```
# A Set in Python is used to store a collection of items with the following properties.
# A set is an unordered, mutable collection of unique elements in Python.
# It is defined using curly braces {} or the set() constructor.
# No duplicate elements. If try to insert the same item again, it overwrites previous one.
# An unordered collection. When we access all items, they are accessed without any specific order and we cannot access items us
# Internally use hashing that makes set efficient for search, insert and delete operations. It gives a major advantage over a l
# Mutable, meaning we can add or remove elements after their creation, the individual elements within the set cannot be changed
```

```
# Creating a set
my_set = {1, 2, 3, 4}
print(my_set)

# Using set() constructor
another_set = set([1, 2, 2, 3, 4])
print(another_set)
```

```
{1, 2, 3, 4}
{1, 2, 3, 4}
```

```
my_set = {1, 2, 3}
my_set.add(4)
print(my_set)
```

```
{1, 2, 3, 4}
```

```
my_set = {1, 2}
my_set.update([2, 4, 5])
print(my_set)
```

```
{1, 2, 4, 5}
```

```
# remove(x): Removes x from the set. Raises an error if x is not found.
# discard(x): Removes x if present, but does not raise an error if x is missing.
my_set = {1, 2, 3}
my_set.remove(2)
print(my_set)  # Output: {1, 3}
my_set.discard(10)  # No error, even though 10 is not in the set
```

```
{1, 3}
```

```
# pop(): Removes and returns a random element (since sets are unordered).

my_set = {10, 20, 30}
removed_item = my_set.pop()
print(removed_item)  # Output: Random element from the set
print(my_set)
```

```
10
{20, 30}
```

```
my_set = {1, 2, 3}
my_set.clear()
print(my_set)  # Output: set()
```

```
set()
```

```
# Union
# union(set2): Returns a new set with elements from both sets.
a = {1, 2, 3}
b = {3, 4, 5}
print(a.union(b))
print(a)


print(a | b)
```

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

```
# Intersection
# intersection(set2): Returns a new set with common elements.
```

```
print(a.intersection(b))  # Output: {3}


print(a & b)  # Output: {3}
```

```
{3}
{3}
```

```
print(a.difference(b))  # Output: {1, 2}

print(a - b)  # Output: {1, 2}
```

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

```
# Symmetric Difference
# symmetric_difference(set2): Returns a set with elements in either set1 or set2, but not both.
print(a.symmetric_difference(b))  # Output: {1, 2, 4, 5}

print(a ^ b)  # Output: {1, 2, 4, 5}
```

```
{1, 2, 4, 5}
{1, 2, 4, 5}
```

```
x = {1, 2}
y = {1, 2, 3, 4}
print(x.issubset(y))  # Output: True


print(x <= y)  # Output: True
```

```
True
True
```

```
print(y.issuperset(x))  # Output: True


print(y >= x)  # Output: True
```

```
True
True
```

```
# isdisjoint(set2): Returns True if both sets have no common elements.
a = {1, 2, 3}
b = {4, 5, 6}
print(a.isdisjoint(b))  # Output: True
```

```
True
```

```
# Frozen Sets
# A frozenset is an immutable set. You can create one using frozenset(iterable).
f_set = frozenset([1, 2, 3, 4])
print(f_set)  # Output: frozenset({1, 2, 3, 4})
# Cannot use methods like add(), remove(), or pop().
```

```
frozenset({1, 2, 3, 4})
```

```
# ✓ Sets store unique values (duplicates are removed).
# ✓ Unordered collection, so indexing and slicing are not possible.
# ✓ Supports mathematical operations (union, intersection, difference, etc.).
# ✓ Mutable (except for frozenset).
```

```
#max
print(max(a))
#min
print(min(a))
#sum
print(sum(a))
#len
print(len(a))
#is present
print(2 in a)
```

```
#is not present
print(2 not in a)
```

```
3
1
6
3
True
False
```

```
#comphrehension
set1 = set(x for x in range(20) if x%2==0)
print(set1)
```

```
{0, 2, 4, 6, 8, 10, 12, 14, 16, 18}
```

```
# Proper Subset (< Operator)
# A set A is a proper subset of set B if:
# ✓ All elements of A exist in B
# ✓ A is NOT equal to B (A must have fewer elements)
A = {1, 2, 3}
B = {1, 2, 3, 4, 5}

print(A < B)  # Output: True (A is a proper subset of B)
print(A.issubset(B))  # Output: True (A is a subset of B)


# ✅ {1, 2, 3} is fully inside {1, 2, 3, 4, 5}
# ✅ A is smaller than B, so it is a proper subset.
```

```
True
True
```

```
# Not a Proper Subset
X = {1, 2, 3}
Y = {1, 2, 3}

print(X < Y)  # Output: False (Because X == Y)
print(X.issubset(Y))  # Output: True (X is a subset but not a proper subset)
# 🚨 If two sets are equal, they are NOT proper subsets!
```

```
False
True
```

```
# Proper Superset (> Operator)
# A set A is a proper superset of set B if:
# ✓ A contains all elements of B
# ✓ A is NOT equal to B (A must have extra elements)
A = {1, 2, 3, 4, 5}
B = {1, 2, 3}

print(A > B)  # Output: True (A is a proper superset of B)
print(A.issuperset(B))  # Output: True (A is a superset of B)
# ✅ {1, 2, 3, 4, 5} contains {1, 2, 3}
# ✅ A has extra elements, so it is a proper superset.

# Edge Case: Not a Proper Superset
X = {1, 2, 3}
Y = {1, 2, 3}

print(X > Y)  # Output: False (Because X == Y)
print(X.issuperset(Y))  # Output: True (X is a superset but not a proper superset)
# 🚨 If two sets are equal, they are NOT proper supersets!
```

```
True
True
False
True
```

```
# intersection_update()
# Modifies the original set to keep only the common elements.
# Does not return a new set; instead, it updates the original set.
A = {1, 2, 3, 4}
B = {3, 4, 5, 6}

A.intersection_update(B)
```

McAfee | WebAdvisor
Your download's being scanned.
We'll let you know if there's an issue.

```
print(A)  # Output: {3, 4}
print(B)  # Output: {3, 4, 5, 6} (unchanged)
# ✓ A is updated to keep only {3, 4} (elements common in both sets).
```

```
{3, 4}
{3, 4, 5, 6}
```

```
# difference_update()
# Modifies the original set to remove elements that exist in another set.
# Does not return a new set; instead, it updates the original set.
A = {1, 2, 3, 4}
B = {3, 4, 5, 6}

A.difference_update(B)

print(A)  # Output: {1, 2}
print(B)  # Output: {3, 4, 5, 6} (unchanged)
# ✓ A is updated to remove {3, 4} because they exist in B.
# ✓ B remains unchanged.
```

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

```
# s = {1,2,[3,4]}   # ❌ TypeError
# print(s)
# Set elements must be immutable (hashable)
# allowed - int, float, string, tuple
# not allowed - list, dict, set

# Python stores set elements using a hash table (same as dictionary keys).



# ⚠️ The Problem if elements were mutable

# Imagine Python allowed a list inside a set:

# s = {[1,2,3]}   # suppose it worked


# Python calculates hash of [1,2,3] and stores it somewhere.

# Now you modify the list:

# x = [1,2,3]
# s = {x}

# x.append(4)   # value changed!


# Now the value is [1,2,3,4]

# What breaks?

# The hash changes
# But Python still thinks it is stored at the old location.

# So Python can no longer find it:

# print(x in s)   # ??? unpredictable / broken

# sets become corrupted
```