# Python Unordered Data Structures and Their Built-in Methods

Welcome to today's lecture on Python unordered data structures and their extensive built-in methods. Sets and dictionaries are fundamental data structures in Python, each serving unique purposes. In this lesson, we will explore these data structures and delve into the numerous built-in methods that make them powerful for data manipulation.

## Table of Contents



---

## Sets

- Python: Sets

Sets are mutable, unordered collections of unique elements. Because the are unordered, sets are not cosidered sequences, like lists, and cannot support indexing or slicing. They do, however, support mathematical operations like union, difference, and symmetric difference. Sets are used for membership testing and deduplication.

### Creating Sets

Empty sets can only be instantiated with the `set()` built-in method. If you want to create a set with elements, you will wrap the elements with curly braces `{}`. Please note that if you define an empty set with empty curly braces, Python will interpret that as an empty dictionary.

In [ ]:
```
my_set = set()
my_set
```

Out[ ]:
```
set()
```

In [ ]:
```
my_set = {1,2,3,4,5}
my_set
```

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

One of the unique properties of sets is that they automatically deduplicate values. Set elements must be unique, so duplicate values are dropped if they exist more than once upon set creation. This property highlights the fact that sets are unordered collections.

In [ ]:
```
my_set = {1,1,2,'a',3,4,'b',1,5,7,'a',6,3,5,1,'y',1,1,1}
my_set
```

Out[ ]:
```
{1, 2, 3, 4, 5, 6, 7, 'a', 'b', 'y'}
```

We know that each value is unique in this set, but which item of the duplicates were actually included?

### Set Methods

To add elements to a set, we use the `.add()` method. To remove individual elements from a set, use the `.remove()` method.

| Syntax | Description |
|---|---|
| `set_variable.add(elem)` | Adds `elem` to the set |
| `set_variable.remove(elem)` | Removes `elem` from the set |
| `set_variable.pop()` | Removes and returns a random item from the set |

In [ ]:
```
my_set.add(500)
my_set
```

Out[ ]:
```
{1, 2, 3, 4, 5, 500, 6, 7, 'a', 'b', 'y'}
```

In [ ]:
```
my_set.remove(500)
my_set
```

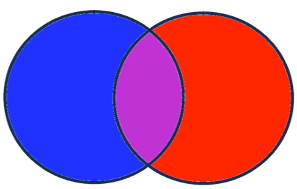
Out[ ]:
```
{1, 2, 3, 4, 5, 6, 7, 'a', 'b', 'y'}
```

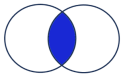
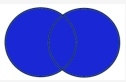
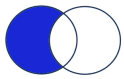
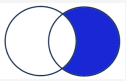
The `pop()` command removed and returns an arbitrary item from the set. You cannot specify an index with `pop()` because sets are unordered.

In [ ]:
```
item = my_set.pop()
print(item, my_set)
```

```
1 {2, 3, 4, 'b', 'a', 5, 7, 6, 'y'}
```

Python sets support specific mathematical operations. These concepts can be illustrated with venn diagrams you've probably seen in math class.



| Code | Mathematical Operation | Description | Graphical Representation |
|---|---|---|---|
| `a.intersection(b)` | Intersection | Returns elements that appear in both set `a` and set `b` |  |
| `a.union(b)` | Union | Returns elements in either set `a` or set `b` |  |
| `a.difference(b)` | Left Set Difference | Returns elements from set `a` that are not in set `b` |  |
| `b.difference(a)` | Right Set Difference | Returns elements from set `b` that are not in set `a` |  |
| `a.symmetric_difference(b)` | Symmetric Set Difference | Returns elements in set `a` or set `b`, but not in both |  |

For example, lets take a look at a set of employees. Who still needs to complete their annual training? We could take the difference between the two sets and find out.

In [ ]:
```
employees = {'Maxim', 'Monica', 'Martez', 'Ala',
             'Cammie', 'Amari', 'Toccara', 'Kassandra',
             'Rosalinda', 'Siddie', 'Luigi', 'Pepper',
             'Damaris', 'Harrell', 'Earl', 'Antoine',
             'Marlene', 'Johnathan', 'Ivonne', 'Jon',
             'Whitney', 'Heidy', 'Terra', 'Jamaal',
             'Jeanie', 'Kyrie', 'Sheila', 'Gunda',
             'Gretta', 'Cornell', 'Lizzie', 'Romona',
             'Tyrek', 'Vilma', 'Adison', 'Saniya',
             'Manda', 'Talan', 'Loring', 'Taraji',
             'Tracy', 'Felicie', 'Jess', 'Coraima',
             'Aliza', 'Oris', 'Edwina', 'Easter',
             'Haskell', 'Mandie', 'Garret', 'Skylar',
             'Lindsey', 'Gurney', 'Kanisha', 'Webb',
             'Boss', 'Shreya', 'Arron', 'Earley',
             'Linnea', 'Page', 'Elouise', 'Shirleen',
             'Nanna', 'Callie', 'Tammy', 'Theodocia',
             'Kiley', 'Eddy', 'Ida', 'Hampton', 'Jaheem',
             'Aja', 'Carlyn'}

completed_training = {'Gunda', 'Jess', 'Page', 'Siddie', 'Kassandra', 'Theodocia', 'Ida', 'Heidy', 'Monica', 'Aliza', 'Cammie', 'Carlyn', 'Taraji'}
```

In [ ]:
```
employees.difference(completed_training)
```

Out[ ]:
```
{'Adison',
 'Aja',
 'Ala',
 'Amari',
 'Antoine',
 'Arron',
 'Boss',
 'Callie',
 'Coraima',
 'Cornell',
 'Damaris',
 'Earl',
 'Earley',
 'Easter',
 'Eddy',
 'Edwina',
 'Elouise',
 'Felicie',
 'Garret',
 'Gretta',
 'Gurney',
 'Hampton',
 'Harrell',
 'Haskell',
 'Ivonne',
 'Jaheem',
 'Jamaal',
 'Jeanie',
 'Johnathan',
 'Jon',
 'Kanisha',
 'Kiley',
 'Kyrie',
 'Lindsey',
 'Linnea',
 'Lizzie',
 'Loring',
 'Luigi',
 'Manda',
 'Mandie',
 'Marlene',
 'Martez',
 'Maxim',
 'Nanna',
 'Oris',
 'Pepper',
 'Romona',
 'Rosalinda',
 'Saniya',
 'Sheila',
 'Shirleen',
 'Shreya',
 'Skylar',
 'Talan',
 'Tammy',
 'Terra',
 'Toccara',
 'Tracy',
 'Tyrek',
 'Vilma',
 'Webb',
 'Whitney'}
```

---

## Dictionaries

- Python: Dictionaries

Dictionaries are mutable, unordered collections of key-value pairs. Unlike sequences, which are indexed, dictionaries are accessible by their keys, which can be of any immutable type. Strings, numbers, and tuples can all serve as keys in a dictionary. The values can be any data type (even other dictionaries or lists).

Dictionaries share many properties with sets:

- Both sets and dictionaries are unordered
- Both sets and dictionaries are wrapped in curly braces
- Sets must have unique entries, dictionaries must have unique keys

A driver's license is a great example of a common object in our daily lives that uses key-value pairs. Each data point is linked to a key that describes what it is.


### Creating Dictionaries

Like sets, dictionaries use curly braces as wrappers. However, a pair of empty curly braces gets interpreted as an empty dictionary, not a set. We can also set up a dictionary that's populated with entries. Note that each entry in the dictionary below is a pair of values separated by a colon ( `:` ). To the left of the colon is the entry's *key*, and to the right is the *value*. Dictionary entries are always structured like this: `key: value` , with commas separating each `key: value` pair.

In [ ]:
```
empty_dictionary = {}
empty_dictionary
```

Out[ ]:
```
{}
```

In [ ]:
```
english_french_dictionary = {'hello': 'bonjour',
                             'world': 'monde',
                             'thank you': 'merci'}
english_french_dictionary
```

Out[ ]:
```
{'hello': 'bonjour', 'world': 'monde', 'thank you': 'merci'}
```

You can access a dictionary's values using its keys. Like with python lists or tuples, Python usus square brackets in the syntax but index is not used to access the data inside a dictionary. Instead, the key is passed in. Since dictionaries are mutable, we can also add and reassign a value using the key.

| Syntax | Description |
|---|---|
| `dict_variable[key]` | Accesses the item at key `[key]` |
| `dict_variable[key] = "what I want"` | Sets the item at key `[key]` to `"what I want` |

> NOTE: Dictionary keys can be any immutable data type.

In [ ]:
```
english_french_dictionary["hello"]
```

Out[ ]:
```
'bonjour'
```

Here we use an assignment operator to create a new entry into the dictionary

In [ ]:
```
english_french_dictionary['good bye'] = 'au revoir'
english_french_dictionary
```

Out[ ]:
```
{'hello': 'bonjour',
 'world': 'monde',
 'thank you': 'merci',
 'good bye': 'au revoir'}
```

We can overwrite an existing value in a dictionary as well.

In [ ]:
```
english_french_dictionary['hello'] = ['bonjour', 'salut']
english_french_dictionary
```

Out[ ]:
```
{'hello': ['bonjour', 'salut'],
 'world': 'monde',
 'thank you': 'merci',
 'good bye': 'au revoir'}
```

We can delete a key/value pair from a dictionary using the `del` keyword. As with all operations with Python, there is not confirmation and the entry is completely erased from memory.

In [ ]:
```
del english_french_dictionary["world"]
english_french_dictionary
```

Out[ ]:
```
{'hello': ['bonjour', 'salut'], 'thank you': 'merci', 'good bye': 'au revoir'}
```

The `keys()` and `values()` methods are used to see just the keys or values of a dictionary (we can use these methods for membership testing). These methods return *list-like* objects.

In [ ]:
```
english_french_dictionary.keys()
```

Out[ ]:
```
dict_keys(['hello', 'thank you', 'good bye'])
```

In [ ]:
```
english_french_dictionary.values()
```

Out[ ]:
```
dict_values([['bonjour', 'salut'], 'merci', 'au revoir'])
```

We can see the leys and values together in a list-list object of tuples with the `items()` method (we will use these methods later when we go over iteration)

In [ ]:
```
english_french_dictionary.items()
```

Out[ ]:
```
dict_items([('hello', ['bonjour', 'salut']), ('thank you', 'merci'), ('good bye', 'au revoir')])
```

### Dictionary Methods

| Syntax | Description |
|---|---|
| `dict.get(key, default)` | Get the value associated with a key, or a default value if the key is not present. |
| `dict.pop(key, default)` | Remove the item with a specific key and return its value, or a default value if the key is not present. |
| `dict.popitem()` | Remove and return an arbitrary key-value pair. |
| `dict.update(other_dict)` | Update the dictionary with key-value pairs from another dictionary. |
| `dict.clear()` | Remove all items from the dictionary. |

In [ ]:
```
english_french_dictionary.clear()
english_french_dictionary
```

Out[ ]:
```
{}
```

In [ ]:
```
new_english_french_dictionary = {
    "hello": "bonjour",
    "goodbye": "au revoir",
    "bathroom": "salle de bain",
    "library": "bibliothèque",
    "cookie": "biscuit",
    "milk": "lait",
    "green": "vert",
    "chicken": "poulet"
}
```

In [ ]:
```
english_french_dictionary.update(new_english_french_dictionary)
english_french_dictionary
```

Out[ ]:
```
{'hello': 'bonjour',
 'goodbye': 'au revoir',
 'bathroom': 'salle de bain',
 'library': 'bibliothèque',
 'cookie': 'biscuit',
 'milk': 'lait',
 'green': 'vert',
 'chicken': 'poulet'}
```

In [ ]:
```
english_french_dictionary.pop("hello")
english_french_dictionary
```

Out[ ]:
```
{'goodbye': 'au revoir',
 'bathroom': 'salle de bain',
 'library': 'bibliothèque',
 'cookie': 'biscuit',
 'milk': 'lait',
 'green': 'vert',
 'chicken': 'poulet'}
```

In [ ]:
```
english_french_dictionary.get('hello', "Not Found")
```

Out[ ]:
```
'Not Found'
```

In [ ]:
```
english_french_dictionary.popitem()
```

Out[ ]:
```
('chicken', 'poulet')
```

## Common Operations Across Data Structures

The functions in the table below also apply to sets and dictionaries. When called on dictionaries, these functions consider only the keys. Because of this, calling `max()` on a dictionary would give you its largest (highest value) key. Keep in mind also that `sorted()` will always return a list, even if you call it on a set or dictionary.

| Syntax | Description |
|---|---|
| `max(collection)` | Returns the maximum value contained in the collection |
| `min(collection)` | Returns the minimum value contained in the collection |
| `sum(collection)` | Returns the sum of the collection's elements (only works if all items are numeric) |
| `len(collection)` | Returns the number of elements in the collection |
| `sorted(collection)` | Returns a sorted list of the collections elements |

In [ ]:
```
my_set = {'F15', 'P17', 'T18', 'E00', 'A18', 'D13', 'D19', 'Z16', 'K17', 'X16', 'V13', 'U13'}

max(my_set)
```

Out[ ]:
```
'Z16'
```

In [ ]:
```
my_set = {'F15', 'P17', 'T18', 'E00', 'A18', 'D13', 'D19', 'Z16', 'K17', 'X16', 'V13', 'U13'}

min(my_set)
```

Out[ ]:
```
'A18'
```

In [ ]:
```
my_list = [45, 42, 57, 55, 46, 60, 50, 44, 40, 43, 52, 54]

sum(my_list)
```

Out[ ]:
```
588
```

In [ ]:
```
my_dict = {'A': 0, 'B': 1, 'C': 2, 'D': 3, 'E': 4, 'F': 5, 'G': 6, 'H': 7, 'I': 8, 'J': 9}

len(my_dict)
```

Out[ ]:
```
10
```

In [ ]:
```
my_dict = {'A': 0, 'B': 1, 'C': 2, 'D': 3, 'E': 4, 'F': 5, 'G': 6, 'H': 7, 'I': 8, 'J': 9}

sorted(my_dict, reverse=True)
```

Out[ ]:
```
['J', 'I', 'H', 'G', 'F', 'E', 'D', 'C', 'B', 'A']
```

## Conclusion

In this lecture, you've explored the power of dictionaries and sets in Python, along with their built-in methods. Dictionaries provide a flexible way to manage data using key-value pairs, while sets offer efficient storage for unique elements. Understanding when and how to use these data structures, as well as their methods, is essential for effective Python programming.

Continue to practice and apply these concepts in your Python projects to become proficient in working with dictionaries and sets.

This lecture comprehensively covers the built-in methods of dictionaries and sets in Python, providing explanations and code examples for each method.

## Assignments

- Number of Characters
- Car Trip
- How much will you spend
- Employee Data