Topic 7: Set and Tuple

1. Sets

Defining sets

- Sets are another collection/container, like lists, which contain multiple values.
- The **key differences** are
  - **Sets don't hold order**
  - **Sets don't allow duplicate elements**
- We can define a set using curly bracket, for example

```
>>> universities = {'Rutgers', 'NYU', 'Princeton'}
>>> universities
{'Rutgers', 'Princeton', 'NYU'}
```

Set size

- List lists, `len()` returns set size.

```
>>> universities = {'Rutgers', 'NYU', 'Princeton'}
>>> len(universities)
3
```

`in` Operation

- Like lists, sets also have `in` operation.

```
universities = {'Rutgers', 'NYU', 'Princeton'}

x = 'NYU'
if x in universities:
    print(f'{x} is in the set.')
else:
    print(f'{x} isn't in the set.')
```

List and set

- Set and list are interchangeable. It would be very useful to delete the duplicates by transforming a list to a set.

```
>>> l = [2, 0, 2, 3]
>>> s = set(l)
>>> s
{0, 2, 3}
>>> l_new = list(s)
[0, 2, 3]
```

2. Set Methods and Operations

`add()` method

- We can use `add()` method to add an element into a set. But **the new element could be anywhere in the set** since sets don't keep order.

```
>>> universities = {'Rutgers', 'NYU', 'Princeton'}
>>> universities.add('Columbia')
>>> universities
{'Rutgers', 'Columbia', 'Princeton', 'NYU'}
```

- If we try to do it again, that is, add 'Columbia' again, we will get the same results. **It is because sets don't allow duplicate elements.**

`union()` method and `|` operator

- Set union can combine sets.

```
>>> set_1 = {1, 2, 3}
>>> set_2 = {2, 3, 4}
>>> set_1.union(set_2)
{1, 2, 3, 4}
```

- More than two sets may be specified with the method.

```
>>> set_1 = {1, 2, 3}
>>> set_2 = {2, 3, 4}
>>> set_3 = {5, 6, 7}
>>> set_1.union(set_2, set_3)
{1, 2, 3, 4, 5, 6, 7}
```

- We can also use `|` operator to get the same result

```
>>> set_1 = {1, 2, 3}
>>> set_2 = {2, 3, 4}
>>> set_3 = {5, 6, 7}
>>> set_1|set_2|set_3
{1, 2, 3, 4, 5, 6, 7}
```

`intersetion()` method and `&` operator

- `intersetion()` computes the intersection of two or more sets.
- `&` operator also computes the intersection.

```
>>> set_1 = {1, 2, 3}
>>> set_2 = {2, 3, 4}
>>> set_1.intersection(set_2)
{2, 3}
>>> set_1 & set_2
```

`difference()` method and `-` operator

- `difference()` and `-` operator compute the difference between two or more sets.

```
>>> set_1 = {1, 2, 3}
>>> set_2 = {2, 3, 4}
>>> set_3 = {5, 6, 7}

>>> set_1.difference(set_2)
{1}
>>> set_1.difference(set_2, set_3)
{1}
```

```
>>> set_1 - set_2 - set_3
{1}

>>> set_2 - set_1
{4}
```

- `set_1.difference(set_2)` or `set_1 - set_2` returns a set containing elements that are in `set_1` but not in `set_2`.

3. Modifying a Set

`remove()` method

- To delete an element, we can use `remove()` methods.

```
>>> universities = {'Rutgers', 'NYU', 'Princeton'}
>>> universities.remove('NYU')
>>> universities
{'Rutgers', 'Princeton'}
```

- If we try to remove an element that doesn't exist in the set, there would be an error.

```
>>> universities = {'Rutgers', 'NYU', 'Princeton'}
>>> universities.remove('Harvard')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: 'Harvard'
```

`discard()` method

- `x.discard(<elem>)` also removes `<elem>` from `x`. However, if `<elem>` is not in `x`, this method quietly does nothing instead of raising an exception.

```
>>> universities = {'Rutgers', 'NYU', 'Princeton'}
>>> universities.discard('Harvard')
```

4. Tuple

Defining tuples

- A tuple is a comma-separated list. We can define a tuple like this:

```
>>> t = 'a', 'b', 'c', 'd'
>>> t
```

```
('a', 'b', 'c', 'd')
```

- We can also enclose tuples in parentheses:

```
>>> t = ('a', 'b', 'c', 'd')
>>> t
('a', 'b', 'c', 'd')
```

- Specially, to create a tuple with a single element, you have to include a final comma:

```
>>> t1 = 'a',
>>> t1
('a',)
```

- The comma is a must, otherwise it is a string, not a tuple. Even we put a single element in parentheses without comma, it is still not a tuple.

```
>>> t1 = ('a')
>>> type(t1)
<class 'str'>
```

Slice operator

- The slice operator selects a range of elements.

```
>>> t = ('a', 'b', 'c', 'd')
>>> t[0]
'a'
>>> t[1:3]
('b', 'c')
```

Tuples are immutable

- Tuples are immutable, and you cannot modify the elements.

```
>>> t = ('a', 'b', 'c', 'd')
>>> t[0] = 'A'
TypeError: 'tuple' object does not support item assignment
```

5. Tuple assignment

- We often need to define several similar variables and assign values to the variables.

```
a = 1
b = 2
c = 3
```

- The conventional method works fine. But with tuple assignment, we can make the code more elegant.

```
a, b, c = 1, 2, 3
```

- Sometimes we need to swap the values of two variables. For example, we would like to swap the values in variables `a` and `b`. It can be done like this:

```
a = 1
b = 2

temp = a
a = b
b = temp

print(a)
print(b)
```

- The above solution seems cumbersome. **Tuple assignment** is more elegant:

```
a = 1
b = 2

a, b = b, a

print(a)
print(b)
```

- In the tuple assignment, `a, b = b, a`, the left side is a tuple of variables; the right side is a tuple of expressions. Each value is assigned to its respective variable. All the expressions on the right side are evaluated before any of the assignments.
- Note: *The number of variables on the left and the number of values on the right have to be the same.*

```
>>> a, b = 1, 2, 3
ValueError: too many values to unpack
```

- Tuple assignment is very useful when we would like to assign values to several variables simultaneously. For example, we learned this code before, which is to extract information from an address string.

```
address = "100 Rockafeller Rd, Piscataway, NJ"
address_list = address.split(', ')

street = address_list[0]
city = address_list[1]
state = address_list[2]
```

- With tuple assignment, we can finish it elegantly.

```
address = "100 Rockafeller Rd, Piscataway, NJ"
address_list = address.split(', ')

street, city, state = address_list
```

- We can see that Python is not that strict. In the last line, the left side is a tuple, while the right slide is a list. It works well, which means that Python transform a list into a tuple automatically. For sure, the above code can be further improved as follows.

```
address = "100 Rockafeller Rd, Piscataway, NJ"
street, city, state = address.split(', ')
```

Tuples as return values

- Strictly speaking, a function can only return one value, but if **the value is a tuple**, the effect is the same as returning multiple values.
- For example, if you want to divide two integers and compute the quotient and remainder, it is inefficient to compute `x//y` and then `x%y` as follows.

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

def get_remainder(x, y):
    return x%y

x, y = 7, 3
quot, rem = get_quotient(x, y), get_remainder(x, y)
```

- It is better to compute them both at the same time.

```
def get_quot_rem(x, y):
    return x//y, x%y

x, y = 7, 3
quot, rem = get_quot_rem(x, y)
```

6. Variable-length argument tuples

- If we would like to create a function to calculate square value of **a given number**, we can do this:

```
def square_number(x):
    return x ** 2

a = 2
square_value = square_number(2)
print(square_value)
```

- What about square values of **more than one values**? We may create the following function, which takes list.

```
def square_list(x):
    return [i ** 2 for i in x]

a = [2, 3, 4]
squares = square_list(a)

print(squares)
```

- It works. However, how about we aren't sure whether we need to calculate square value for a single value or more than one values? In other words, can we create a function, which can handle variable-length argument? **What is Python *args?**
- Functions can take a variable number of arguments. *A parameter name that begins with * gathers arguments into a tuple*.
- To solve the above question, we can create a function with *args:

```
def square(*args):
    return [i ** 2 for i in args]

squares_multiple = square(2, 3, 4)
print(squares_multiple)

squares_single = square(2)
print(squares_single)
```

- Note: The function can take one or more arguments, but cannot take a list.

7. Lists and tuples

`zip()` function

- `zip()` is a built-in function that takes two or more sequences and interleaves them. The name of the function refers to a zipper, which interleaves two rows of teeth.
- Sometimes we would like to collect information for different items. For example, we have lists about County, City, and Zipcode.

```
counties = ['MIDDLESEX', 'MIDDLESEX', 'SOMERSET']
cities = ['Piscataway', 'HIGHLAND PARK', 'FLAGTOWN']
zipcodes = ['08854', '08904', '08821']

print(zip(counties, cities, zipcodes))
```

- We will get

```
<zip object at xxxx090xxxxxx>
```

- The result is a zip object that knows how to iterate through the pairs. The most common use of zip is in a `for` loop or transform it to a list:

```
counties = ['MIDDLESEX', 'MIDDLESEX', 'SOMERSET']
cities = ['Piscataway', 'HIGHLAND PARK', 'FLAGTOWN']
zipcodes = ['08854', '08904', '08821']

for pair in zip(counties, cities, zipcodes):
    print(pair)
```

- We will get

```
('MIDDLESEX', 'Piscataway', '08854')
('MIDDLESEX', 'HIGHLAND PARK', '08904')
('SOMERSET', 'FLAGTOWN', '08821')
```

- We can also transform it to a list

```
counties = ['MIDDLESEX', 'MIDDLESEX', 'SOMERSET']
cities = ['Piscataway', 'HIGHLAND PARK', 'FLAGTOWN']
zipcodes = ['08854', '08904', '08821']
```

```
l_pairs = list(zip(counties, cities, zipcodes))

print(l_pairs)
```

- We will get a list with three tuples.

```
[('MIDDLESEX', 'Piscataway', '08854'), ('MIDDLESEX', 'HIGHLAND PARK',
'08904'), ('SOMERSET', 'FLAGTOWN', '08821')]
```

`enumerate()` function

- The result from `enumerate()` is an enumerate object, which iterates a sequence of pairs; each pair contains an index (starting from 0) and an element from the given sequence.
- For example:

```
universities = ["Princeton", "MIT", "Harvard", "Stanford"]

for ranking, university in enumerate(universities):
    print(ranking, university)
```

8. Practice questions

1. We've provided you with a list of lottery players, and also with 6 random lottery numbers. The random lottery numbers are generated like this:

```
import random
lottery_numbers = set(random.sample(list(range(22)), 6))
```

And the list of players we've given you are:

```
players = [("Rolf", {1, 3, 5, 7, 11, 20}),
("Charlie", {2, 7, 9, 5, 12, 15}),
("Anna", {7, 8, 1, 3, 13, 16}),
("Jen", {4, 7, 3, 5, 12, 21})
]
```

Try to find out the number of winnings for each person. For example, if the lottery number is `6, 8, 9, 13, 16, 19`, you need to print:

```
Rolf won 0.
Charlie won 1.
```

```
Anna has won 3.
Jen has won 0.
```

2. Try to create a function, which can take variable-length arguments and return the square root of the arguments.

3. Summary data. Create a function, named `summarize_data`, which takes a list of numbers and returns a tuple containing the minimum, mean, and the maximum values. For example, given `[1, 2, 5]`, it would return `1, 2.6666, 5`

    - You may use some built-in functions, like `sum`, `min`, and `max` .

4. **(Optional)** Letter frequency. Create a function, named `letter_frequency`, which takes a string and prints the letters in the string (case-insensitive, or just use upper case) and corresponding frequency. For example,

    - Given 'Rutgers, RBS', the function would print `[('G', 1), ('U', 1), ('B', 1), ('T', 1), ('R', 3), (',', 1), ('E', 1), ('S', 2)]`