Department of Computer Science and Engineering (UG Studies)

PES University, Bangalore, India

Introduction to Computing using Python (UE19CS101)

Mr. Prakash C O
Asst. Professor,
Dept. of CSE, PESU
coprakasha@pes.edu

Data Structure: Set

A set is a collection of distinct elements, which is unordered, unindexed and mutable. In Python sets are written with curly brackets.

Examples

```
fruitset = { “apple”, “orange”, “kiwi”, “banana”, “cherry”}
set1 = {(1, 'a', 'hi'), 2, 'hello', 5.56, 3+5j, True}
```

A set is a data structure with zero or more elements with the following attributes.

- **Elements are unique – does not support repeated elements**
- **Elements should be hashable.**
    - **An object is hashable if it has a hash value which never changes during its lifetime** (it needs a **__hash__()** method), and can be compared to other objects (it needs an **__eq__()** method). Hashable objects which compare equal must have the same hash value.
    - **Hashability makes an object usable as a dictionary key and a set member, because these data structures use the hash value internally.**
    - **All of Python’s immutable built-in objects are hashable**, while no mutable containers (such as lists or dictionaries) are. **Objects which are instances of user-defined classes are hashable by default**; they all compare unequal, and their hash value is their **id()**.

Examples of hashable objects:

```
int, float, complex, bool, string, tuple, range, frozenset, bytes, decimal
```

Examples of Unhashable objects:

```
list, dict, set, bytearray
```

Example 1:

```
#set1={ [ 1, 'a', 'hi' ], 2, 'hello', {3, 4.5, 'how r u' } }
#TypeError: unhashable type: 'list'

#set2={ ( 1, 'a', 'hi' ), 2, 'hello', {1:'hi', 2:'hello', 3:'how r u'} }
#TypeError: unhashable type: 'dict'

set3={ ( 1, 'a', 'hi' ), 2, 'hello', 5.56, 3+5j, True }
print(set3)
```

Example 2:

```
my_dict = {'name': 'John', tuple([1,2,3]):'values'}
print(my_dict)
```

- **Set is mutable**

  Mutable objects can change their value but keep their **id()**

  Note: immutable
  - **An object with a fixed value**.
  - **Immutable objects include numbers/numeric(int, float, complex), bool, strings and tuples**.
  - **Immutable object cannot be altered.** A new object has to be created if a different value has to be stored.
  - **Immutable objects play an important role in places where a constant hash value is needed**, for example as a key in a dictionary.

- **Set is unordered – we cannot assume the order of elements in a set.**
- **Set is an iterable – eager and not lazy**
  - **A container object capable of returning its members one at a time.**
    Examples of iterables include all sequence types (such as **list, str, and tuple**) and some non-sequence types like **set, dict** and **file** and objects of any classes you define with an **__iter__()** or **__getitem__()** method.
  - Iterables can be used in a **for** loop and in many other places where a sequence is needed (**zip()**, **map()**, ...). When an iterable object is passed as an argument to the built-in function **iter()**, it returns an iterator for the object.
    This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call **iter()** or deal with iterator objects yourself.
    The for statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop. See also *iterator*, *sequence*, and *generator*.

- **We cannot index on a set. You cannot access items in a set by referring to an index, since sets are unordered the items has no index.**
- **We can check for membership using the *in* operator. This would be faster in case of a set compared to a list, a tuple or a string.**
- **Sets support many mathematical operations on sets.**
  - **Membership : in**
  - **Union : |**
  - **Intersection : &**
  - **Set difference : -**

- Symmetric difference : ^
- Equality and inequality: = !=
- Subset and superset : < <= > >=
- Set constructor { … }

- To create an empty set, we must use the set constructor `set()` and not { }. The latter would become a dict.

We use sets for

- Deduplication : removal of repeated elements
- Finding unique elements
- Comparing two iterables for common elements or differences.

Note: Sets are unordered, so you cannot be sure in which order the items will appear.

Let us look at a few programs illustrating the usefulness and the power of the set data structure.

```
# name : 0_intro_set.py

# set:
#     is a data structure
#     has # of elements
#     elements are unique
#     make a set: { }

a = { 10, 30, 10, 40, 20, 50, 30 }
print(a)     # elements are unique; there is no particular order
{40, 10, 50, 20, 30}

# list: l
#     ith element : l[i]
#     next element : l[i + 1]
#     previous element : l[i - 1]
#     is a sequence
#     concept of position for each element

# set :
#     not sequence
#     no concept of an element in a particular position
#     represents a finite set of math

# set is an iterable

a = { 10, 30, 10, 40, 20, 50, 30 }
for i in a :
```

```
      print(i, end = " ")
print()
40 10 50 20 30

# check for membership
print(100 in a) # False
print(20 in a) # True

# set operations
s1 = {1, 2, 3, 4, 5}
s2 = {1, 3, 5, 7, 9}

# union
print(s1 | s2) # {1, 2, 3, 4, 5, 7, 9}

# intersection
print(s1 & s2) # {1, 3, 5}

# set diference
print(s1 - s2) # {2, 4}

# symmetric diference
print(s1 ^ s2) # {2, 4, 7, 9}


#print("what : ", s1[0]) # TypeError: 'set' object does not support indexing

# Creates a set; initialized by calling the constructor of set
s3 = set()
s4 = set([11, 33, 22, 11, 33, 11, 11, 44, 22])
print(s3)      # set()
print(s4)      # { 33, 11, 44, 22 }

s5 = set("mississippi")
print(s5)      # {'s', 'i', 'm', 'p'}

# string : double or single quote : string should be on a single line
# 3 double quotes or 3 single quotes : string can appear or span multiple lines

str1 = """  do not trouble trouble
            till trouble
            troubles you  """
#print(str1.split())
print(set(str1.split()))

"""
# output in order
# version 1
l = list(set(str1.split()))
l.sort()
print(l)
```

```
"""

# version 2:
print(sorted(set(str1.split())))
str2 = """  betsy botsome bought some butter but the butter was bitter
            betsy botsome bought some better butter to make the bitter butter better """
print(sorted(set(str2.split())))
```

Let us look at some pieces of code from this program.

- **This creates a set – all the elements are enumerated.**

  The elements shall be unique.
  The order of output is not defined as the set is an unordered collection.

```
a = { 10, 30, 10, 40, 20, 50, 30 }
print(a) # elements are unique; there is no particular order
```

- **Set is iterable, but not indexable. a[2] will be an error.**
  The order of output is still not defined.

```
# set in an iterable
for i in a :
      print(i, end = " ")
print()
```

- **The code is self evident.**

```
# check for membership
print(100 in a) # False
print(20 in a) # True
```

- **Set operations**
  These are self evident. The order of the output is not defined in each of these cases.
- **# set operations**

```
      s1 = {1, 2, 3, 4, 5}
      s2 = {1, 3, 5, 7, 9}
```

  - **# union**

```
print(s1 | s2) # {1, 2, 3, 4, 5, 7, 9}
```

  - **# intersection**

```
print(s1 & s2) # {1, 3, 5}
```

  - **# set difference**

```
print(s1 - s2) # {2, 4}
print(s2 - s1) # {7, 9}
```

  - **# symmetric difference**

```
print(s1 ^ s2) # {2, 4, 7, 9}
```

- **Empty set creation**

```
      s3 = set()
```

- **Finding unique elements in a string**

```
s5 = set("mississippi")
print(s5) # {'s', 'i', 'm', 'p'}

# s6 = {"mississippi"}
# print(s6)
```

- **Finding unique words in a *string str1*; words separated by white space**
  We split the string based on white space – pass this list of strings as argument to the set constructor. If necessary pass this as argument to the list constructor to make a list.

```
l = list(set(str1.split()))
```

- This is same as the earlier case – but the elements of the set are sorted into a list by using the function sorted.

```
print(sorted(set(str2.split())))
```

Another example of removing repeated elements.

```
# name : 1_remove_repeated.py
# deduplication : removed repeated elements
a = [11, 33, 11, 33, 11, 44, 22, 55, 55, 11]
a = list(set(a))
print(a)
[11, 33, 44, 22, 55]
```

Let us try creating sets with the required elements.

```
# name : 2_operations_set.py

# create sets with the required elements
# make a set of numbers from 2 to n

n = 10
#s = {} #not an empty set,   it is <class dict>

# not good; it works
# version 1
s = set()
for i in range(2, n + 1):
      s.add(i)
print(s, type(s))    #{2, 3, 4, 5, 6, 7, 8, 9, 10} <class 'set'>

# version 2
s = set(range(2, n + 1))
print(s, type(s))    # {2, 3, 4, 5, 6, 7, 8, 9, 10} <class 'set'>
#-----------------------------------

# is set empty or not
s1 = set()
```

```
s2 = set("fool")      # len(s2) is 3
print("empty : ", len(s1) == 0)     # empty : True
print("empty : ", len(s2) == 0)     # empty : False

"""
if len(s1) == 0 :
      print("empty")
else:
      print("non empty")

if len(s2) == 0 :
      print("empty")
else:
      print("non empty")
"""

if s1 :
      print("empty")
else:
      print("non empty")

if s2 :
      print("empty")
else:
      print("non empty")

# empty data structure => False
# non-empty data structure => True

# remove elements from a set
s3 = set(range(10)) # 0 .. 9

# remove 2 4 6 8 # remove multiples of 2
for ele in range(2, 10, 2) :
      s3.remove(ele)
print(s3)

s3 = set(range(10)) # s3 is {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
s3 = s3 - set(range(2, 10, 2))   # set(range(2, 10, 2)) is {8, 2, 4, 6}
print(s3)   # {0, 1, 3, 5, 7, 9}
```

Let us discuss the fragments of the code from this example.

- **Create a set of numbers from 2 to n.**
  - **Method 1:** use a for loop; iterate on the range object `range(2, n + 1);`

    add each element to the set
  - **Method 2:** pass the range object as an argument to the set constructor

    This method is clean and elegant.

    `s = set(range(2, n + 1))`

- **Check whether a set is empty**
  - `len(s) == 0 # not preferred`
  - `s  # preferred, but empty set is False; non-empty set is True`

- **Remove elements from a set**

  Given a set **`s3={1, 2, 3, 4, 5, 6, 7, 8, 9}`** remove elements 2, 4, 6, 8.
  - **Method 1:**
    iterate through a range or a list object containing 2, 4, 6, 8. Call remove for each element on the set.
  - **Method 2:**
    create a set of the elements to be removed – use set difference to remove the elements. This is preferred.
    `s3 = s3 - set(range(2, 10, 2))`

We shall now discuss a very interesting method to generate prime numbers which does not involve any division operator at all. This algorithm is called Sieve of Eratosthenes.

```
# name: 3_sieve.py
# generate prime numbers (no division; most efficient algorithm)
   Sieve of Eratosthenes
   • get a number(say n)
   • make a set of numbers from 2 to n - say sieve

   • while sieve is not empty
        ○ find the smallest (small)
        ○ print it (that is a prime)
        ○ remove small and its multiples from the sieve
```

```
n = int(input("Enter an integer : "))
# make a set of numbers from 2 to n - say sieve
sieve = set(range(2, n + 1))
#print(sieve)
while sieve :
      small = min(sieve)
      print(small, end = " ")
      sieve = sieve - set(range(small, n + 1,small))
```

```
Enter an integer: 30
2 3 5 7 11 13 17 19 23 29
```

Make a set called sieve of elements from 2 to a given number n.
While the sieve is not empty, remove the smallest element – which will be a prime. Then remove that element and its multiples.
Thats all. **Are the sets powerful?**

Set Methods

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

Method	Description
`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 **Note:** Sets are unordered, so when using the pop() method, you will not know which item that gets removed.
`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

Access Items

You cannot access items in a set by referring to an index, since sets are unordered the items has no index.
But you can loop through the set items using a `for` loop, or ask if a specified value is present in a set, by using the `in` keyword.

Example

Loop through the set, and print the values:

```
carset = {"apple", "banana", "cherry"}
for x in carset:
  print(x)
```

Example

Check if "banana" is present in the set:

```
carset = {"apple", "banana", "cherry"}
print("banana" in carset)
```

Change Items

Once a set is created, you cannot change its items, but you can add new items.

Add Items

To add one item to a set use the `add()` method.
To add more than one item to a set use the `update()` method.

Example

Add an item to a set, using the `add()` method:

```
carset = {"apple", "banana", "cherry"}
carset.add("orange")
print(carset)
```

Example

Add multiple items to a set, using the `update()` method:

```
carset = {"apple", "banana", "cherry"}
carset.update(["orange", "mango", "grapes"])
print(carset)
```

Get the Length of a Set

To determine how many items a set has, use the `len()` method.

Example

Get the number of items in a set:

```
carset = {"apple", "banana", "cherry"}
print(len(carset))
```

Remove Item

To remove an item in a set, use the `remove()`, or the `discard()` method.

Example

Remove "banana" by using the `remove()` method:

```
carset = {"apple", "banana", "cherry"}
carset.remove("banana")
print(carset)
```

Note: If the item to remove does not exist, `remove()` will raise an error.

Example

Remove "banana" by using the `discard()` method:

```
carset = {"apple", "banana", "cherry"}
carset.discard("banana")
print(carset)
```

Note: If the item to remove does not exist, `discard()` will **NOT** raise an error.

You can also use the `pop()`, method to remove an item, but this method will remove the *last* item. Remember that sets are unordered, so you will not know what item that gets removed.

The return value of the `pop()` method is the removed item.

Example

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

```
carset = {"apple", "banana", "cherry"}
x = carset.pop()
print(x)
print(carset)
```

Note: Sets are *unordered*, so when using the `pop()` method, you will not know which item that gets removed.

Example

The `clear()` method empties the set:

```
carset = {"apple", "banana", "cherry"}
carset.clear()
print(carset)
```

Example

The `del` keyword will delete the set completely:

```
carset = {"apple", "banana", "cherry"}
del(carset)
print(carset)
```

Join Two Sets

There are several ways to join two or more sets in Python.

You can use the `union()` method that returns a new set containing all items from both sets, or the `update()` method that inserts all the items from one set into another:

Example

The `union()` method returns a new set with all items from both sets:

```
set1 = {"a", "b" , "c"}
set2 = {1, 2, 3}

set3 = set1.union(set2)
print(set3)
```

Example

The `update()` method inserts the items in set2 into set1:

```
set1 = {"a", "b" , "c"}
set2 = {1, 2, 3}

set1.update(set2)
print(set1)
```

Note: Both `union()` and `update()` will exclude any duplicate items.

There are other methods that joins two sets and keeps ONLY the duplicates, or NEVER the duplicates, check the full list of set methods in the bottom of this page.

The set() Constructor

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

Example

Using the set() constructor to make a set:

```
carset = set(("apple", "banana", "cherry")) # note the double round-brackets
print(carset)
```

x1.isdisjoint(x2)

Determines whether or not two sets have any elements in common.

`x1.isdisjoint(x2)` returns `True` if `x1` and `x2` have no elements in common

Note: There is no operator that corresponds to the `.isdisjoint()` method.

x1.issubset(x2)

x1 <= x2

Determine whether one set is a subset of the other.

In set theory, a set `x1` is considered a subset of another set `x2` if every element of `x1` is in `x2`.

`x1.issubset(x2)` and `x1 <= x2` return `True` if `x1` is a subset of `x2`:

```
>>> x1 = {'foo', 'bar', 'baz'}
>>> x1.issubset({'foo', 'bar', 'baz', 'qux', 'quux'})
True
>>> x2 = {'baz', 'qux', 'quux'}
>>> x1 <= x2
False
```

A set is considered to be a subset of itself:

```
>>> x = {1, 2, 3, 4, 5}
>>> x.issubset(x)
True
>>> x <= x
True
```

It seems strange, perhaps. But it fits the definition—every element of `x` is in `x`.

x1 < x2

Determines whether one set is a proper subset of the other.

A proper subset is the same as a subset, except that the sets can't be identical. A set `x1` is considered a proper subset of another set `x2` if every element of `x1` is in `x2`, and `x1` and `x2` are not equal.

`x1 < x2` returns `True` if `x1` is a proper subset of `x2`:

```
>>> x1 = {'foo', 'bar'}
>>> x2 = {'foo', 'bar', 'baz'}
>>> x1 < x2
True
>>> x1 = {'foo', 'bar', 'baz'}
>>> x2 = {'foo', 'bar', 'baz'}
>>> x1 < x2
False
```

While a set is considered a subset of itself, it is not a proper subset of itself:

```
>>> x = {1, 2, 3, 4, 5}
>>> x <= x
True
>>> x < x
False
```

Note: The `<` operator is the only way to test whether a set is a proper subset. There is no corresponding method.

x1.issuperset(x2)
x1 >= x2

Determine whether one set is a superset of the other.

A superset is the reverse of a subset. A set x1 is considered a superset of another set x2 if x1 contains every element of x2.

`x1.issuperset(x2)` and `x1 >= x2` return `True` if x1 is a superset of x2:

References:

1. **set_dict.pdf – Prof. N S Kumar, Dept. of CSE, PES University.**
2. **https://www.w3schools.com/python/python_sets.asp**
3. **https://www.w3schools.com/python/python_dictionaries.asp**
4. **https://docs.python.org/3.1/glossary.html**