function, which is equivalent to the above divide function.
divide = lambda x, y: x/y

print(divide(1,3))
```

- Which is interesting in lambda functions is that *if we don't assign lambda function to a variable, Python will destroy it immediately.*

Lamdba function in `map()`

- Lambda function is often used in when a function is used only once.
- For example, in the above map program, we can use lambda function instead of defining an explicit function.

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

squared = map(lambda x: x**2, numbers)

print(list(squared))
```

9. List operation 2: Filter

- Let us create a program, which can select positive numbers from a given list.

```
def select_positive(x):
    positive_numbers = []
    for i in x:
        if i>0:
            positive_numbers.append(i)
    return positive_numbers
```

```
numbers = [0, -1, 2, -3, 4]
print(select_positive(numbers))
```

- An operation like `select_positive` above is called a **filter** because it selects some of the elements and filters out the others.
- We can achieve the same result without using an explicit loop by using `filter()`.

```
def is_positive(n):
    return n > 0

numbers = [0, -1, 2, -3, 4]
positive_values = filter(is_positive, numbers)

print(list(positive_values))
```

- You can see the syntax is the same as `map()`. It just `filter(function, iterable)`, in which return value in the function should be boolean (True or False)
- We may use lambda function in `filter()`.

```
numbers = [0, -1, 2, -3, 4]
positive_values = filter(lambda n: n>0, numbers)

print(list(positive_values))
```

10. List comprehension

- Comprehensions help create new lists simply.
- For example, we would like to create a new list containing the values that are two times values in another list.

```
numbers = [1, 3, 5]
double_numbers = []

for n in numbers:
    double_numbers.append(n * 2)

print(double_numbers)
```

- Comprehension has similar syntax above, but much shorter and easier to understand.

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

double_numbers = [n * 2 for n in numbers]
```

```
print(double_numbers)
```

Comprehensions with conditionals

- We can also use conditionals in comprehensions.
- For example, if we would like to print out the positive numbers in a list. We can use comprehensions with conditionals.

```
numbers = [1, -2, 3, -4]
positive_numbers = [n for n in numbers if n > 0]
print(positive_numbers)
```

11. Practice questions

1. Find even numbers, using loop and `append()` method.
    - Method 1:
        - Define a list containing a list of numbers
        - Use `for` loop to traverse the elements in the list. If the element is an even number, put it into a new list using `append` method.
        - For example, given a list `[2, 3, 10, 17, 20]`, the result is [2, 10, 20].
    - Method 2:
        - Use comprehension to finish the question.
2. Find prime numbers
    - Create a function which takes an integer, return `True` if the number is a prime number and return `False` otherwise.
    - Define a list containing a list of numbers
    - Use `filter` function to obtain the prime numbers in the list and print them.
        - For example, given a list `[2, 3, 10, 17, 20]`, the prime numbers are 2, 3 and 17.
3. Capitalize all words in a list, using `map` function.
    - Create a function which takes a string and returns capitalized string.
    - Define a list containing several words/strings.
    - Use `map` function to capitalize all words in the list.
    - For example, given a list `['Rbs', 'Rutgers']`, the result is ['RBS', 'RUTGERS'].
    - Use comprehension to finish the question.
4. Write a function called `nested_sum` that takes a list of lists of integers and adds up the elements from all of the nested lists. For example:

```
>>> t = [[1, 2], [3], [4, 5, 6]]
>>> print(nested_sum(t))
21
```

```
d = {
    'Jack': ["Mary", "Joan", "Hellen"],
    'Mary': ["Jack", "Joan", "Peter"],
    'Jin':['Mary', 'Joan']
}
```