

Digital Career Institute

Python Course - Collections



Sets

Collections in Python

Sets distinctive feature is that they do not allow duplicate values.

They:

- Have an **no order**.
- **Do not store duplicate** values.
- **Do not allow** their objects to be **changed**.
- Allow objects of **different types**.

Defining Sets

They are defined using curly brackets `{}`.



```
>>> fruits = {  
...     "Apple",  
...     "Orange",  
...     "Pear",  
...     "Banana",  
...     "Apricot"  
... }
```

Any **iterable** can be converted into a set by using the `set` constructor.



```
>>> hello = set("hello")  
>>> print(hello)  
{'o', 'e', 'h', 'l'}  
>>> empty_set = set()  
>>> print(empty_set)  
set()
```

The `set` constructor can also be used to create empty sets if no argument is given.



Defining Sets

Sets can contain values of any type. Repeated values can be added, but they will only be stored once.

Items in the set can be of mixed types.

But the items themselves can not be sets.

A set can also be created using the `copy` method of another set.



```
>>> fruits = {"Apple", "Apple"}
>>> letters = set("hello")
>>> ages = {32, 45, 42, 12, 34, 57}
>>> dates = {datetime, datetime}
>>> data = {"John", 32, datetime}
>>> sets = {
...     {"John", "Mary", "Amy"},
...     {32, 43, 51}
... }
TypeError: unhashable type: 'set'
>>> copy = data.copy()
```

Python Sets: Accessing Values

DLI

Printing the set will show its values in a **different order** than the one used when the set was defined.



```
>>> print(fruits)
{'Apricot', 'Banana', 'Pear', ...}
```

The values in a set cannot be accessed directly using indexing.



```
>>> print(fruits[0])
Traceback (most recent call
last):
  File "<stdin>", line 1, in
<module>
TypeError: 'set' object does not
support indexing
```

Therefore, there is no **index** method. Because the set has no order, it has no method **sort** or **reverse**. And because the set has no repeated values, it does not have the **count** method either.

Python Sets: Adding Values

Adding new values can be done with the method **add**. The new value may show anywhere, as there is no order.



The **update** method can be used to add multiple values at once. It accepts any iterable as an argument.



Adding a value twice throws no error, but only stores the value once.



```
>>> fruits.add('Pineapple')
>>> print(fruits)
{'Apricot', 'Pineapple', ...}

>>> fruits.update(
...     ['Mango', 'Mango']
... )
>>> fruits
{'Apricot', 'Pineapple',
 'Banana', 'Mango', 'Pear',
 'Apple', 'Orange'}
```

Python Sets: Removing Values

DLI

The method **pop** will remove a random element and will return it. It accepts no argument.



```
>>> random = fruits.pop()
>>> print(random)
```

Apricot

The **discard** method removes the given value and returns nothing.



```
>>> print(fruits.discard('Mango'))
```

None

```
>>> print(fruits)
```

```
{'Pineapple', 'Banana', 'Pear',
 'Apple', 'Orange'}
```

```
>>> fruits.remove('Mango')
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

KeyError: 'Mango'

```
>>> fruits.clear()
```

```
>>> print(fruits)
```

set()

The **remove** method is like **discard** but it throws an error exception if the value does not exist in the set.



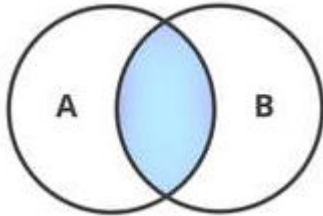
The **clear** method will remove all items in the set.



Python Sets: Set Operations

DLI

The set type includes methods to perform the standard operations between sets of Set Theory and return new sets.



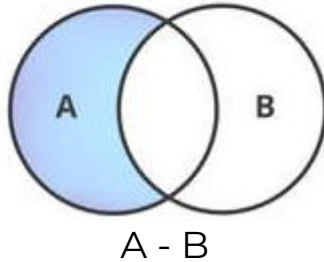
A and B

The method **intersection** returns a set containing the values present in both sets. The original sets remain unchanged.

```
>>> fruits = {  
...     "Pear",  
...     "Mango",  
...     "Apple",  
...     "Strawberry"  
... }  
>>> smoothie = {  
...     "Mango",  
...     "Orange",  
...     "Pear"  
... }  
>>> fruits.intersection(smoothie)  
{ 'Mango', 'Pear' }
```

Python Sets: Set Operations

DLI

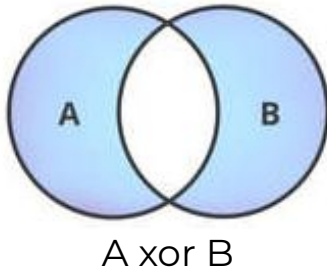


The method **difference** returns a set containing the values present in A that don't exist in B. The original sets remain unchanged.

```
>>> fruits = {  
...     "Pear",  
...     "Mango",  
...     "Apple",  
...     "Strawberry"  
... }  
>>> smoothie = {  
...     "Mango",  
...     "Orange",  
...     "Pear"  
... }  
>>> fruits.difference(smoothie)  
{'Strawberry', 'Apple'}
```

Python Sets: Set Operations

DLI



The method `symmetric_difference` returns a set containing the values present in A or B that don't exist at the same time in both. The original sets remain unchanged.

```
>>> fruits = {  
...     "Pear",  
...     "Mango",  
...     "Apple",  
...     "Strawberry"  
... }  
>>> smoothie = {  
...     "Mango",  
...     "Orange",  
...     "Pear"  
... }  
>>> fruits.symmetric_difference(smoothie)  
{'Orange', 'Strawberry', 'Apple'}
```

Python Sets: Set Operations

The `difference_update`, `intersection_update` and `symmetric_difference_update` methods will calculate the same thing as do the `difference`, `intersection` and `symmetric_difference` methods.

The difference is that the first methods will overwrite the original set (fruits, in this case) instead of returning the result.

```
>>> fruits = {  
...     "Pear",  
...     "Mango",  
...     "Apple",  
...     "Strawberry"  
... }  
>>> smoothie = {  
...     "Mango",  
...     "Orange",  
...     "Pear"  
... }  
>>> fruits.difference_update(smoothie)  
>>> print(fruits)  
{'Strawberry', 'Apple'}
```

Python Sets: Set Operations

The method **union** returns a set containing the values present in A or B (or both). The original sets remain unchanged.

```
>>> fruits = {  
...     "Pear",  
...     "Mango",  
...     "Apple",  
...     "Strawberry"  
... }  
>>> smoothie = {  
...     "Mango",  
...     "Orange",  
...     "Pear"  
... }  
>>> fruits.union(smoothie)  
{'Strawberry', 'Orange', 'Mango', 'Pear',  
'Apple'}
```

Python Sets: Set Operations

DLI

The previous methods return a set with the result. Sometimes the script only requires to know if some kind of relationships exists between the sets, not the particular values.

The method `isdisjoint` returns `True` if the two sets have no item in common and `False` if they share at least one value.

```
>>> fruits = {  
...     "Pear",  
...     "Mango",  
...     "Apple",  
...     "Strawberry"  
... }  
>>> smoothie = {  
...     "Mango",  
...     "Orange",  
...     "Pear"  
... }  
>>> fruits.isdisjoint(smoothie)  
False
```

Python Sets: Set Operations

DLI

The method `issubset` returns `True` if the first set is completely contained in the set passed as argument.

The method `issuperset` returns `True` if the first set completely contains the set passed as argument.

```
>>> fruits = {  
...     "Pear",  
...     "Mango",  
...     "Apple",  
...     "Strawberry"  
... }  
>>> smoothie = {  
...     "Mango",  
...     "Pear"  
... }  
>>> fruits.issubset(smoothie)  
False  
>>> smoothie.issubset(fruits)  
True
```

Comparing Python Sets

DLI

Comparing two sets will return **True** if the following statement is true for the elements in it:

- They are the same

A set is very useful when we need to know if the items in two iterables are the same.

*Using **set()** on both iterables before comparing them will remove duplicates and ignore the order.*

```
>>> set1 = {1, 2, 3}
>>> set2 = {1, 2, 3}
>>> set1 == set2
True
>>> set1 is set2
False
>>> set1 == {3, 2, 1}
True
>>> set1 == {1, 1, 2, 2, 3, 3}
True
>>> list1 = [1, 2, 3]
>>> list2 = [3, 1, 2, 1, 3, 2]
>>> set(list1) == set(list2)
True
```


Python Sets: Use Case Examples

DLI

CITIES VISITED

- Berlin
- Barcelona
- Stockholm
- Trondheim
- Salzburg
- Brno
- Girona
- Manchester
- Ljubljana
- Tijuana

REGISTERED

- Mary
- John
- Eva
- Susie
- Peter
- Lucy
- Ronnie
- Gerald
- Anna
- Anthony

SPORTS

- Badminton
- Tennis
- Athletics
- Swimming
- Basketball
- Football
- Baseball
- Table-tennis
- Skiing
- Curling

Python Set Methods: Summary

DLI

Sets have a long number of methods:

Add

- Add
- Update

Remove

- Pop
- Remove
- Discard
- Clear

Manage

- Copy

Set Operations

- Intersection
- Difference
- ...

```
>>> print(dir(fruits))
[... , 'add', 'clear', 'copy',
'difference', 'difference_update',
'discard', 'intersection',
'intersection_update',
'isdisjoint', 'issubset',
'issuperset', 'pop', 'remove',
'symmetric_difference',
'symmetric_difference_update',
'union', 'update']
```

We learned ...

- That one of the most common type of collections in Python is the list. Lists have an implicit order according to the position of the value in the list.
- That lists can be manipulated to add, remove and changes its elements.
- That tuples are very similar to lists, but they are immutable. Their elements cannot be changed.
- That comparing between lists works the same as comparing between tuples. They are only the same if they have the same values and in the same order.
- That sets, as opposed to lists and tuples, do not allow repeated values and their values have no order.
- That two sets are the same if they have the same values, no matter the order in which they were added to the set.

Dictionaries

Collections in Python

Dictionaries are associative arrays.
They:

- Have an **order**.
- **Allow duplicate** values.
- **Allow** their objects to be **changed**.
- Allow objects of **different types**.

Defining Dictionaries

They are defined using curly brackets `{}`. Each key-value pair is separated by a comma and every key is separated by a colon from the value.



Some **iterables** can be converted into a dictionary by using the `dict` constructor.



The `dict` constructor can also be used to create empty tuples if no argument is given.



```
>>> address = {  
...     "name": "Harry Potter",  
...     "street": "Private Drive",  
...     "number": 4,  
...     "city": "Little Whinging",  
...     "county": "Surrey",  
... }  
>>> choice_dict = dict(choices)  
>>> print(choice_dict)  
{'Mon': 'Monday', 'Tue': 'Tuesday', ...}  
>>> empty_dict = dict()  
>>> print(empty_dict)  
( )
```

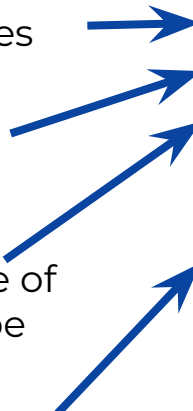
Defining Dictionaries

Dictionaries can contain values of any type.

Dictionary indices (keys) can also be of any type.

Items in the dictionary can be of mixed types and values can be repeated.

The items themselves can also be dictionaries.



```
>>> values = {"a": 1, "b": "string"}
>>> keys = {1: "one", date: "day"}
>>> repeated = {1: "hi", 2: "hi"}
>>> users = {
...     "john": {
...         "name": "John Doe",
...         "age": 30
...     },
...     "jane": {
...         "name": "Jane Doe",
...         "age": 40
...     }
... }
```

Defining Dictionaries

Dictionaries can also be initialized with the **fromkeys** class method.



This method takes a first argument as a sequence of keys and an optional second argument as the default value to initialize the values of each key.

Dictionaries can also be created by copying other dictionaries, using **copy**.



```
>>> template = dict.fromkeys(
...     ("street", "number", "zip",
...     "city", "country"),
...     "Unknown"
... )
>>> print(template)
{'street': 'Unknown', 'number': 'Unknown',
'zip': 'Unknown', 'city': 'Unknown',
'country': 'Unknown'}
>>> address = template.copy()
>>> address.update({"street": "Hogwarts"})
{'street': 'Hogwarts', 'number':
'Unknown', 'zip': 'Unknown', 'city':
'Unknown', 'country': 'Unknown'}
```


Python Dictionaries: Accessing Values


DLI

Printing the dictionary will show its values in the **same order** used when the list was defined.

!! This is only true in versions >3.6 of the Python interpreter. In Python 3.6 dictionaries do not have an order.

Each value in the dictionary can be accessed indexing its **key** or using the **get** method.

The **get** method accepts a second argument that will be used as default value if the given key does not exist.



```
>>> print(users)
{'john': {'name': 'John Doe', ...
>>> print(users["john"])
{'name': 'John Doe', 'age': 30}
>>> print(users.get("jane"))
{'name': 'Jane Doe', 'age': 40}
>>> print(users.get(
...     "mario",
...     {"name": "Unknown"}
... ))
{"name": "Unknown"}
```

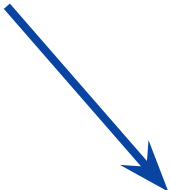

Python Dictionaries: Accessing Values

DLI

The `setdefault` method works similarly to the `get` method. It returns the value if the key exists, and if not it returns the default value given.

The difference is that if the given key does not exist in the dictionary, this key is created with the given default value.

If no default value is passed, it is created with the value `None`.



```
>>> print(users.setdefault(
...     "mario",
...     {"name": "Mario Sanz",
...      "age": 25}
... ))
{"name": "Mario Sanz", "age": 25}
>>> print(users)
{'john': {'name': 'John Doe',
'age': 30},
'jane': {'name': 'Jane Doe',
'age': 40},
'mario': {'name': 'Mario Sanz',
'age': 25}}
```

Python Dictionaries: Changing Values

The values in the dictionary can be changed by using the keys as indexes.

The `get` method can not be used to accomplish this.

```
>>> addr = {  
...     "name": "Harry Potter",  
...     "street": "Private Drive",  
...     "number": 4,  
...     "city": "Little Whinging",  
...     "county": "Surrey",  
... }  
>>> addr["street"] = "Hogwarts End"  
>>> print(addr["street"])  
Hogwarts End
```

Python Dictionaries: Adding Values

New values can be added to the dictionary the same way they are changed.

If the key does not exist, it will be created.

The **update** method can be used to merge a dictionary into another.

Values will be overwritten or created.

```
>>> addr = {  
...     "name": "Harry Potter",  
...     "street": "Private Street",  
...     "number": 4,  
...     "city": "Little Whinging"  
... }  
>>> addr["country"] = "UK"  
>>> print(addr["country"])  
UK  
>>> fix = {"country": "UK", "continent": "Europe"  
...       "street": "Hogwarts End"}  
>>> addr.update(fix)  
>>> print(addr)  
{'name': 'Harry Potter', 'street': 'Hogwarts End',  
'number': 4, 'city': 'Little Whinging', 'country':  
'UK', 'continent': 'Europe'}
```

Python Dictionaries: Removing Values

The `popitem` method removes and returns the last item in the dictionary. It does not accept any argument.

The `pop` method removes the item with the given key and returns its value. The argument is required.

Notice the `popitem` method returns a tuple containing both the key and the value. The `pop` method only returns the value.

```
>>> addr = {  
...     "name": "Harry Potter",  
...     "street": "Private Drive",  
...     "number": 4,  
...     "city": "Little Whinging",  
...     "county": "Surrey",  
... }  
>>> addr.popitem()  
('county', 'Surrey')  
>>> addr.popitem()  
('city', 'Little Whinging')  
>>> addr.pop("street")  
Private Drive
```

Python Dictionaries: Removing Values

The **clear** method removes all the items in the dictionary.

```
>>> addr = {  
...     "name": "Harry Potter",  
...     "street": "Private Drive",  
...     "number": 4,  
...     "city": "Little Whinging",  
...     "county": "Surrey",  
... }  
>>> addr.clear()  
>>> print(addr)  
{}
```

Python Dictionaries: Other Methods

Dictionaries have three additional methods that are used often.

The method **keys** returns an iterable with all the keys (and no values) of the dictionary.

The method **values** returns an iterable with all the values (and no keys) of the dictionary.

The method **items** returns an iterable with all the keys and values of the dictionary.


```
>>> addr = {  
...     "name": "Harry Potter",  
...     "street": "Hogwarts",  
...     "number": 4,  
...     "city": "Little Whinging"  
... }  
>>> addr.keys()  
dict_keys(['name', 'street', 'num...  
>>> addr.values()  
dict_values(['Harry Potter', 'Hog...  
>>> addr.items()  
dict_items([('name', 'Harry Potte...
```

Comparing Python Dictionaries

Comparing two dictionaries will return **True** if the following statements are true for the elements in it:

- They have the same keys
- For each key the value is the same

Notice the order of the keys in the dictionary is not considered.



```
>>> dict1 = {"a": 1, "b": 2}
>>> dict1 = {"a": 1, "b": 2}
>>> dict1 == dict2
True
>>> dict1 is dict2
False

>>> dict1 == {"b": 2, "a": 1}
True

>>> dict1 == {"a": 2, "b": 1}
False
```


Python Dictionaries: Use Case Examples

DLI

USER PROFILE

- First name
- Family name
- Date of birth
- City of residence
- Country of residence
- Sex
- Job title
- Company
- Interests

REGISTRATION

- Student
- Course
- Tutor
- Date of registration
- Passed (Yes/No)
- Date of finalization

ADDRESS

- Type of street
- Street name
- Street number
- Door number
- Postal code
- District
- City
- Country

Python Dictionary Methods: Summary

DLI

Dictionaries have the following methods:

Add & Create

- Setdefault
- Update
- Copy
- Fromkeys

Remove

- Pop
- Popitem
- Clear

Access

- Get

Other

- Items
- Keys
- Values

```
>>> print(dir(address))  
[..., 'clear', 'copy', 'fromkeys',  
'get', 'items', 'keys', 'pop',  
'popitem', 'setdefault', 'update',  
'values']
```

We learned ...

- That Python's associative arrays are called dictionaries.
- That dictionaries, as opposed to lists and tuples, use keys instead of indices to refer to each of the values inside them.
- That template dictionaries can be created with the **fromkeys** method using a custom default value.
- That dictionaries have specific methods to return different types of iterables: **keys**, **values** and **items**.
- That two dictionaries are considered the same if they have the same keys and values, even if they are in different order.