02 Python - listing Lists

List Methods

index

```python
spam = ['hello', 'hi', 'howdy', 'heyas']
print(spam.index('hello')) # Output: 0
print(spam.index('heyas')) # Output: 3
```

1. The index() method in Python is used to find the first occurrence of an item in a list and returns the index of that item.
2. If the item is not found in the list, a ValueError is raised.
3. The index() method returns the index of the first occurrence of the item in the list.
4. If the list contains duplicates of the item, the index of the first occurrence is returned.
5. In this example, we create a list called `spam` with four string elements. We then use the `index()` method to find the index of the string `'hello'` and the string `'heyas'`. The first call to `index()` returns `0` because `'hello'` is the first element of the list, and the second call returns `3` because `'heyas'` is the last element of the list.

append

```python
spam = ['cat', 'dog', 'bat']
spam.append('moose')
print(spam) # Output: ['cat', 'dog', 'bat', 'moose']
```

1. The append() method adds an element to the end of the list.
2. The append() method modifies the list in place and doesn't return any value.
3. The append() method is a list method and can be called only on list values.
4. Syntax: list_name.append(element)

insert

```python
spam = ['cat', 'dog', 'bat']
spam.insert(1, 'chicken')
print(spam) # Output: ['cat', 'chicken', 'dog', 'bat']
```

1. The insert() method inserts an element at the specified index of the list.
2. The insert() method modifies the list in place and doesn't return any value.
3. The insert() method is a list method and can be called only on list values.
4. Syntax: list_name.insert(index, element)

remove

```python
spam = ['cat', 'bat', 'rat', 'elephant']
spam.remove('bat')
print(spam)  # Output: ['cat', 'rat', 'elephant']
```

1. The remove() method is used to remove a specific value from a list.
2. The method is called on the list and is passed the value to be removed as an argument.
3. If the value is not in the list, a ValueError error will be raised.
4. If the value appears multiple times in the list, only the first instance will be removed.
5. The del statement can be used when you know the index of the value to be removed.

sort

```python
# Sort a list of numbers in ascending order
numbers = [2, 5, 3.14, 1, -7]
numbers.sort()
print(numbers)  # Output: [-7, 1, 2, 3.14, 5]

# Sort a list of strings in ascending order
words = ['ants', 'cats', 'dogs', 'badgers', 'elephants']
```

```python
words.sort()
print(words)  # Output: ['ants', 'badgers', 'cats', 'dogs', 'elephants']

# Sort a list of strings in descending order
words.sort(reverse=True)
print(words)  # Output: ['elephants', 'dogs', 'cats', 'badgers', 'ants']

# Sort a list of strings in regular alphabetical order
words = ['Alice', 'ants', 'Bob', 'badgers', 'Carol', 'cats']
words.sort(key=str.lower)
print(words)  # Output: ['Alice', 'ants', 'badgers', 'Bob', 'Carol', 'cats']
```

1. The `sort()` method can be used to sort lists of number values or lists of strings in ascending order.
2. The `sort()` method sorts the list in place, meaning that it modifies the original list and does not return a new list.
3. The `reverse` keyword argument can be set to True to sort the list in descending order.
4. Lists that have both number values and string values cannot be sorted using the `sort()` method since Python cannot compare these values.
5. By default, the `sort()` method uses "ASCIIbetical order" for sorting strings, which means that uppercase letters come before lowercase letters. However, you can pass `str.lower` as the `key` argument to sort the values in regular alphabetical order.

del

```python
# Create a list and a variable
my_list = ['apple', 'banana', 'cherry', 'date']
my_var = 42

# Delete an item from the list
del my_list[2] # removes 'cherry'
print(my_list) # output: ['apple', 'banana', 'date']

# Delete a slice from the list
del my_list[1:3] # removes 'banana' and 'date'
print(my_list) # output: ['apple']

# Delete a variable
del my_var
print(my_var) # output: NameError: name 'my_var' is not defined

# Try to delete an item that doesn't exist in the list
del my_list[2] # output: ValueError: list assignment index out of range

# Try to delete a variable that doesn't exist
del my_missing_var # output: NameError: name 'my_missing_var' is not defined
```

1. The `del` statement removes an item or slice from a list and shifts the remaining items to fill the gap.
2. It can also be used to delete variables and clear their values.
3. Using `del` on an item that doesn't exist in a list will result in a `ValueError` error.
4. Using `del` on a variable that has not been defined will result in a `NameError` error.

Passing reference

```python
# Assigning an integer to a variable
spam = 42
cheese = spam
spam = 100
print(spam)   # Output: 100
print(cheese) # Output: 42

# Assigning a list to a variable
spam = [0, 1, 2, 3, 4, 5]
cheese = spam
cheese[1] = 'Hello!'
print(spam)   # Output: [0, 'Hello!', 2, 3, 4, 5]
print(cheese) # Output: [0, 'Hello!', 2, 3, 4, 5]
```

1. In Python, variables can store either values or references to values.
2. When you assign an integer or a string to a variable, the variable contains the actual value, not a reference to it.
3. When you assign a list to a variable, the variable contains a reference to the list, not the list itself.
4. When you copy a list variable to another variable, you're actually copying the reference to the list, not the list itself.
5. Modifying the list through one variable will affect all other variables that reference the same list.

6. In the first example, changing the value of `spam` does not affect the value of `cheese`, because the variable `cheese` contains a copy of the value in `spam`, not a reference to it.
7. In the second example, changing the value of `cheese[1]` affects the value of `spam`, because both `spam` and `cheese` contain references to the same list.

Copy and DeepCopy

```
import copy

original_list = ['hello', 'world', [1, 2, 3]]
copied_list = copy.copy(original_list)
deepcopied_list = copy.deepcopy(original_list)

# modifying the original list and the copied list
original_list[0] = 'goodbye'
original_list[2].append(4)

print(original_list)      # Output: ['goodbye', 'world', [1, 2, 3, 4]]
print(copied_list)        # Output: ['hello', 'world', [1, 2, 3, 4]]
print(deepcopied_list)    # Output: ['hello', 'world', [1, 2, 3]]
```

1. `copy()` function is used to make a shallow copy of a mutable object.

2. A shallow copy is a new object that is created with a new memory address, but its contents still refer to the original object.

3. Changes made to the original object will also be reflected in the copied object.

4. It is useful for creating a new object with the same data but not modifying the original object.

5. `deepcopy()` function is used to make a deep copy of a mutable object.

6. A deep copy is a new object with new memory address and all its contents are recursively copied as well.

7. Changes made to the original object will not affect the copied object, and vice versa.

8. It is useful for creating a new object with the same data but modifying it independently from the original object.

Tuple

1. A tuple is a collection of ordered and immutable elements enclosed in parentheses (). Once created, the elements of a tuple cannot be modified. Tuples are similar to lists in that they can store any type of data, including integers, floats, strings, and other tuples

2. Immutable: Unlike lists, tuples are immutable, which means that once created, their contents cannot be changed. This makes tuples useful for storing data that should not be changed, such as constants or configuration settings.

3. Ordered: Like lists, tuples are ordered, which means that the order of their elements is important. The order of the elements in a tuple is determined by their position, or index, within the tuple.

4. Can contain any data type: Tuples can store any type of data, including integers, floats, strings, and other tuples. This makes tuples a versatile data structure that can be used in a variety of applications.

5. Accessing elements: Elements in a tuple can be accessed using their index, just like with a list. For example, if t is a tuple containing the values (1, 2, 3), then t[0] would return 1.

6. Creating tuples: Tuples are created by enclosing a sequence of elements in parentheses, separated by commas. For example, to create a tuple containing the values 1, 2, and 3, you would write (1, 2, 3).

7. Converting between tuples and lists: It is possible to convert a tuple to a list and vice versa. To convert a tuple to a list, you can use the list() function. For example, if t is a tuple containing the values (1, 2, 3), you can convert it to a list using the following code: l = list(t). To convert a list to a tuple, you can use the tuple() function. For example, if l is a list containing the values [1, 2, 3], you can convert it to a tuple using the following code: t = tuple(l).

Dictionary Methods

get()

```
person = {'name': 'John', 'age': 25}
print(person.get('name'))  # Output: 'John'
print(person.get('address'))  # Output: None
print(person.get('address', 'N/A'))  # Output: 'N/A'
```

- `get()` method: It returns the value for the specified key if the key is in the dictionary. If the key is not found, it returns a default value.
- It doesn't raise a KeyError if the key is not found, instead it returns the default value or `None` if not specified.
- It takes an optional second argument, which specifies the default value to be returned if the key is not found.
- It's a safer alternative to directly accessing dictionary keys using `[]` because it doesn't throw an error if the key is not present.

keys()

```
person = {'name': 'John', 'age': 25, 'address': '123 Main St'}
keys_view = person.keys()
print(keys_view)  # Output: dict_keys(['name', 'age', 'address'])
person['phone'] = '555-1234'
print(keys_view)  # Output: dict_keys(['name', 'age', 'address', 'phone'])
```

- `keys()` and `items()` are methods in Python's dictionary that return list-like objects containing the keys and key-value pairs, respectively, of the dictionary.
- `keys()` method: It returns a view object that contains the keys of the dictionary. Three points to note about this method are:
  - The view object returned by this method is dynamic, meaning that any changes to the dictionary are reflected in the view object.
  - The order of the keys returned is not guaranteed, and it may vary between different runs of the program.
  - The view object can be used as an iterable in for loops.

values()

```
person = {'name': 'John', 'age': 25, 'address': '123 Main St'}
values_view = person.values()
print(values_view)  # Output: dict_values(['John', 25, '123 Main St'])
person['phone'] = '555-1234'
print(values_view)  # Output: dict_values(['John', 25, '123 Main St', '555-1234'])
```

```
- `values()` is another method in Python's dictionary that returns a list-like object containing the values of the dictionary.
- `values()` method: It returns a view object that contains the values of the dictionary. Three points to note about this method are:
-   The view object returned by this method is dynamic, meaning that any changes to the dictionary are reflected in the view object.
-   The order of the values returned is not guaranteed, and it may vary between different runs of the program.
-   The view object can be used as an iterable in for loops.
```

items()

```
person = {'name': 'John', 'age': 25, 'address': '123 Main St'}
items_view = person.items()
print(items_view)  # Output: dict_items([('name', 'John'), ('age', 25), ('address', '123 Main St')])
person['phone'] = '555-1234'
print(items_view)  # Output: dict_items([('name', 'John'), ('age', 25), ('address', '123
```

```
-  `items()` method: It returns a view object that contains the key-value pairs of the dictionary as tuples. Three points to note about this method are:
-   The view object returned by this method is dynamic, meaning that any changes to the dictionary are reflected in the view object.
-   The order of the key-value pairs returned is not guaranteed, and it may vary between different runs of the program.
-   The view object can be used as an iterable in for loops, and each iteration returns a tuple containing a key-value pair.
```

setdefault()

```python
# Create a dictionary to count the frequency of each letter in a string
my_string = "hello world"
my_dict = {}
for letter in my_string:
    my_dict.setdefault(letter, 0)
    my_dict[letter] += 1
print(my_dict)
```

1. The `setdefault()` method is used to set a default value for a key in a dictionary if the key does not already exist in the dictionary.
2. The method takes two arguments: the key to be checked and the default value to be set if the key is not already in the dictionary.
3. If the key already exists in the dictionary, `setdefault()` returns the existing value for that key. If the key does not exist, the method sets the key to the default value and returns that value.
4. One use case for `setdefault()` is to create a dictionary that counts the frequency of items in a list or other iterable. By setting a default value of 0 for each new key, the method can increment the count for each item in the iterable as it appears in the dictionary.

Pprint

```python
import pprint
message = 'It was a bright cold day in April, and the clocks were striking thirteen.'
count = {}

for character in message:
    count.setdefault(character, 0)
    count[character] = count[character] + 1

pprint.pprint(count)
```

1. The pprint module is used for formatting complex data structures like dictionaries, lists, and tuples in a more readable format.
2. The pprint.pprint() function takes a single argument, which is the data structure to be formatted and printed.
3. The pprint function automatically adds indentation and line breaks to the output, making it easier to read.
4. The given code uses the setdefault() method to initialize the count of each character to 0 in the dictionary, then increments the count for each character as it is encountered in the message string. Finally, the resulting count dictionary is printed using the pprint function.

Nested Dictionary

```python
person = {
    'name': 'Alice',
    'age': 25,
    'address': {
        'street': '123 Main St',
        'city': 'Anytown',
        'state': 'CA',
        'zip': '12345'
    }
}

print(person['name'])           # Output: Alice
print(person['address']['city']) # Output: Anytown
```

1. A dictionary in Python is a built-in data structure that is used to store data in key-value pairs. Key-value pairs: Each element in a dictionary consists of a key-value pair. The key is a unique identifier that is used to access the value associated with it. The value can be of any data type such as string, integer, list, tuple, or even another dictionary.
2. Unordered: Dictionaries are unordered, meaning that the elements in a dictionary are not stored in any particular order. This is because the keys are used to access the values, so the order in which they are stored is not important.
3. Mutable: Dictionaries are mutable, meaning that you can add, remove, and modify elements in a dictionary. You can add a new key-value pair, delete an existing key-value pair, or update the value associated with a key.
4. Efficient: Dictionaries are highly efficient when it comes to searching for elements. They use a hash table to store the key-value pairs, which allows for fast access to elements in the dictionary, even if the dictionary contains a large number of elements.

Tic Tac toe

```python
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ',
            'mid-L': ' ', 'mid-M': ' ', 'mid-R': ' ',
            'low-L': ' ', 'low-M': ' ', 'low-R': ' '}
```

1. A dictionary can be used to represent the state of a tic-tac-toe board.
2. The keys of the dictionary can represent the different positions on the board, and the values can represent what is in each position (e.g., 'X', 'O', or ' ').
3. TheBoard variable is used to store the dictionary that represents the initial empty state of the tic-tac-toe board.
4. Using a dictionary to represent the tic-tac-toe board allows for easy manipulation of the state of the board during gameplay.

Programming Code

Tranversing Dictionary Values

```
most_populous_city = ''
max_population = 0

for city, population in city_populations.items():
    if population > max_population:
        max_population = population
        most_populous_city = city

print(f"The most populous city is {most_populous_city} with a population of {max_population}.")
```

- A list is an ordered collection of elements that can be accessed by their index position, while a dictionary is an unordered collection of key-value pairs that can be accessed by their keys. In a list, the index of an element is an integer starting from `0`, whereas in a dictionary, the key can be any hashable data type.
- In this program, we first define a dictionary `city_populations` that contains city and population data. Then we initialize two variables `most_populous_city` and `max_population` to store the most populous city and its population, respectively. We then use a `for` loop to iterate over the key-value pairs in the dictionary using the `items()` method. For each pair, we compare the population with the current maximum population and update the `most_populous_city` and `max_population` variables if necessary.

```
sentence = input("Enter a sentence: ")

# Split the sentence into a list of words
words = sentence.split()

# Initialize variables to hold the longest word and its length
longest_word = ""
longest_word_length = 0

# Iterate over the words and update the longest word if needed
for word in words:
    if len(word) > longest_word_length:
        longest_word = word
        longest_word_length = len(word)

# Print the longest word and its length
print(f"The longest word is '{longest_word}' with a length of {longest_word_length} characters.")
```

1. The program prompts the user to enter a sentence using the `input()` function and stores the result in the `sentence` variable.
2. The program splits the sentence into a list of words using the `split()` method, which splits the string at whitespace characters by default and returns a list of substrings.
3. The program initializes two variables `longest_word` and `longest_word_length` to hold the longest word found so far and its length.
4. The program iterates over the words in the list using a `for` loop and compares the length of each word to the current longest word length. If the current word is longer, the program updates the `longest_word` and `longest_word_length` variables with the current word and its length.
5. The program prints out the longest word and its length using a formatted string with the `print()` function.