Python Class Notes

Clarusway

Collection Data Types

Nice to have VSCode Extentions:

- Jupyter

Needs

- Python (for Windows OS: add to the path while installing!)

Summary

- List
- Tuples
- Dictionary
- Set
    - Set vs List and Tuple

Data Structure	Ordered	Mutable	Constructor	Example
List	Yes	Yes	`[ ]` or `list()`	`[5.7, 4, 'yes', 5.7]`
Tuple	Yes	No	`( )` or `tuple()`	`(5.7, 4, 'yes', 5.7)`
Set	No	Yes	`{}`* or `set()`	`{5.7, 4, 'yes'}`
Dictionary	No	Yes**	`{ }` or `dict()`	`{'Jun': 75, 'Jul': 89}`

List

Lists are one of the most common and versatile collection data types in Python. They are ordered, mutable, and can hold any type of data.

```
list1 = ["Apple", 1, 3.5, True, [1, 3], {"Ready": "yes"}]
```

Lists may be constructed in several ways:

- Using a pair of square brackets to denote the empty list: []
- Using the type constructor: list() or list(iterable)

```
a = list()
print(a)
```

The constructor builds a list whose items are the same and in the same order as iterable's items.

- list('abc') returns ['a', 'b', 'c']
- list((1, 2, 3)) returns [1, 2, 3]

Python has a set of built-in methods that you can use on lists:

```
append()  # Adds a -single element- at the end of the list
```

```
stack = []

for i in range(5):
    stack.append(i)
```

```
    print(stack)

while len(stack):
    print(stack.pop())
```

```
clear()   # Removes all the elements from the list
copy()    # Returns a copy of the list
```

- The copy() method in Python returns a copy of the List. We can copy a list to another list using the = operator, however copying a list using = operator means that when we change the new list the copied list will also be changed, if you do not want this behaviour then use the copy() method instead of = operator.

```
count()   # Returns the number of elements with the specified value
extend()  # Add the elements of a list (or any iterable), to the end of the
current list
index()   # Returns the index of the first element with the specified value
insert(index, elem)  # Adds an element at the specified position
pop(index)    # Removes the element at the specified position, assigning the
popped item to a new variable
remove()  # Removes the item with the specified value (the first value)
reverse() # Reverses the order of the list
```

reverse() actually reverses the elements in the container. reversed() doesn't actually reverse anything, it merely returns an object that can be used to iterate over the container's elements in reverse order.

```
sort()    # Sorts the list
```

How to reach a value inside a list:

```
# Reach to the index no 1 inside inner list:
a = ['ali', 2, [1,2], 7]
a[2][1]
2
```

How to join two list:

```
list1 = ["a", "b", "c"]
list2 = [1, 2, 3]

list3 = list1 + list2
print(list3)
```

```
# ['a', 'b', 'c', 1, 2, 3]
```

Tuples

Tuples are similar to lists in many ways, but they are immutable, which means they cannot be modified once created. They are often used to represent fixed collections of related data, like a point in a coordinate system, choices of a pizza object like Medium, Small, and Large or a date and time.

- Less memory
- Stable, unchanged values
- Faster

Tuples may be constructed in a number of ways:

- Using a pair of parentheses to denote the empty tuple: ()
- Using a trailing comma for a singleton tuple: "a", or ("a",)
- Separating items with commas: "a", "b", "c" or ("a", "b", "c")
- Using the tuple() built-in: tuple() or tuple(iterable)

Tuple items are ordered, unchangeable, and allow duplicate values.

To create a tuple with only one item, you have to add a comma after the item:

```
thistuple = ("apple",)
print(type(thistuple))

# NOT a tuple
thistuple = ("apple")
print(type(thistuple))

# A tuple with strings, integers and boolean values:
tuple1 = ("abc", 34, True, 40, "male")

# Print the second item in the tuple:
thistuple = ("apple", "banana", "cherry")
print(thistuple[1])
```

Python has two built-in methods that you can use on tuples.

```
count() # Returns the number of times a specified value occurs in a tuple
index() # Searches the tuple for a specified value and returns the position of
where it was found
```

```
# Join two tuples:
tuple1 = ("a", "b" , "c")
```

```
tuple2 = (1, 2, 3)

tuple3 = tuple1 + tuple2
print(tuple3)

# ('a', 'b', 'c', 1, 2, 3)
```

Dictionary

Dictionaries are a powerful collection data type in Python that allow you to store data in key-value pairs. They are often used to represent real-world entities like people, products, or countries.

A dictionary is a collection which is ordered, changeable and does not allow duplicates.

Dictionaries are written with curly brackets, and have keys and values:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict)
# {'brand': 'Ford', 'model': 'Mustang', 'year': 1964}

# You can access the items of a dictionary by referring to its key name, inside
square brackets:
x = thisdict["model"]
print(x)
# Mustang

# If there is not such a key, it will raise key error. To avoid this use get()
method.


# There is also a method called get() that will give you the same result:
x = thisdict.get("model")
print(x)
# Mustang

# If not, then it will return None (if get() is used with only one argument).
# Default None can be changed to a worning message:
thisdict.get("bran", "There is no such a key!")


# The keys() method will return a list of all the keys in the dictionary.
x = thisdict.keys()
print(x)
# dict_keys(['brand', 'model', 'year'])
# Keys can be displayed on a loop:
for key in  thisdict.keys():
  print(key)
```

```
# Add a new item to the original dictionary, and see that the keys list gets
updated as well:
thisdict["color"] = "white"


# You can change the value of a specific item by referring to its key name:
thisdict["year"] = 2022

# The update() method will update the dictionary with the items from the given
argument. The argument must be a dictionary, or an iterable object with key:value
pairs.
thisdict.update({"year": 2020})

# Adding an item to the dictionary is done by using a new index key and assigning
a value to it:
thisdict["color"] = "red"

# The pop() method removes the item with the specified key name:
thisdict.pop("model")

# The popitem() method removes the last inserted item
thisdict.popitem()


# The del keyword removes the item with the specified key name:
del thisdict["model"]

# The del keyword can also delete the dictionary completely.

# The clear() method empties the dictionary.
```

- Reaching list item is the same with lists

Dictionary Methods

Python has a set of built-in methods that you can use on dictionaries.

```
clear() # Removes all the elements from the dictionary
get()   # Returns the value of the specified key
items() # Returns a list containing a tuple for each key value pair
keys()  # Returns a list containing the dictionary's keys
pop()   # Removes the element with the specified key
popitem()   # Removes the last inserted key-value pair
update()    # Updates the dictionary with the specified key-value pairs
values()    # Returns a list of all the values in the dictionary
```

Set

Sets are another collection data type in Python that can hold any type of data, but they are both unordered and unindexed, and contain only unique elements.

Sets are written with curly brackets. You cannot access items in a set by referring to an index or a key.

It is possible to use the set() constructor to make a set.

```
# Create a Set
thisset = {"apple", "banana", "cherry"}

# Get the number of items in a set:
print(len(thisset))

# To add one item to a set use the add() method.
thisset.add("orange")

# To remove an item in a set, use the remove(), or the discard() method.
thisset.remove("banana")
thisset.discard("banana")

# If the item to remove does not exist, discard() will NOT raise an error.

# Remove the last item by using the pop() method:

# Sets are unordered, so when using the pop() method, you do not know which item
that gets removed.
```

Python has a set of built-in methods that you can use on sets.

```
add()   # Adds an element to the set
clear() # Removes all the elements from the set
copy()  # Returns a copy of the set
difference()    # Returns a set containing the difference between two or more sets
difference_update() # Removes the items in this set that are also included in
another, specified set
discard()   # Remove the specified item
intersection()  # Returns a set, that is the intersection of two other sets
intersection_update()   # Removes the items in this set that are not present in
other, specified set(s)
isdisjoint()    # Returns whether two sets have a intersection or not
issubset()  # Returns whether another set contains this set or not
issuperset()    # Returns whether this set contains another set or not
pop()   # Removes an element from the set
remove()    # Removes the specified element
symmetric_difference()  # Returns a set with the symmetric differences of two sets
symmetric_difference_update()   # inserts the symmetric differences from this set
and another
union() # Return a set containing the union of sets
update()    # Update the set with the union of this set and others
```

```
a = {1, 2, 3, 10, 32, 100}
b = {1, 2, 32}

a.difference(b)
a.intersection(b)
a.union(b)
```

Set vs List and Tuple

Sets, unlike lists or tuples, cannot have multiple occurrences of the same element and store unordered values.

Usecase: Remove duplicate values from a list or tuple and to perform common math operations like unions and intersections.

☺ **Happy Coding!** ✍

Clarusway