5. Collections

More Data Types

(Additional Resources)

ICTAcademy

INFORMATION AND COMMUNICATIONS TECHNOLOGY

Recap/1

COLLECTIONS:

- Lists [1, 2, 1, 3, “a”] — insertion order
- Sets {1, 3, 2, “a”} — unordered
- Dictionary {“a”: 1, “b”: 2} — key -> value

INDEXING:

mylist[0] -> first element

mylist[-1] -> last element

mydictionary[“key”] -> value associated with the specified key

NOTE:

Accessing elements in a set requires looping through the elements.

Recap/2

MODIFYING:

mylist.append(“new”) -> adds “new” element

mylist.remove(“new”) -> removes “new” element

mylist.pop(4) -> removes element associated with the index 4

mylist.pop() -> removes the last lement of mylist

myset.add(“new”) -> adds “new” element

myset.remove(“new”) -> removes “new” element

myset.pop() -> removes the first element

mydictionary[“new”] = value -> adds or updates the value associated with the key “new”

mydictionary.pop(“new”) -> removes the specified key and its associated value

Additional resources: Set update

set.update(iterable): adds all elements from the specified iterable (e.g., list, tuple, set, or dictionary) to the set, effectively updating the original set with new elements.

firstset: set[int] = {1, 5, 7, 8}

secondset: set[int] = {2, 3, 7, 6}

firstset.update(secondset) results in:

firstset is {1, 2, 3, 5, 6, 7, 8}

This method modifies the original set in place, unlike the union() function, which instead returns a new set containing all the elements of the union.

Additional resources: Exploring Sets

Sets in Python are unordered collections of unique elements. To access or iterate through the elements of a set, you must use a loop, typically a for loop, since sets do not support indexing or slicing due to their unordered nature.

myset: set[int] = {5, 4, 8, 10}

```
for value in myset:
    print(value)
```

results in:

5

4

8

10

Addictional resources: Set/1

SET:

```
myset={1, "Hello", 3, 0.5, True, False,2}
print(myset)
Output: {False, 0.5, 1, 2, 3, 'Hello'}
```

- In Python, **sets** are unordered collections of unique elements, so they do not guarantee a fixed order. However, the output of `print(myset)` appears to follow some kind of "order," and we can analyze the reason.
- Although sets are officially unordered, Python internally implements them as **hash tables**, and **the order** in which the elements appear in the output **depends** on **hashing** and **how the elements are stored in memory**.
- **Hash tables** are a data structure used to store and quickly retrieve values associated with keys.
- Python uses hash tables to implement sets and dictionaries.
- Python uses hash functions to determine the position of an element in a set. Therefore, the order of a set is not guaranteed, as it depends on the internal hash table.

Addictional resources: Set/2

SET:

```
myset={1, "Hello", 3, 0.5, True, False,2}
print(myset)
Output: {False, 0.5, 1, 2, 3, 'Hello'}
```

The elements are stored based on their internal hash function, which depends on the data type and how Python manages hashes to optimize search operations.

- In Python, **False** is internally represented as **0** and **True** as **1**. So, when we add **True** and **1** to the same set, the set considers them duplicates and keeps only **1** or **True** , depending on which is stored first. A similar reasoning applies to the values **0** and **False** when both are present in the same set.
- **''Hello''** is a string and is positioned separately.
- **3** is an integer, so it is another distinct integer.
- **0.5** is a distinct float value, so it's retained.
- **2** is an integer, so it is another distinct integer.

Addictional resources: Set/3

SET:

```
myset={"Hello", 3, 0.5, True, False,2}
print(myset)
Output: {0.5, True, 2, 3, False, 'Hello’}
```

- Removing **1** from myset and leaving **True** only, in output True will be retained because it is interpreted as **1**.

```
myset={1, "Hello", 3, 0.5, False,2}
print(myset)
Output: {0.5, 1, 2, 3, False, 'Hello’}
```

- Removing **True** from myset and leaving **1** only, in output **1** will be retained because it is interpreted as **1** (integer value), and it’s different from the other integer values in myset.

Additional resources: Exploring Dictionaries/1

dictionary.items(): returns a view of the dictionary's items (key-value pairs) as tuples. This view reflects changes to the dictionary, providing a dynamic and direct way to iterate over both keys and values.

mydict: dict[str, int] = {“a”: 5, “b”: 2}

tuple

```
for key, value in mydict.items():
    print(key, value)
```

results in:

a 5

b 2

Additional resources: Exploring Dictionaries/2

dictionary.values(): returns a view of the dictionary's values. This view is dynamic and reflects changes to the dictionary, allowing for efficient iteration over values.

mydict: dict[str, int] = {“a”: 5, “b”: 2}

```
for value in mydict.values():
    print(value)
```

results in:

5

2

Additional resources: Exploring Dictionaries/3

dictionary.keys(): returns a view of the dictionary's keys. This view is dynamic and reflects changes to the dictionary, allowing for efficient iteration over keys.

mydict: dict[str, int] = {"a": 5, "b": 2}

```
for key in mydict.keys():
    print(key)
```

results in:

a

b

Additional resources: List extension/1

list.extend(iterable): adds all the elements from the specified iterable (e.g., list, tuple, set, or dictionary) to the end of the list, extending the list.

firstlst: list[int] = [1, 5, 7, 8]

secondlst: list[int] = [2, 3, 7, 6]

1st METHOD (this method modifies the original list in place):

- **firstlst.extend(secondlst)** results in:

firstlst is [1, 5, 7, 8, 2, 3, 7, 6]

2nd METHOD (this method creates a new extended list):

- **thirdlst: list[int] = firstlst + secondlst** results in:

thirdlst is [1, 5, 7, 8, 2, 3, 7, 6]

Additional resources: List extension/2

list.extend(iterable): adds all the elements from the specified iterable (e.g., list, tuple, set, or dictionary) to the end of the list, extending the list.

firstlst: list[int] = [1, 2, 3, 4, 5]

- **firstlst.extend((8, 7, 6))** results in:

firstlst is [1, 2, 3, 4, 5, 8, 7, 6]

- **firstlst.extend({8, 7, 6})** results in:

firstlst is [1, 2, 3, 4, 5, 8, 7, 6]

Additional resources: List extension/3

list.extend(iterable): adds all the elements from the specified iterable (e.g., list, tuple, set, or dictionary) to the end of the list, extending the list.

firstlst: list[int] = [1, 2, 3, 4, 5]

- **firstlst.extend({"key": "value", "Bool": True})** results in:

firstlst is [1, 2, 3, 4, 5, ‘key’, ‘Bool’]

The **extend(iterable)** method takes an iterable and adds each element of the iterable separately to the original list.

When you pass a dictionary to **extend()** function, Python treats it as an iterable, and dictionaries, when iterated, return only their keys (and not the associated values).

Additional resources: List extension/4

list.extend(iterable): adds all the elements from the specified iterable (e.g., list, tuple, set, or dictionary) to the end of the list, extending the list.

firstlst: list[int] = [1, 2, 3, 4, 5]

- **firstlst.extend({"key": "value", "Bool": True}.values())** results in:

firstlst is [1, 2, 3, 4, 5, ‘value’, True]

It’s possible to add dictionary values using the **values()** function.