Python Unit 4-part 4

Dictionary and Set

Python Dictionary

Python dictionary is an unordered collection of items. Each item of a dictionary has a `key/value` pair.

Dictionaries are optimized to retrieve values when the key is known.

Creating Python Dictionary

Creating a dictionary is as simple as placing items inside curly braces `{}` separated by commas.

An item has a `key` and a corresponding `value` that is expressed as a pair (key: value).

While the values can be of any data type and can repeat, keys must be of immutable type (string, number or tuple with immutable elements) and must be unique.

we can also create a dictionary using the built-in `dict()` function.

Accessing Elements from Dictionary

While indexing is used with other data types to access values, a dictionary uses `keys`. Keys can be used either inside square brackets `[]` or with the `get()` method.

If we use the square brackets `[]`, `KeyError` is raised in case a key is not found in the dictionary. On the other hand, the `get()` method returns `None` if the key is not found.

```python
# get vs [] for retrieving elements
my_dict = {'name': 'Jack', 'age': 26}

# Output: Jack
print(my_dict['name'])

# Output: 26
print(my_dict.get('age'))

# Trying to access keys which doesn't exist throws error
# Output None
print(my_dict.get('address'))

# KeyError
print(my_dict['address'])
```

Output

```
Jack
26
None
Traceback (most recent call last):
  File "<string>", line 15, in <module>
    print(my_dict['address'])
KeyError: 'address'
```

Changing and Adding Dictionary elements

Dictionaries are mutable. We can add new items or change the value of existing items using an assignment operator.

If the key is already present, then the existing value gets updated. In case the key is not present, a new (key: value) pair is added to the dictionary.

```python
# Changing and adding Dictionary Elements
my_dict = {'name': 'Jack', 'age': 26}

# update value
my_dict['age'] = 27

#Output: {'age': 27, 'name': 'Jack'}
print(my_dict)

# add item
my_dict['address'] = 'Downtown'

# Output: {'address': 'Downtown', 'age': 27, 'name': 'Jack'}
print(my_dict)
```

Output

```
{'name': 'Jack', 'age': 27}
{'name': 'Jack', 'age': 27, 'address': 'Downtown'}
```

Removing elements from Dictionary

We can remove a particular item in a dictionary by using the `pop()` method. This method removes an item with the provided `key` and returns the `value`.

The `popitem()` method can be used to remove and return an arbitrary `(key, value)` item pair from the dictionary. All the items can be removed at once, using the `clear()` method.

We can also use the `del` keyword to remove individual items or the entire dictionary itself.

```python
# Removing elements from a dictionary

# create a dictionary
squares = {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

# remove a particular item, returns its value
# Output: 16
print(squares.pop(4))

# Output: {1: 1, 2: 4, 3: 9, 5: 25}
print(squares)

# remove an arbitrary item, return (key,value)
# Output: (5, 25)
print(squares.popitem())

# Output: {1: 1, 2: 4, 3: 9}
print(squares)

# remove all items
squares.clear()

# Output: {}
print(squares)

# delete the dictionary itself
del squares

# Throws Error
```

Output

```
16
{1: 1, 2: 4, 3: 9, 5: 25}
(5, 25)
{1: 1, 2: 4, 3: 9}
{}
Traceback (most recent call last):
  File "<string>", line 30, in <module>
    print(squares)
NameError: name 'squares' is not defined
```

Python Dictionary Methods

Method	Description
clear()	Removes all items from the dictionary.
copy()	Returns a shallow copy of the dictionary.
fromkeys(seq[, v])	Returns a new dictionary with keys from `seq` and value equal to `v` (defaults to `None`).
get(key[,d])	Returns the value of the `key`. If the `key` does not exist, returns `d` (defaults to `None`).
items()	Return a new object of the dictionary's items in (key, value) format.
keys()	Returns a new object of the dictionary's keys.
pop(key[,d])	Removes the item with the `key` and returns its value or `d` if `key` is not found. If `d` is not provided and the `key` is not found, it raises `KeyError`.
popitem()	Removes and returns an arbitrary item (**key, value**). Raises `KeyError` if the dictionary is empty.
setdefault(key[,d])	Returns the corresponding value if the `key` is in the dictionary. If not, inserts the `key` with a value of `d` and returns `d` (defaults to `None`).

```python
# Dictionary Methods
marks = {}.fromkeys(['Math', 'English', 'Science'], 0)

# Output: {'English': 0, 'Math': 0, 'Science': 0}
print(marks)

for item in marks.items():
    print(item)

# Output: ['English', 'Math', 'Science']
print(list(sorted(marks.keys())))
```

Output

```
{'Math': 0, 'English': 0, 'Science': 0}
('Math', 0)
('English', 0)
('Science', 0)
['English', 'Math', 'Science']
```

Python Dictionary Comprehension

Dictionary comprehension is an elegant and concise way to create a new dictionary from an iterable in Python.

Dictionary comprehension consists of an expression pair (**key: value**) followed by a `for` statement inside curly braces `{}`.

Here is an example to make a dictionary with each item being a pair of a number and its square.

```
# Dictionary Comprehension
squares = {x: x*x for x in range(6)}

print(squares)
```

Run Code »

Output

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Other Dictionary Operations

Dictionary Membership Test

We can test if a `key` is in a dictionary or not using the keyword `in`. Notice that the membership test is only for the `keys` and not for the `values`.

```
# Membership Test for Dictionary Keys
squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}

# Output: True
print(1 in squares)

# Output: True
print(2 not in squares)

# membership tests for key only not value
# Output: False
print(49 in squares)
```

Run Code »

Output

```
True
True
False
```

Iterating Through a Dictionary

We can iterate through each key in a dictionary using a `for` loop.

```
# Iterating through a Dictionary
squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
for i in squares:
    print(squares[i])
```

Output

```
1
9
25
49
81
```

Dictionary Built-in Functions

Built-in functions like `all()`, `any()`, `len()`, `cmp()`, `sorted()`, etc. are commonly used with dictionaries to perform different tasks.

Function	Description
all()	Return `True` if all keys of the dictionary are True (or if the dictionary is empty).
any()	Return `True` if any key of the dictionary is true. If the dictionary is empty, return `False`.
len()	Return the length (the number of items) in the dictionary.
cmp()	Compares items of two dictionaries. (Not available in Python 3)
sorted()	Return a new sorted list of keys in the dictionary.

Here are some examples that use built-in functions to work with a dictionary.

```python
# Dictionary Built-in Functions
squares = {0: 0, 1: 1, 3: 9, 5: 25, 7: 49, 9: 81}

# Output: False
print(all(squares))

# Output: True
print(any(squares))

# Output: 6
print(len(squares))

# Output: [0, 1, 3, 5, 7, 9]
print(sorted(squares))
```

Output

```
False
True
6
[0, 1, 3, 5, 7, 9]
```

```python
# empty dictionary
my_dict = {}

# dictionary with integer keys
my_dict = {1: 'apple', 2: 'ball'}

# dictionary with mixed keys
my_dict = {'name': 'John', 1: [2, 4, 3]}

# using dict()
my_dict = dict({1:'apple', 2:'ball'})

# from sequence having each item as a pair
my_dict = dict([(1,'apple'), (2,'ball')])
```

Python Dictionary items()

The `items()` method returns a view object that displays a list of dictionary's (key, value) tuple pairs.
The syntax of `items()` method is:

```
dictionary.items()
```

Note: `items()` method is similar to dictionary's `viewitems()` method .

The `items()` method doesn't take any parameters.

The `items()` method returns a view object that displays a list of a given dictionary's (key, value) tuple pair.

Example 1: Get all items of a dictionary with items()

```
# random sales dictionary
sales = { 'apple': 2, 'orange': 3, 'grapes': 4 }

print(sales.items())
```

Output

```
dict_items([('apple', 2), ('orange', 3), ('grapes', 4)])
```

Example 2: How items() works when a dictionary is modified?

```
# random sales dictionary
sales = { 'apple': 2, 'orange': 3, 'grapes': 4 }

items = sales.items()
print('Original items:', items)

# delete an item from dictionary
del[sales['apple']]
print('Updated items:', items)
```

Run Code »

Output

```
Original items: dict_items([('apple', 2), ('orange', 3), ('grapes', 4)])
Updated items: dict_items([('orange', 3), ('grapes', 4)])
```

Python Dictionary keys()

The keys() method returns a view object that displays a list of all the keys in the dictionary

The syntax of `keys()` is:

```
dict.keys()
```

`keys()` doesn't take any parameters.

`keys()` returns a view object that displays a list of all the keys.

When the dictionary is changed, the view object also reflects these changes.

Example 1: How keys() works?

```
person = {'name': 'Phill', 'age': 22, 'salary': 3500.0}
print(person.keys())

empty_dict = {}
print(empty_dict.keys())
```

Output

```
dict_keys(['name', 'salary', 'age'])
dict_keys([])
```

Example 2: How keys() works when dictionary is updated?

```
person = {'name': 'Phill', 'age': 22, }

print('Before dictionary is updated')
keys = person.keys()
print(keys)

# adding an element to the dictionary
person.update({'salary': 3500.0})
print('\nAfter dictionary is updated')
print(keys)
```

Run Cod

Output

```
Before dictionary is updated
dict_keys(['name', 'age'])

After dictionary is updated
dict_keys(['name', 'age', 'salary'])
```

Python Dictionary fromkeys()

The fromkeys() method creates a new dictionary from the given sequence of elements with a value provided by the user.

The syntax of `fromkeys()` method is:

```
dictionary.fromkeys(sequence[, value])
```

`fromkeys()` method takes two parameters:

- sequence - sequence of elements which is to be used as keys for the new dictionary
- value (Optional) - value which is set to each each element of the dictionary

`fromkeys()` method returns a new dictionary with the given sequence of elements as the keys of the dictionary.

If the value argument is set, each element of the newly created dictionary is set to the provided value.

Example 1: Create a dictionary from a sequence of keys

```
# vowels keys
keys = {'a', 'e', 'i', 'o', 'u' }

vowels = dict.fromkeys(keys)
print(vowels)
```

Run

Output

```
{'a': None, 'u': None, 'o': None, 'e': None, 'i': None}
```

Example 2: Create a dictionary from a sequence of keys with value

```
# vowels keys
keys = {'a', 'e', 'i', 'o', 'u' }
value = 'vowel'

vowels = dict.fromkeys(keys, value)
print(vowels)
```

Run Code »

Output

```
{'a': 'vowel', 'u': 'vowel', 'o': 'vowel', 'e': 'vowel', 'i': 'vowel'}
```

Python Dictionary popitem()

The Python popitem() method removes and returns the last element (key, value) pair inserted into the dictionary.

The syntax of `popitem()` is:

```
dict.popitem()
```

The `popitem()` doesn't take any parameters.

The `popitem()` method removes and returns the (key, value) pair from the dictionary in the Last In, First Out (LIFO) order.

- Returns the latest inserted element (key,value) pair from the dictionary.
- Removes the returned element pair from the dictionary.

Note: The `popitem()` method raises a `KeyError` error if the dictionary is empty.

Example: Working of popitem() method

```
person = {'name': 'Phill', 'age': 22, 'salary': 3500.0}

# ('salary', 3500.0) is inserted at the last, so it is removed.
result = person.popitem()

print('Return Value = ', result)
print('person = ', person)

# inserting a new element pair
person['profession'] = 'Plumber'

# now ('profession', 'Plumber') is the latest element
result = person.popitem()

print('Return Value = ', result)
print('person = ', person)
```

Output

```
Return Value =  ('salary', 3500.0)
person =  {'name': 'Phill', 'age': 22}
Return Value =  ('profession', 'Plumber')
person =  {'name': 'Phill', 'age': 22}
```

Python Dictionary update()

The `update()` method updates the dictionary with the elements from another dictionary object or from an iterable of key/value pairs.

The syntax of `update()` is:

```
dict.update([other])
```

The `update()` method takes either a dictionary or an iterable object of key/value pairs (generally tuples).

If `update()` is called without passing parameters, the dictionary remains unchanged.

`update()` method updates the dictionary with elements from a dictionary object or an iterable object of key/value pairs.

It doesn't return any value (returns `None`).

Example 1: Working of update()

```
d = {1: "one", 2: "three"}
d1 = {2: "two"}

# updates the value of key 2
d.update(d1)
print(d)

d1 = {3: "three"}

# adds element with key 3
d.update(d1)
print(d)
```

Output

```
{1: 'one', 2: 'two'}
{1: 'one', 2: 'two', 3: 'three'}
```

Example 2: update() When Tuple is Passed

```
dictionary = {'x': 2}

dictionary.update([('y', 3), ('z', 0)])
print(dictionary)
```

Run Code »

Output

```
{'x': 2, 'y': 3, 'z': 0}
```

Here, we have passed a list of tuples `[('y', 3), ('z', 0)]` to the `update()` function. In this case, the first element of tuple is used as the key and the second element is used as the value.

Python Dictionary setdefault()

The setdefault() method returns the value of a key (if the key is in dictionary). If not, it inserts key with a value to the dictionary.

The syntax of `setdefault()` is:

```
dict.setdefault(key[, default_value])
```

setdefault() Parameters

`setdefault()` takes a maximum of two parameters:

- key - the key to be searched in the dictionary
- default_value (optional) - `key` with a value `default_value` is inserted to the dictionary if the key is not in the dictionary. If not provided, the `default_value` will be `None`.

`setdefault()` returns:

- value of the `key` if it is in the dictionary
- None if the key is not in the dictionary and default_value is not specified
- `default_value` if `key` is not in the dictionary and `default_value` is specified

Example 1: How setdefault() works when key is in the dictionary?

```
person = {'name': 'Phill', 'age': 22}

age = person.setdefault('age')
print('person = ',person)
print('Age = ',age)
```

R

Output

```
person =  {'name': 'Phill', 'age': 22}
Age =  22
```

Example 2: How setdefault() works when key is not in the dictionary?

```
person = {'name': 'Phill'}

# key is not in the dictionary
salary = person.setdefault('salary')
print('person = ',person)
print('salary = ',salary)

# key is not in the dictionary
# default_value is provided
age = person.setdefault('age', 22)
print('person = ',person)
print('age = ',age)
```

Run Code

Output

```
person =  {'name': 'Phill', 'salary': None}
salary =  None
person =  {'name': 'Phill', 'age': 22, 'salary': None}
age =  22
```

Python Sets

Python Sets

A set is an unordered collection of items. Every set element is unique (no duplicates) and must be immutable (cannot be changed).

However, a set itself is mutable. We can add or remove items from it.

Sets can also be used to perform mathematical set operations like union, intersection, symmetric difference, etc.

Creating Python Sets

A set is created by placing all the items (elements) inside curly braces `{}`, separated by comma, or by using the built-in `set()` function.

It can have any number of items and they may be of different types (integer, float, tuple, string etc.). But a set cannot have mutable elements like lists, sets or dictionaries as its elements.

```
# set cannot have duplicates
# Output: {1, 2, 3, 4}
my_set = {1, 2, 3, 4, 3, 2}
print(my_set)

# we can make set from a list
# Output: {1, 2, 3}
my_set = set([1, 2, 3, 2])
print(my_set)

# set cannot have mutable items
# here [3, 4] is a mutable list
# this will cause an error.

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

Output

```
{1, 2, 3, 4}
{1, 2, 3}
Traceback (most recent call last):
  File "<string>", line 15, in <module>
    my_set = {1, 2, [3, 4]}
TypeError: unhashable type: 'list'
```

Creating an empty set is a bit tricky.

Empty curly braces `{}` will make an empty dictionary in Python. To make a set without any elements, we use the `set()` function without any argument.

```
# Distinguish set and dictionary while creating empty set

# initialize a with {}
a = {}

# check data type of a
print(type(a))

# initialize a with set()
a = set()

# check data type of a
print(type(a))
```

Ru

Output

```
<class 'dict'>
<class 'set'>
```

Modifying a set in Python

Sets are mutable. However, since they are unordered, indexing has no meaning.

We cannot access or change an element of a set using indexing or slicing. Set data type does not support it.

We can add a single element using the `add()` method, and multiple elements using the `update()` method. The `update()` method can take tuples, lists, strings or other sets as its argument. In all cases, duplicates are avoided.

```python
# initialize my_set
my_set = {1, 3}
print(my_set)

# my_set[0]
# if you uncomment the above line
# you will get an error
# TypeError: 'set' object does not support indexing

# add an element
# Output: {1, 2, 3}
my_set.add(2)
print(my_set)

# add multiple elements
# Output: {1, 2, 3, 4}
my_set.update([2, 3, 4])
print(my_set)

# add list and set
# Output: {1, 2, 3, 4, 5, 6, 8}
my_set.update([4, 5], {1, 6, 8})
print(my_set)
```

Output

```
{1, 3}
{1, 2, 3}
{1, 2, 3, 4}
{1, 2, 3, 4, 5, 6, 8}
```

Removing elements from a set

A particular item can be removed from a set using the methods `discard()` and `remove()`.

The only difference between the two is that the `discard()` function leaves a set unchanged if the element is not present in the set. On the other hand, the `remove()` function will raise an error in such a condition (if element is not present in the set).

```python
# Difference between discard() and remove()

# initialize my_set
my_set = {1, 3, 4, 5, 6}
print(my_set)

# discard an element
# Output: {1, 3, 5, 6}
my_set.discard(4)
print(my_set)

# remove an element
# Output: {1, 3, 5}
my_set.remove(6)
print(my_set)

# discard an element
# not present in my_set
# Output: {1, 3, 5}
my_set.discard(2)
print(my_set)

# remove an element
# not present in my_set
# you will get an error.
# Output: KeyError

my_set.remove(2)
```

Output

```
{1, 3, 4, 5, 6}
{1, 3, 5, 6}
{1, 3, 5}
{1, 3, 5}
Traceback (most recent call last):
  File "<string>", line 28, in <module>
KeyError: 2
```

Similarly, we can remove and return an item using the `pop()` method.

Since set is an unordered data type, there is no way of determining which item will be popped. It is completely arbitrary.

We can also remove all the items from a set using the `clear()` method.

```python
# initialize my_set
# Output: set of unique elements
my_set = set("HelloWorld")
print(my_set)

# pop an element
# Output: random element
print(my_set.pop())

# pop another element
my_set.pop()
print(my_set)

# clear my_set
# Output: set()
my_set.clear()
print(my_set)
```

Output

```
{'H', 'l', 'r', 'W', 'o', 'd', 'e'}
H
{'r', 'W', 'o', 'd', 'e'}
set()
```

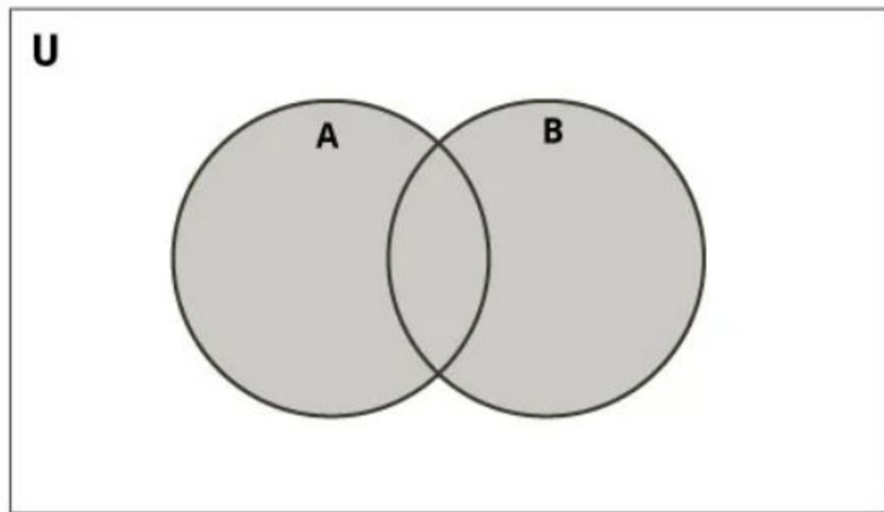
Python Set Operations

Sets can be used to carry out mathematical set operations like union, intersection, difference and symmetric difference. We can do this with operators or methods.

Let us consider the following two sets for the following operations.

```
>>> A = {1, 2, 3, 4, 5}
>>> B = {4, 5, 6, 7, 8}
```

Set Union



Set Union in Python

Union of `A` and `B` is a set of all elements from both sets.

Union is performed using `|` operator. Same can be accomplished using the `union()` method.

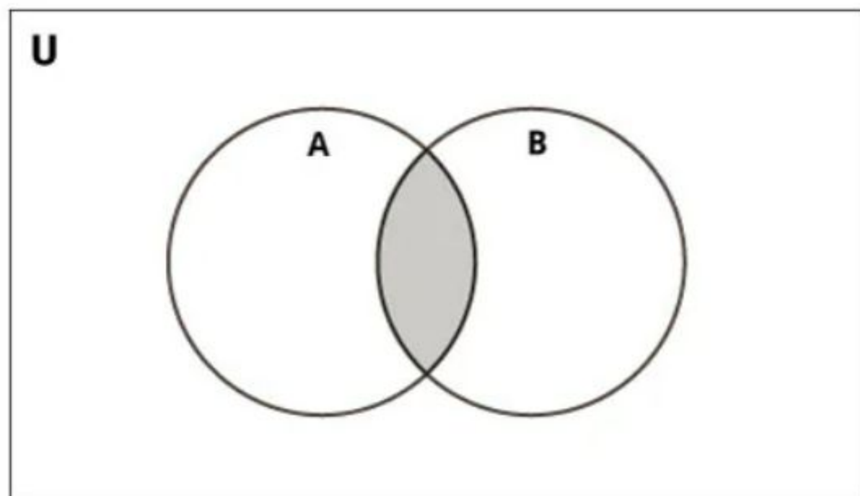
```
# Set union method
# initialize A and B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

# use | operator
# Output: {1, 2, 3, 4, 5, 6, 7, 8}
print(A | B)
```

Output

```
{1, 2, 3, 4, 5, 6, 7, 8}
```

Set Intersection



Set Intersection in Python

Intersection of `A` and `B` is a set of elements that are common in both the sets.

Intersection is performed using `&` operator. Same can be accomplished using the `intersection()` method.

```
# Intersection of sets
# initialize A and B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

# use & operator
# Output: {4, 5}
print(A & B)
```

Output

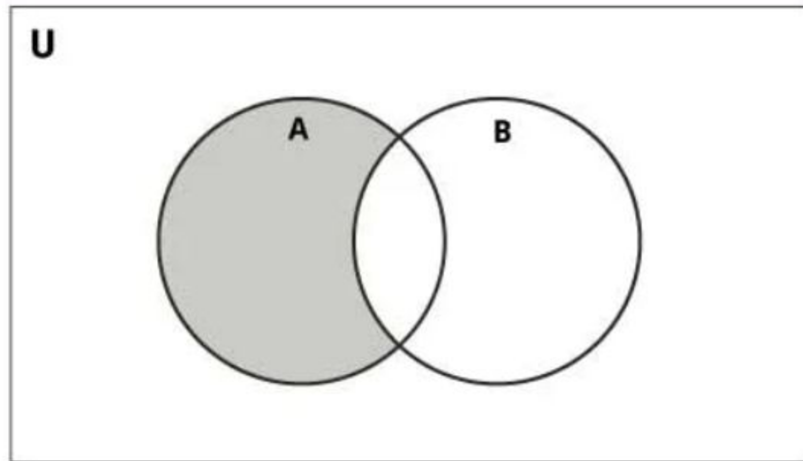
```
{4, 5}
```

Try the following examples on Python shell.

```
# use intersection function on A
>>> A.intersection(B)
{4, 5}

# use intersection function on B
>>> B.intersection(A)
{4, 5}
```

Set Difference



Set Difference in Python

Difference of the set `B` from set `A` (`A` - `B`) is a set of elements that are only in `A` but not in `B`. Similarly, `B` - `A` is a set of elements in `B` but not in `A`.

Difference is performed using `-` operator. Same can be accomplished using the `difference()` method.

```python
# initialize A and B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

# use - operator on A
# Output: {1, 2, 3}
print(A - B)
```

Output

```
{1, 2, 3}
```

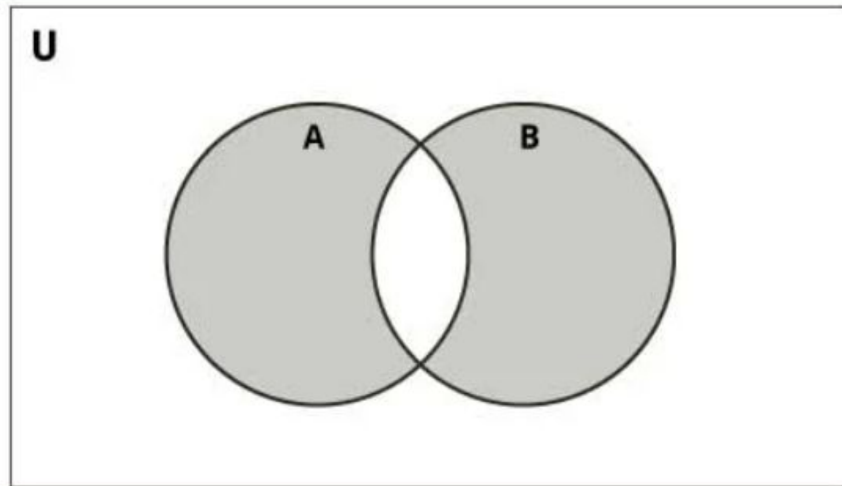
Try the following examples on Python shell.

```python
# use difference function on A
>>> A.difference(B)
{1, 2, 3}

# use - operator on B
>>> B - A
{8, 6, 7}

# use difference function on B
>>> B.difference(A)
{8, 6, 7}
```

Set Symmetric Difference



Set Symmetric Difference in Python

Symmetric Difference of `A` and `B` is a set of elements in `A` and `B` but not in both (excluding the intersection).

Symmetric difference is performed using `^` operator. Same can be accomplished using the method `symmetric_difference()`.

```
# Symmetric difference of two sets
# initialize A and B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

# use ^ operator
# Output: {1, 2, 3, 6, 7, 8}
print(A ^ B)
```

Output

```
{1, 2, 3, 6, 7, 8}
```

Try the following examples on Python shell.

```
# use symmetric_difference function on A
>>> A.symmetric_difference(B)
{1, 2, 3, 6, 7, 8}

# use symmetric_difference function on B
>>> B.symmetric_difference(A)
{1, 2, 3, 6, 7, 8}
```

Other Python Set Methods

Method	Description
add()	Adds an element to the set
clear()	Removes all elements from the set
copy()	Returns a copy of the set
difference()	Returns the difference of two or more sets as a new set
difference_update()	Removes all elements of another set from this set
discard()	Removes an element from the set if it is a member. (Do nothing if the element is not in set)
intersection()	Returns the intersection of two sets as a new set
intersection_update()	Updates the set with the intersection of itself and another
isdisjoint()	Returns `True` if two sets have a null intersection
issubset()	Returns `True` if another set contains this set
issuperset()	Returns `True` if this set contains another set

pop()	Removes and returns an arbitrary set element. Raises `KeyError` if the set is empty
remove()	Removes an element from the set. If the element is not a member, raises a `KeyError`
symmetric_difference()	Returns the symmetric difference of two sets as a new set
symmetric_difference_update()	Updates a set with the symmetric difference of itself and another
union()	Returns the union of sets in a new set
update()	Updates the set with the union of itself and others

Other Set Operations

Set Membership Test

We can test if an item exists in a set or not, using the `in` keyword.

```
# in keyword in a set
# initialize my_set
my_set = set("apple")

# check if 'a' is present
# Output: True
print('a' in my_set)

# check if 'p' is present
# Output: False
print('p' not in my_set)
```

Run

Output

```
True
False
```

Iterating Through a Set

We can iterate through each item in a set using a `for` loop.

```
>>> for letter in set("apple"):
...     print(letter)
...
a
p
e
l
```

Built-in Functions with Set

Built-in functions like `all()`, `any()`, `enumerate()`, `len()`, `max()`, `min()`, `sorted()`, `sum()` etc. are commonly used with sets to perform different tasks.

Function	Description
all()	Returns `True` if all elements of the set are true (or if the set is empty).
any()	Returns `True` if any element of the set is true. If the set is empty, returns `False`.
enumerate()	Returns an enumerate object. It contains the index and value for all the items of the set as a pair.
len()	Returns the length (the number of items) in the set.
max()	Returns the largest item in the set.
min()	Returns the smallest item in the set.
sorted()	Returns a new sorted list from elements in the set(does not sort the set itself).
sum()	Returns the sum of all elements in the set.

Python Set add()

The `add()` method adds a given element to a set. If the element is already present, it doesn't add any element.

The syntax of `add()` method is:

```
set.add(elem)
```

`add()` method doesn't add an element to the set if it's already present in it.

Also, you don't get back a set if you use `add()` method when creating a set object.

```
noneValue = set().add(elem)
```

The above statement doesn't return a reference to the set but 'None', because the statement returns the return type of add which is `None`.

`add()` method takes a single parameter:

- elem - the element that is added to the set

`add()` method doesn't return any value and returns `None`.

Example 1: Add an element to a set

```
# set of vowels
vowels = {'a', 'e', 'i', 'u'}

# adding 'o'
vowels.add('o')
print('Vowels are:', vowels)

# adding 'a' again
vowels.add('a')
print('Vowels are:', vowels)
```

Output

```
Vowels are: {'a', 'i', 'o', 'u', 'e'}
Vowels are: {'a', 'i', 'o', 'u', 'e'}
```

Note: Order of the vowels can be different.

Python Set remove()

The `remove()` method removes the specified element from the set.

The syntax of the `remove()` method is:

```
set.remove(element)
```

The `remove()` method takes a single element as an argument and removes it from the set.

The `remove()` removes the specified element from the set and updates the set. It doesn't return any value.

If the element passed to `remove()` doesn't exist, KeyError exception is thrown.

Example 1: Remove an Element From The Set

```
# language set
language = {'English', 'French', 'German'}

# removing 'German' from language
language.remove('German')

# Updated language set
print('Updated language set:', language)
```

Output

```
Updated language set: {'English', 'French'}
```

Example 2: Deleting Element That Doesn't Exist

```
# animal set
animal = {'cat', 'dog', 'rabbit', 'guinea pig'}

# Deleting 'fish' element
animal.remove('fish')

# Updated animal
print('Updated animal set:', animal)
```

Output

```
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
    animal.remove('fish')
KeyError: 'fish'
```

Python Set discard()

The discard() method removes a specified element from the set (if present).

The syntax of `discard()` in Python is:

```
s.discard(x)
```

`discard()` method takes a single element `x` and removes it from the set (if present).

`discard()` removes element `x` from the set if the element is present.

This method returns `None` (meaning, absence of a return value).

```python
numbers = {2, 3, 4, 5}

numbers.discard(3)
print('numbers = ', numbers)

numbers.discard(10)
print('numbers = ', numbers)
```

Output

```
numbers =  {2, 4, 5}
numbers =  {2, 4, 5}
```

```python
numbers = {2, 3, 5, 4}

# Returns None
# Meaning, absence of a return value
print(numbers.discard(3))

print('numbers =', numbers)
```

Output

```
None
numbers = {2, 4, 5}
```

Python Set intersection()

The `intersection()` method returns a new set with elements that are common to all sets.

The syntax of `intersection()` in Python is:

```
A.intersection(*other_sets)
```

`intersection()` allows arbitrary number of arguments (sets).

> Note: `*` is not part of the syntax. It is used to indicate that the method allows arbitrary number of arguments.

`intersection()` method returns the intersection of set `A` with all the sets (passed as argument).

If the argument is not passed to `intersection()`, it returns a shallow copy of the set (`A`).

Example 1: Python Set intersection()

```
A = {2, 3, 5, 4}
B = {2, 5, 100}
C = {2, 3, 8, 9, 10}

print(B.intersection(A))
print(B.intersection(C))
print(A.intersection(C))
print(C.intersection(A, B))
```

Output

```
{2, 5}
{2}
{2, 3}
{2}
```

Example 3: Set Intersection Using & operator

You can also find the intersection of sets using `&` operator.

```
A = {100, 7, 8}
B = {200, 4, 5}
C = {300, 2, 3, 7}
D = {100, 200, 300}

print(A & C)
print(A & D)

print(A & C & D)
print(A & B & C & D)
```

Output

```
{7}
{100}
set()
set()
```

Python Set difference()

The difference() method returns the set difference of two sets.

If `A` and `B` are two sets. The set difference of `A` and `B` is a set of elements that exists only in set `A` but not in `B`.

For example:

```
If A = {1, 2, 3, 4}
B = {2, 3, 9}

Then,
A - B = {1, 4}
B - A = {9}
```

The syntax of the set `difference()` method in Python is:

```
A.difference(B)
```

Here, `A` and `B` are two sets. The following syntax is equivalent to `A-B`.

Return Value from difference()

`difference()` returns the difference between two sets which is also a set. It doesn't modify the original sets.

Example 1: How difference() works in Python?

```
A = {'a', 'b', 'c', 'd'}
B = {'c', 'f', 'g'}

# Equivalent to A-B
print(A.difference(B))

# Equivalent to B-A
print(B.difference(A))
```

Output

```
{'b', 'a', 'd'}
{'g', 'f'}
```

Python Set difference_update()

The difference_update() updates the set calling difference_update() method with the difference of sets.

If `A` and `B` are two sets. The set difference of `A` and `B` is a set of elements that exists only in set `A` but not in `B`.

The syntax of difference_update() is:

```
A.difference_update(B)
```

Here, `A` and `B` are two sets. `difference_update()` updates set `A` with the set difference of `A-B`.

`difference_update()` returns `None` indicating the object (set) is mutated.

Suppose,

```
result = A.difference_update(B)
```

When you run the code,

- `result` will be `None`
- `A` will be equal to `A-B`
- `B` will be unchanged

Example: How difference_update() works?

```
A = {'a', 'c', 'g', 'd'}
B = {'c', 'f', 'g'}

result = A.difference_update(B)

print('A = ', A)
print('B = ', B)
print('result = ', result)
```

Output

```
A =  {'d', 'a'}
B =  {'c', 'g', 'f'}
result =  None
```

Python Set issubset()

The issubset() method returns True if all elements of a set are present in another set (passed as an argument). If not, it returns False.

Set `A` is said to be the subset of set `B` if all elements of `A` are in `B`.

The syntax of `issubset()` is:

```
A.issubset(B)
```

The above code checks if `A` is a subset of `B`.

`issubset()` returns

- `True` if A is a subset of B
- `False` if A is not a subset of B

```
A = {1, 2, 3}
B = {1, 2, 3, 4, 5}
C = {1, 2, 4, 5}

# Returns True
print(A.issubset(B))

# Returns False
# B is not subset of A
print(B.issubset(A))

# Returns False
print(A.issubset(C))

# Returns True
print(C.issubset(B))
```

Output

```
True
False
False
True
```

Python Set isdisjoint()

The isdisjoint() method returns True if two sets are disjoint sets. If not, it returns False.

Two sets are said to be disjoint sets if they have no common elements. For example:

```
A = {1, 5, 9, 0}
B = {2, 4, -5}
```

Here, sets `A` and `B` are disjoint sets.

The syntax of `isdisjoint()` is:

```
set_a.isdisjoint(set_b)
```

isdisjoint() Parameters

`isdisjoint()` method takes a single argument (a set).

You can also pass an iterable (list, tuple, dictionary, and string) to `disjoint()`. `isdisjoint()` method will automatically convert iterables to set and checks whether the sets are disjoint or not.

Return Value from isdisjoint()

`isdisjoint()` method returns

- `True` if two sets are disjoint sets (if `set_a` and `set_b` are disjoint sets in above syntax)
- `False` if two sets are not disjoint sets

Example 1: How isdisjoint() works?

```
A = {1, 2, 3, 4}
B = {5, 6, 7}
C = {4, 5, 6}

print('Are A and B disjoint?', A.isdisjoint(B))
print('Are A and C disjoint?', A.isdisjoint(C))
```

Output

```
Are A and B disjoint? True
Are A and C disjoint? False
```

Example 2: isdisjoint() with Other Iterables as arguments

```
A = {'a', 'b', 'c', 'd'}
B = ['b', 'e', 'f']
C = '5de4'
D ={1 : 'a', 2 : 'b'}
E ={'a' : 1, 'b' : 2}

print('Are A and B disjoint?', A.isdisjoint(B))
print('Are A and C disjoint?', A.isdisjoint(C))
print('Are A and D disjoint?', A.isdisjoint(D))
print('Are A and E disjoint?', A.isdisjoint(E))
```

Run Co

Output

```
Are A and B disjoint? False
Are A and C disjoint? False
Are A and D disjoint? True
Are A and E disjoint? False
```

THE END