3. Strings and Lists

1. A string is a sequence

- We mentioned strings in Topic 1. We discussed string format, concatenation, repetition.
- In previous topics, we have mentioned several data types, i.e., integer, float, boolean, and strings. However, string is not like integer, float, and boolean. It is an ordered collection of other values.
- A string is a sequence of characters. You can access the characters one at a time with the bracket operator.

```
>>> s = "Hello"
>>> letter = s[1]
```

```
>>> letter
```

- The second statement selects character number 1 from `s` and assigns it to `letter`.
- The expression in brackets is called an *index*. The index indicates which character in the sequence you want.
- But the result might not be what you expect. It is `e`, rather than `H`. It is because index starts from 0.

len function

- Like list, we can use `len()` function of get the length of a string, i.e., the total number of characters in a string.

```
>>> s = "Hello"
>>> len(s)
5
```

- Note that the indices of a string are $0, 1, 2, \cdots, n-1$, whether n is the length of the string. Hence, to get the last letter of a string, you will get an error if you try something like this:

```
>>> s = "Hello"
>>> s[len(s)]
IndexError: string index out of range
```

- The reason for the `IndexError` is that there is no letter in `Hello` with the index 5. The five letters in the string are indexed 0 to 4. So To get the last character, you have to subtract 1 from length:

```
>>> s = "Hello"
>>> s[len(s)-1]
'o'
```

String slices

- A segment of a string is called a *slice*. Selecting a slice is similar to selecting a character, which can be done by using indices.

```
>>> s = "Hello, Jin!"
>>> s[0:5]
```

- Guess which characters would be printed, the first five or six characters? That is, `0:5` includes indices, 0, 1, 2, 3, 4 or 0, 1, 2, 3, 4, 5? The answer is 0, 1, 2, 3, 4, and there is no 5. Hence, the output is "Hello", the first five characters.

- You may have noticed that `n:m` includes the indices staring from `n` and stopping at `m-1`. For example, `2:5` includes 2, 3, and 4. For example:

```
>>> s = "Hello, Jin!"
>>> s[2:5]
'llo'
```

- As an exercise, try to get `Jin`.
- In addition to the formula, `n:m`, you can also use `:m` and `n:`. If you omit the index before the colon, the indices start at 0. If you omit the index after the colon, the indices stop at the end. For example

```
>>> s = "Hello, Jin!"
>>> s[:5]
'Hello'
>>> s[7:]
'Jin!'
```

- If the first index is greater than or equal to the second, the result is an empty string, represented by two quotation marks:

```
>>> s = "Hello, Jin!"
>>> s[3:3]
''
>>> s[3:2]
''
```

2. Strings are immutable

- It is tempting to use the [] operator on the left side of an assignment, with the intention of changing a character in a string.

```
>>> s = "Hello, Jin!"
>>> s[8] = 'o'
TypeError: 'str' object does not support item assignment
```

- An error occurs here saying 'str' object does not support item assignment. The “object” in this case is the string and the “item” is the character you tried to assign.
- The reason for the error is that strings are immutable, which means you can’t change an existing string.
- But you can get what you want in another way, creating a new string that is a variation on the original.

```

```

```
>>> s = "Hello, Jin!"
>>> new_s = s[:8] + 'o' + s[9:]
```

3. String methods

.strip()

- `input` function can be used for a user to input a string. But we cannot expect users will cooperate as intended. Hence, we need to either correct or check the input. For example:

```
# ask user to input name
name = input("Please input your name: ")

# print output
print(f"Hello, {name}!")
```

- It is common that users may mistakenly put extra spaces before or after their input. For example, run the program, then input `Jin ` (there are 3 white spaces before and after 'Jin'). The output would be `Hello, Jin !`
- It is fine, but awkward. We don't expect the white spaces.
- A string built-in method `strip()` can strip all whitespaces on the left and right of a string.
- We can modify the program to be:

```
# ask user to input name
name = input("Please input your name: ")

# Remove white spaces from the string
name = name.strip()

# print output
print(f"Hello, {name}!")
```

- Now if we input `Jin ` , the output would be `Hello, Jin!` as expected.

.title()

- Again, in the name input example, users might just put small letters, for example, "jin", rather than "Jin" with capital letter at the beginning.
- We can use `.title()` method to capitalize the first letter of each word.

```
# ask user to input name
name = input("Please input your name: ")

# Remove white spaces from the string
name = name.strip()
```

```
# Capitalize the first letter of each word
name = name.title()

# print output
print(f"Hello, {name}!")
```

- If we input `jin wang`, or even `jIN wAng`, the output would be `Hello, Jin Wang!`
- You may be tired of adding methods. You can put the methods together to make the coding more efficient. You can get the same output as the previous code.

```
# ask user to input name
name = input("Please input your name: ")

# Remove white spaces from the string and Capitalize the first letter
of each word
name = name.strip().title()

# print output
print(f"Hello, {name}!")
```

- Or even go further!

```
# ask user to input name, Remove white spaces from the string, and
Capitalize the first letter of each word
name = input("Please input your name: ").strip().title()

# print output
print(f"Hello, {name}!")
```

.upper()

- Similar to `.title()`, we can also use `.upper()` to capitalize all letters in a string.

```
# ask user to input name, Remove white spaces from the string, and
Capitalize the all letters
name = input("Please input your name: ").strip().upper()

# print output
print(f"Hello, {name}!")
```

- If we input `jin`, the output would be `Hello, JIN!`

.split()

- The `split()` method splits a string into a list.

- We can specify the separator, default separator is any whitespace. For example,

```
# ask user to input name, including first name and last name
name = input("Input first name, then last name (e.g., Jin Wang): ")

# split the name into two parts
x = name.split()

# print output
print(f"First name: {x[0]}")
print(f"First name: {x[1]}")
```

- In this program, if we input "Jin Wang", `x` value would be a list `['Jin', 'Wang']`
- We can also specify the separator. For example,

```
# ask user to input name, including first name and last name
name = input("Input first name, then last name (e.g., Wang, Jin): ")

# split the name into two parts
x = name.split(',')

# print output
print(f"First name: {x[1].strip()}")
print(f"First name: {x[0].strip()}")
```

- We can obtain the same output as the previous code.

4. Traversal with a `for` loop

- A lot of computations involve processing a string one character at a time.
- Often they start at the beginning, select each character in turn, do something to it, and continue until the end.
- This pattern of processing is called a *traversal*. One way to write a traversal is with a while loop:

```
s = "Hello"
i = 0
while i<len(s):
    letter = s[i]
    print(letter)
    i += 1
```

- As an exercise, write a function that takes a string as an argument and displays the letters *backward*, one per line.
- Another way to write a traversal is with a for loop:

```
s = "Hello"
for l in s:
    print(l)
```

5. Applications:

Searching

- With loop, we can do a lot of interesting operations on strings. For example, we can try to find out *whether there is a particular character in a string*.
- Try to understand the following function.

```
def find(word, letter):
    index = 0
    while index<len(word):
        if word[index] == letter:
            return index
        index = index + 1
    return -1
```

- This program takes a word and a character and finds the index where that character appears in the word. If the character is not found, the function returns -1.
- As an exercise, try to use `for` loop to implement the function.

Looping and counting

- We can also use loop to count the number of times a letter appears in a string. For example,

```
word = 'banana'
count = 0
for letter in word:
    if letter == 'a':
        count = count + 1
print(count)
```

- This program demonstrates another pattern of computation called a counter. The variable count is initialized to 0 and then incremented each time an "a" is found.

7. 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']
```

8. 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']
```

9. 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]
```

10. 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.

11. 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])
```

12. 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
```

13. 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.

14. 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))
```

15. 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))
```

16. 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)
```

Practice questions

1. Use two `input` function twice, one for inputing first name and another one for last name. Use `.strip()` and `.title()` methods to correct user's input. Then print "Last name, First name". For example, input: " jin" as first name, "wang" as last name, the output would be "Wang, Jin".

2. Given an address, e.g., "100 Rockafeller Road, Piscataway, NJ 08854". Try to split the string, and get the information, street, city, state, zipcode.

3. 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.

4. 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.

5. 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.