ITECH2302 Big Data Management

Foundation of Python programming
Two-dimensional List, Dictionary, Sets

Task 1: List methods

In this Task, we learn basic operation on list and list's main methods, including sorting lists, searching sequences, filter and map methods, etc.

1. Removing elements from a list

Type the following codes:

```
In [42]: numbers = list(range(0, 10))

In [43]: numbers
Out[43]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [44]: numbers[-10]
Out[44]: 0

In [45]: numbers[-1]
Out[45]: 9

In [46]: del numbers[-1]

In [47]: numbers
Out[47]: [0, 1, 2, 3, 4, 5, 6, 7, 8]
```

We use **del** statement(line 46) to remove elements from a list. Note that lists can be accessed from the end of list by using negative indices. Here the list number's last element(numbers[9] can be accessed with numbers[-1] and its first element with numbers[-10].

2. Passing lists to functions

Type the following codes:

```
In [52]: def modify_elements(items):
    ...:     """multiplies all element values in items by 2"""
    ...:     for i in range(len(items)):
    ...:         items[i] *= 2
    ...:

In [53]: numbers = [ 2, 5, 6, 1, 8]

In [54]: modify_elements(numbers)

In [55]: numbers
Out[55]: [4, 10, 12, 2, 16]
```

In the codes above, function **modify_elements** receives a reference to a list and multiplies each of the list's element values by 2. More precisely, items parameter of function modify_elements receives a reference to the original list(numbers), therefore statement in the loop modifies each element in the original list objects.

3. Sorting lists

Sorting task is used to arrange data either in ascending or descending order.

(1) Sorting a list in ascending order

Type the following codes:

(2)

```
In [55]: numbers
Out[55]: [4, 10, 12, 2, 16]

In [56]:  numbers =[ 2, 10, 4, 1, 5, 9, 7, 3, 6, 8]

In [57]: numbers.sort()

In [58]: numbers
Out[58]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Sorting a list in descending order

We can use list method **sort** with optional keyword argument **reverse** set to **True** (False is the default) to sort a list in descending order.

Type the following codes:

(3) Using built-in function *sorted*

```
In [59]: numbers.sort(reverse = True)

In [60]: numbers
Out[60]: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

List provides a built-in function called ***sorted*** that returns a new list containing the sorted elements of its argument sequence, but the original sequence is unmodified.

Type the following codes:

4.

```
In [61]:  numbers =[ 2, 10, 4, 1, 5, 9, 7, 3, 6, 8]

In [62]: ascending_numbers = sorted(numbers)

In [63]: ascending_numbers
Out[63]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

In [64]: numbers
Out[64]: [2, 10, 4, 1, 5, 9, 7, 3, 6, 8]

In [65]: colors =('red','orange','yellow','green','blue')

In [66]: ascending_colors = sorted(colors)

In [67]: ascending_colors
Out[67]: ['blue', 'green', 'orange', 'red', 'yellow']

In [68]: colors
Out[68]: ('red', 'orange', 'yellow', 'green', 'blue')
```

Searching sequences

If we want to determine whether a sequence(such as a list, tuple or string) contains a value that matches a particular key value, we can use search method. Searching is the process of locating a key in a sequence.

(1) Using list method *index*

List method **index** takes as an argument a search key, value to locate in the list, then searches through the list from index 0 and returns the index of the first element that matches the search key:

Type the following codes:

```
In [1]: numbers = [2, 5, 1, 4, 3, 8,7]

In [2]: numbers.index(8)
Out[2]: 5

In [3]:
```

(2) Specifying starting index of a search

Specifying the starting and ending indices causes index to search from the starting index up to but not including the ending index location.

Type the following codes:

```
In [1]: numbers = [2, 5, 1, 4, 3, 8,7]

In [2]: numbers.index(8)
Out[2]: 5

In [3]: numbers.index(4, 2)
Out[3]: 3
```

In the codes above, line 3 searches the value 4 starting from index 2 and continuing through the end of the list.

If the search starts from the index 5, a **ValueError** occurs because the value 4 is not in the remaining list.

Type the following codes:

```
In [5]: numbers.index(4, 5)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-5-c0ba33250d53> in <module>
----> 1 numbers.index(4, 5)

ValueError: 4 is not in list
```

(**3) Operators** *in* **and** *not in*

Operator **in** tests whether its right operand's list contains the left operand's value:

```
In [6]: numbers = [2, 5, 1, 4, 3, 8,7]

In [7]: 20 in numbers
Out[7]: False

In [8]: 7 in numbers
Out[8]: True

In [9]:
```

Type the following codes:

```
In [9]: numbers = [2, 5, 1, 4, 3, 8,7]

In [10]: 20 not in numbers
Out[10]: True

In [11]: 7 not in numbers
Out[11]: False

In [12]:
```

(4) Using operator in **to prevent a ValueError**

Type the following codes

```
In [12]: numbers = [2, 5, 1, 4, 3, 8,7]

In [13]: key = 50

In [14]: if key in numbers:
    ...:     print(f'found{key}at index {numbers.index(search_key)}')
    ...: else:
    ...:     print(f'{key}  not found')
    ...:
50  not found

In [15]:
```

In the codes above, operator **in** ensures that calls to method index do not result in ValueErrors for search key(50) that is not in the sequence.

5. Using Filter and Map functions

We can use the built-in **filter** and **map** functions for filtering and mapping tasks.

(1) Using built-in filter function to filter a sequence's values

Type the following codes

In the above,

```
In [1]: numbers = [1, 4, 5, 2, 3, 10, 9, 7, 8, 6]

In [2]: def is_odd(x):
    ...:     """Returns true only if x is odd. """
    ...:     return x % 2 !=0
    ...:

In [3]: list(filter(is_odd, numbers))
Out[3]: [1, 5, 3, 9, 7]

In [4]:
```

codes **filter** function's first argument must be a function that receives one argument and returns **True** if the value should be included in the result. The self-defined function **is_odd** returns True if its argument is odd. The filter function calls **is_odd** once for each value in its second argument's iterable(numbers).

(2) Using a lambda expression

We can use a **lambda** expression to define the function inline where it's needed, typically as it's passed to another function:

Type the following codes

In the above,

```
In [4]: numbers = [1, 4, 5, 2, 3, 10, 9, 7, 8, 6]

In [5]: list(filter(lambda x: x % 2 != 0, numbers))
Out[5]: [1, 5, 3, 9, 7]

In [6]:
```

codes a **lambda** begins with the lambda keyword followed by a comma-separated parameter list, a colon(**:**) and an expression. In this case, the parameter list has one parameter named x. A **lambda** implicitly returns its expression's value. Here we pass filter's return value (an iterator) to function list to convert the results to a list and display them.

In the filter call, the first argument is the lambda

lambda x: x % 2 != 0

(3) Using map function to map a sequence's values to new values

Type the following codes

In the above,

```
In [6]: numbers = [1, 4, 5, 2, 3, 10, 9, 7, 8, 6]

In [7]: list(map(lambda x: x ** 2, numbers))
Out[7]: [1, 16, 25, 4, 9, 100, 81, 49, 64, 36]

In [8]:
```

codes function map's first argument is a function that receives one value and returns a new value, in this case, a **lambda** that squares its argument. The second argument is an iterable of values to map. Function **map** uses lazy evaluation, so we pass to the list function the iterator that map returns.

6. Creating two- dimensional lists

Lists can contain other lists as elements. We can use this feature to represent tables of values that consist of rows and columns. To identify a particular table element, we need to specify two indices; the first identifies the element's row, and the second the element's column. So lists that require two indices to identify an element are called two-dimensional lists.

(1) Creating a two-dimensional list

Now let us to represent three students' grades with a two-dimensional list:

Type the following codes

```
In [10]: a = [[88, 65, 79, 95],  # first student's grades
    ...:      [79, 77, 80, 60],  # second student's grades
    ...:      [50, 60, 87, 92]]  # third student's grades

In [11]: a
Out[11]: [[88, 65, 79, 95], [79, 77, 80, 60], [50, 60, 87, 92]]

In [12]:
```

We can identify the elements in a two-dimensional list by using **a[i][j]** where ***i*** is row index, ***j*** for column index.

```
In [12]: a[2][3]
Out[12]: 92

In [13]:
```

(2) Executing nested loops

Type the following codes

```
In [13]: a = [[88, 65, 79, 95],  # first student's grades
    ...:      [79, 77, 80, 60],  # second student's grades
    ...:      [50, 60, 87, 92]]  # third student's grades

In [14]: for i, row in enumerate(a):
    ...:     for j, item in enumerate(row):
    ...:         print(f'a[{i}][{j}] = {item} ', end= ' ')
    ...:         print()
    ...:
a[0][0] = 88
a[0][1] = 65
a[0][2] = 79
a[0][3] = 95
a[1][0] = 79
a[1][1] = 77
a[1][2] = 80
a[1][3] = 60
a[2][0] = 50
a[2][1] = 60
a[2][2] = 87
a[2][3] = 92

In [15]:
```

In the codes above, the outer for statement iterates over the two-dimensional list's rows one row at a time. During each iteration of the outer for statement, the inner for statement iterates over each column in the current row. For example, in the first iteration of the outer loop, row 0 is [88, 65, 79, 95].

Task 2: Dictionaries and Sets

In the previous Tasks, we have discussed built-in sequence collections such as lists and tuples. In this and next Tasks, we study built-in non-sequence collections, **dictionaries** and **sets**. A dictionary is an **unordered** collection which stores **key-value pairs** that map immutable keys to values, just as a conventional dictionary maps words to definitions. A set is an **unordered** collection of unique immutable elements.

1. Creating a Dictionary

A dictionary associates **keys** with **values**. Each key maps to a specific value. The following table gives examples of dictionaries with their keys, key types, values and value types

Keys	Key type	Values	Value type
Country names	str	Internet country code	str
Decimal numbers	int	Roman numerals	str
States	str	Agricultural products	list of str
Hospital patients	str	Vital signs	tuple of ints and floats
Baseball players	str	Batting averages	float

Please note that a dictionary's keys must be **immutable**(such as strings, numbers or tuples) and **unique**(that is, no duplicates). Multiple keys can have the same values, such as two different inventory codes that have the same quantity in stock.

Type the following codes

```
In [1]: country_codes = {'Finland ': 'fi','South Africa':'za','Nepal':'np'}

In [2]: country_codes
Out[2]: {'Finland ': 'fi', 'South Africa': 'za', 'Nepal': 'np'}

In [3]:
```

(1) Determining if a dictionary is empty

We can use a dictionary as a condition to determine if it's empty because a non-empty dictionary evaluates to **True**:

Type the following codes

(2) Iterating through a dictionary

```
In [5]: if country_codes:
   ...:     print('country_codes is not empty')
   ...: else:
   ...:     print('country_codes is empty')
   ...:
country_codes is not empty

In [6]:
```

We first use a dictionary to map month-name string to **int** values that represent the numbers of days in the corresponding month.

Type the following codes

In the above, use **for** codes we then use statement(line 11) to iterate through days_per_month's key-value pairs. Dictionary method **items** returns each key-value pair as a tuple that is unpacked into month and days.

```
In [9]: days_per_month = {'January':31,'February': 28, ' March': 31}

In [10]: days_per_month
Out[10]: {'January': 31, 'February': 28, ' March': 31}

In [11]:  for month, days in days_per_month.items() :
   ...:     print(f'{month} has {days} days')
   ...:
   ...:
January has 31 days
February has 28 days
 March has 31 days
```

(3) Using basic dictionary operations

We first create a dictionary **roman_numerals** and access the values associated with a key.

Updating an key-value pair values of existing

```
In [12]: roman_numerals = {'I': 1, 'II': 2, 'III': 3, 'V' : 5,'X': 100}

In [13]: roman_numerals['V']
Out[13]: 5

In [14]:
```

Type the following codes

```
In [16]: roman_numerals['X'] = 10

In [17]: roman_numerals
Out[17]: {'I': 1, 'II': 2, 'III': 3, 'V': 5, 'X': 10}

In [18]:
```

In the codes above, we replace value 100 with 10, associated with the key "X".

Adding and removing a key-value pair.

Type the following codes

```
In [18]: roman_numerals['L'] = 50

In [19]: roman_numerals
Out[19]: {'I': 1, 'II': 2, 'III': 3, 'V': 5, 'X': 10, 'L': 50}

In [20]: del roman_numerals['III']

In [21]: roman_numerals
Out[21]: {'I': 1, 'II': 2, 'V': 5, 'X': 10, 'L': 50}

In [22]:
```

(4) Application:

Application 1 - create a dictionary of student grades

The following example uses a dictionary to map each student's name(a string) to a list of integers containing that student's grades on three exams.

Type the following codes:

After this we get :

```
1   #
2   """Using a dictionary to represent a student grades."""
3   grade_student = {
4       'Susan': [92, 85, 100],
5       'Eduardo': [83, 95, 79],
6       'Azizi': [91, 89, 82],
7       'Pantipa': [97, 91, 92]
8   }
9
10  all_grades_total = 0
11  all_grades_count = 0
12
13  for name, grades in grade_student.items():
14      total = sum(grades)
15      print(f'Average for {name} is {total/len(grades):.2f}')
16      all_grades_total += total
17      all_grades_count += len(grades)
18
19  print(f"Class's average is: {all_grades_total / all_grades_count:.2f}")
20
21
```

running code,

```
Average for Susan is 92.33
Average for Eduardo is 85.67
Average for Azizi is 87.33
Average for Pantipa is 93.33
Class's average is: 89.67
```

In the codes above, lines 13 -17 unpack a key-value pair into the variables name and grades containing one student's name and the corresponding list of three grades. Line 14 uses built-in function ***sum*** to total a given student's grades, then line 15 calculates and displays that student's average by dividing total by the number of grades for that student(***len***(grades)). Lines 16-17 keep track of the total of all four students' grades and the number of grades for all the students.

Application 2 - word counts

The following example uses a dictionary to count the number of occurrences of each word in a string.

Type the following codes:

After

```
1  #
2  """Tokenizing a string and counting unique words."""
3
4  text = ('this is sample text with several words '
5          'this is more sample text with some different words')
6
7  word_counts = {}
8
9  # count occurrences of each unique word
10 for word in text.split():
11     if word in word_counts:
12         word_counts[word] += 1  # update existing key-value pair
13     else:
14         word_counts[word] = 1  # insert new key-value pair
15
16 print(f'{"WORD":<12}COUNT')
17
18 for word, count in sorted(word_counts.items()):
19     print(f'{word:<12}{count}')
20
21 print('\nNumber of unique words:', len(word_counts))
22
```

running this code, we get :

```
WORD        COUNT
different   1
is          2
more        1
sample      2
several     1
some        1
text        2
this        2
with        2
words       2

Number of unique words: 10
```

In the codes above, lines 4-5 create a string text that we'll break into words--a process known as tokenizing a string. Line 10 tokenizes text by calling string method split that separates the words using the method's delimiter string argument. If we do not provide an argument, split uses a space. The method returns a list of tokens(that is, the words in text). Line 10-14 iterate through the list of words.

For each word, line 11 determines whether that word(the key) is already in the dictionary. If so, line 12 increments that word's count; otherwise, line 14 inserts a new key-value pair for that word with an initial count of 1.

Lines 16-21 summarize the results in a two-column table containing each word and its corresponding count. The for statement in lines 18 and 19 iterates through the dictionary's key-value pairs. It unpacks each key and value into the variables word and count.

Task 3: Sets

A set is an unordered collection of **unique** values. Sets may contain only immutable objects, like **strings**, **ints**, **floats** and **tuples** that contain only immutable elements. Though sets are iterable, they are **not sequences** and **do not support indexing** and slicing with square brackets, [].

1. Creating a set with curly braces

Type the following codes:

```
In [22]: colors = {'red','orange','yellow','green','red','blue'}

In [23]: colors
Out[23]: {'blue', 'green', 'orange', 'red', 'yellow'}
```

In the codes above, please note that the duplicate string 'red' was ignored(without error). A useful use of sets is **duplicate elimination**, which is automatic when creating a set. Though the color names are displayed in sorted order, sets are unordered.

2. Determining a set's length and checking whether a value is in a set

Type the following codes:

3.

```
In [22]: colors = {'red','orange','yellow','green','red','blue'}

In [23]: colors
Out[23]: {'blue', 'green', 'orange', 'red', 'yellow'}

In [24]: len(colors)
Out[24]: 5

In [25]: 'red' in colors
Out[25]: True

In [26]: 'purple' in colors
Out[26]: False

In [27]: 'purple' not  in colors
Out[27]: True
```

Iterating through a set

We can process each set element with a **for** loop because sets are iterable.

Type the following codes:

4.

```
In [28]: colors = {'red','orange','yellow','green','red','blue'}

In [29]: for color in colors:
    ...:     print(color.upper(), end= ' ' )
    ...:
GREEN ORANGE RED BLUE YELLOW
```

Mathematical set operations

In this section, we discuss set type's mathematical operators **|, &** and **^.**

(1) Union

The union of two sets is a set consisting of all the unique elements from both sets. We can calculate the union with the **|** operator or with the set type's union method.

Type the following codes:

```
In [30]: {1, 3, 5} | {2, 3, 4}
Out[30]: {1, 2, 3, 4, 5}

In [31]: {1, 3, 5}.union([20 , 20, 3, 40, 40])
Out[31]: {1, 3, 5, 20, 40}
```

In the above codes, the operands of the **binary** set operators, like |, must both be sets.

(2) Intersection

The intersection of two sets is a set consisting of all the unique elements that the two sets have in common. We can calculate the intersection with the **&** operator or with the set type's intersection method.

Type the following codes:

```
In [32]: {1, 3, 5} & {2, 3, 4}
Out[32]: {3}

In [33]: {1, 3, 5}.intersection([1, 2, 2, 3, 3, 4, 4])
Out[33]: {1, 3}
```

(3) Difference

The difference between two sets is a set consisting of the elements in the left operand that are not in the right operand. We can calculate the difference with the **-** operator or with the set type's difference method.

Type the following codes:

```
In [34]: {1, 3, 5} -  {2, 3, 4}
Out[34]: {1, 5}

In [35]: {1, 3, 5, 7}.difference([ 2, 2, 3, 3, 4, 4])
Out[35]: {1, 5, 7}
```

Task 4. Answering questions (Please do this at your home by using your own computer)

1. What are main purposes of variable transformation in data pre-processing? How was it implemented?

2. In the following codes, the dictionary temperatures contains three Fahrenheit temperature samples for each of four days. What does the **for** statement do?

```
temperatures = {
'Monday': [66, 70, 74],
'Tuesday': [50, 56, 64],
'Wednesday': [75, 80, 83],
'Thursday': [67, 74, 81]
}
for k, v in temperatures.items():
    print(f'{k}: {sum(v)/len(v):.2f}')
```