Python Sets

So far, you have learned about:

```
Lists  → ordered, changeable collections
Tuples → ordered, unchangeable collections
```

Now we need another useful Python data structure:

> **Set**

A set is a collection designed primarily for **unique values** and **set operations** such as union, intersection, and difference.

The most important idea is:

> **A set does not store duplicate values.**

This makes sets extremely useful when you need to remove duplicates, check membership, or compare groups of data.

---

1. What Is a Set?

A set is an unordered collection of unique values.

Example:

```
languages = {"Python", "JavaScript", "Python", "Java"}

print(languages)
```

The duplicate `"Python"` is automatically removed.

The result contains each value only once.

```
{"Python", "JavaScript", "Java"}
```

The order may vary because sets are not used as ordered sequences.

---

2. Why Do We Need Sets?

Imagine a website collecting programming languages selected by users:

```
languages = [
    "Python",
    "JavaScript",
    "Python",
    "Java",
```

```
    "JavaScript",
    "Python"
]
```

There are duplicates.

If you only want to know which **unique languages** were selected:

```
unique_languages = set(languages)

print(unique_languages)
```

Now each language appears only once.

This is one of the most common uses of sets.

---

3. Creating a Set

Use curly braces `{}`:

```
languages = {"Python", "JavaScript", "Java"}
```

Check the type:

```
print(type(languages))
```

Output:

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

---

4. Empty Set

There is an important difference between an empty set and an empty dictionary.

This:

```
empty = {}
```

creates an **empty dictionary**, not a set.

To create an empty set:

```
empty = set()
```

Check it:

```
print(type(empty))
```

Output:

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

Remember:

```
{}        # empty dictionary
set()     # empty set
```

---

5. Sets Automatically Remove Duplicates

Example:

```
numbers = {1, 2, 2, 3, 3, 3}

print(numbers)
```

The result contains:

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

A set keeps only one copy of each value.

This makes sets useful for duplicate removal.

---

6. Sets Are Unordered

Lists have positions:

```
languages = ["Python", "JavaScript", "Java"]

print(languages[0])
```

You can use an index because lists are ordered.

Sets do not work this way.

This is invalid:

```
languages = {"Python", "JavaScript", "Java"}

print(languages[0])
```

A set does not support indexing.

The main idea is:

> **A set cares about membership, not position.**

---

7. Sets Do Not Support Indexing

You cannot write:

```
languages[0]
```

or:

```
languages[-1]
```

with a set.

If you need positional access, use a list or tuple.

You can convert a set to a list:

```
languages = {"Python", "JavaScript", "Java"}

languages_list = list(languages)

print(languages_list[0])
```

However, do not rely on that position representing a meaningful original order.

---

8. Sets Are Mutable

Unlike tuples, sets can be modified.

You can add items:

```
languages = {"Python", "JavaScript"}

languages.add("Java")

print(languages)
```

You can also remove items.

```
languages.remove("JavaScript")
```

So:

```
Tuple → immutable
Set   → mutable
```

---

9. `add()`

The `add()` method adds one item to a set.

```
languages = {"Python", "JavaScript"}
```

```
languages.add("Java")

print(languages)
```

The set now contains:

```
{"Python", "JavaScript", "Java"}
```

---

10. Adding a Duplicate

What happens if you add something that already exists?

```
languages = {"Python", "JavaScript"}

languages.add("Python")

print(languages)
```

Nothing is duplicated.

The set still contains only one `"Python"`.

This is one of the main advantages of sets.

---

11. `update()`

`update()` adds multiple values to a set.

```
languages = {"Python", "JavaScript"}

languages.update(["Java", "C++", "Go"])

print(languages)
```

All new unique values are added.

You can pass different iterables:

```
languages.update(("Rust", "Swift"))
```

You can also update from another set:

```
languages.update({"Kotlin", "Dart"})
```

---

12. `add()` vs `update()`

This difference is important.

`add()`

Adds one item:

```
languages.add("Python")
```

`update()`

Adds multiple items from an iterable:

```
languages.update(["Python", "Java", "C++"])
```

Mental model:

```
add()    → add this one item
update() → add these items
```

---

13. `remove()`

`remove()` removes a specific value.

```
languages = {"Python", "JavaScript", "Java"}

languages.remove("JavaScript")

print(languages)
```

The value is removed.

But there is an important rule:

If the value does not exist, `remove()` raises a `KeyError`.

Example:

```
languages.remove("C++")
```

If `"C++"` isn't in the set, Python raises an error.

---

14. `discard()`

`discard()` also removes a value.

The difference is that `discard()` does **not** raise an error if the value does not exist.

```
languages = {"Python", "JavaScript"}

languages.discard("C++")

print(languages)
```

No error occurs.

This makes `discard()` useful when you are not sure whether an item exists.

---

15. `remove()` vs `discard()`

Method	Item exists	Item does not exist
`remove()`	Removes it	Raises `KeyError`
`discard()`	Removes it	Does nothing

Example:

```
languages.remove("Python")
```

Use when the item is expected to exist.

```
languages.discard("Python")
```

Use when you want safe removal without an error.

---

16. `pop()`

Sets also have `pop()`.

But there is an important difference from list `pop()`.

For a list:

```
items.pop(2)
```

removes a specific index.

For a set:

```
items.pop()
```

removes and returns an **arbitrary item**.

Example:

```
languages = {"Python", "JavaScript", "Java"}
```

```
removed = languages.pop()

print(removed)
print(languages)
```

You should not rely on which value gets removed.

A set has no positional index.

---

17. `clear()`

`clear()` removes all values from the set.

```
languages = {"Python", "JavaScript", "Java"}

languages.clear()

print(languages)
```

Output:

```
set()
```

---

18. Checking Membership

One of the biggest reasons to use sets is fast membership checking.

Use:

```
in
```

Example:

```
languages = {"Python", "JavaScript", "Java"}

print("Python" in languages)
```

Output:

```
True
```

And:

```
print("C++" in languages)
```

Output:

```
False
```

You can also use:

```
if "Python" in languages:
    print("Python is available")
```

---

19. Why Sets Are Good for Membership

Imagine a website has thousands of registered usernames.

You want to check whether a username already exists.

A set is often an excellent structure for this:

```
registered_users = {
    "hafsa",
    "asim",
    "ali",
    "sara"
}

username = "hafsa"

if username in registered_users:
    print("Username already exists")
```

The main purpose is not finding the user's position.

It is simply answering:

> Does this value exist?

---

20. Set Operations

Sets become especially powerful when you compare groups of data.

The main set operations are:

```
Union
Intersection
Difference
Symmetric Difference
```

These operations are commonly used in:

- data processing
- databases
- permissions

- analytics
- recommendation systems
- user segmentation
- filtering
- duplicate detection

---

21. Union

Union combines all unique values from two sets.

Suppose:

```
python_students = {"Ali", "Hafsa", "Sara"}
javascript_students = {"Hafsa", "Usman", "Sara"}
```

We want everyone who studies either Python or JavaScript.

Use:

```
all_students = python_students | javascript_students

print(all_students)
```

The result contains every unique student.

You can also use:

```
all_students = python_students.union(javascript_students)
```

Mental model:

```
A ∪ B

Everything in A or B
```

---

22. Intersection

Intersection finds values that exist in **both** sets.

```
python_students = {"Ali", "Hafsa", "Sara"}
javascript_students = {"Hafsa", "Usman", "Sara"}

both = python_students & javascript_students

print(both)
```

Result:

```
{"Hafsa", "Sara"}
```

These students appear in both groups.

You can also use:

```
both = python_students.intersection(javascript_students)
```

Mental model:

```
A ∩ B

Only what both sets share
```

---

23. Difference

Difference finds values that exist in one set but not the other.

```
python_students = {"Ali", "Hafsa", "Sara"}
javascript_students = {"Hafsa", "Usman", "Sara"}

python_only = python_students - javascript_students

print(python_only)
```

Result:

```
{"Ali"}
```

This means:

> Students who are in the Python group but not the JavaScript group.

You can also use:

```
python_only = python_students.difference(javascript_students)
```

---

24. Reverse Difference

Order matters with difference.

```
python_students - javascript_students
```

is not necessarily the same as:

```
javascript_students - python_students
```

Example:

```
print(python_students - javascript_students)
print(javascript_students - python_students)
```

The first gives values unique to Python.

The second gives values unique to JavaScript.

---

25. Symmetric Difference

Symmetric difference finds values that belong to **one set or the other, but not both**.

```
python_students = {"Ali", "Hafsa", "Sara"}
javascript_students = {"Hafsa", "Usman", "Sara"}

different = python_students ^ javascript_students

print(different)
```

Result:

```
{"Ali", "Usman"}
```

These students belong to only one of the two groups.

You can also use:

```
different = python_students.symmetric_difference(
    javascript_students
)
```

Mental model:

```
A ^ B

Everything except the common values
```

---

26. Set Operations Visualized

Imagine two groups:

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

Union

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

Intersection

```
{3}
```

A Difference B

```
{1, 2}
```

B Difference A

```
{4, 5}
```

Symmetric Difference

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

---

27. Set Operation Cheat Sheet

```
A | B
```

Union

```
A & B
```

Intersection

```
A - B
```

Difference

```
A ^ B
```

Symmetric difference

You can also use method versions:

```
A.union(B)
A.intersection(B)
A.difference(B)
A.symmetric_difference(B)
```

---

28. Subsets

A set is a subset of another set if every item in the first set also exists in the second set.

Example:

```
languages = {"Python", "JavaScript", "Java"}

backend = {"Python", "Java"}
```

Check:

```
print(backend.issubset(languages))
```

Output:

```
True
```

You can also use:

```
backend <= languages
```

---

29. Supersets

A set is a superset if it contains every item from another set.

```
languages = {"Python", "JavaScript", "Java"}
backend = {"Python", "Java"}

print(languages.issuperset(backend))
```

Output:

```
True
```

You can also use:

```
languages >= backend
```

---

30. Disjoint Sets

Two sets are disjoint if they have **no values in common**.

Example:

```
frontend = {"HTML", "CSS", "JavaScript"}
databases = {"MySQL", "PostgreSQL", "MongoDB"}

print(frontend.isdisjoint(databases))
```

Output:

```
True
```

If they share something:

```
frontend = {"HTML", "CSS", "JavaScript"}
web = {"JavaScript", "Python"}

print(frontend.isdisjoint(web))
```

Output:

```
False
```

---

31. Converting a List to a Set

This is one of the most useful practical patterns.

```
languages = [
    "Python",
    "JavaScript",
    "Python",
    "Java",
    "JavaScript"
]

unique_languages = set(languages)

print(unique_languages)
```

Duplicates are removed.

This is useful for:

- removing duplicate records
- cleaning imported data
- finding unique categories
- checking unique users
- processing tags

---

32. Converting a Set to a List

Use `list()`:

```
languages = {"Python", "JavaScript", "Java"}

languages_list = list(languages)
```

```
print(languages_list)
```

Remember that sets do not provide a meaningful positional order, so you should not assume the resulting list will be in the order you originally inserted values.

If you need sorted output:

```
languages_list = sorted(languages)
```

---

33. Sets and Loops

You can loop through a set:

```
languages = {"Python", "JavaScript", "Java"}

for language in languages:
    print(language)
```

But do not depend on the iteration order.

If you need predictable sorted output:

```
for language in sorted(languages):
    print(language)
```

---

34. Sets and Duplicate Detection

Suppose a user submits a list of tags:

```
tags = [
    "python",
    "backend",
    "python",
    "api",
    "backend"
]
```

Find unique tags:

```
unique_tags = set(tags)

print(unique_tags)
```

You can also determine whether duplicates existed:

```
if len(tags) != len(set(tags)):
    print("Duplicates found")
```

This is a useful data validation technique.

---

35. Real World Example: haas.dev Resource Categories

Imagine your resources have categories:

```
categories = [
    "Python",
    "JavaScript",
    "AI",
    "Python",
    "Web Development",
    "AI"
]
```

Find unique categories:

```
unique_categories = set(categories)

print(unique_categories)
```

Now the application can display each category only once.

For example:

```
Python
JavaScript
AI
Web Development
```

The exact set iteration order should not be relied upon.

---

36. Real World Example: User Permissions

Sets are useful for permissions.

Suppose a user has:

```
user_permissions = {
    "read",
    "write",
    "comment"
}
```

An administrator requires:

```
required_permissions = {
    "read",
    "write"
}
```

Check whether the user has all required permissions:

```
if required_permissions.issubset(user_permissions):
    print("Access granted")
else:
    print("Access denied")
```

This is much cleaner than manually checking every permission.

---

37. Real World Example: Common Skills

Two developers have different skill sets:

```
hafsa_skills = {
    "Python",
    "JavaScript",
    "Git",
    "SQL"
}

asim_skills = {
    "Python",
    "Git",
    "Docker",
    "Linux"
}
```

Find their common skills:

```
common = hafsa_skills & asim_skills

print(common)
```

Result:

```
{"Python", "Git"}
```

Find skills unique to Hafsa:

```
unique_to_hafsa = hafsa_skills - asim_skills
```

Find all skills:

```
all_skills = hafsa_skills | asim_skills
```

This is a practical example of why set operations matter.

---

38. Frozen Sets

Python also provides:

```
frozenset()
```

A `frozenset` is an immutable version of a set.

Example:

```
languages = frozenset(
    ["Python", "JavaScript", "Java"]
)

print(languages)
```

You cannot use:

```
languages.add("C++")
```

because a frozenset cannot be modified.

Regular set:

```
set → mutable
```

Frozen set:

```
frozenset → immutable
```

For now, focus mainly on regular sets. `frozenset` becomes useful when immutable set-like data is needed.

---

39. What Can a Set Contain?

A set can contain hashable values such as:

```
numbers = {1, 2, 3}
```

```
languages = {"Python", "JavaScript"}
```

```
mixed = {1, "Python", 3.5, True}
```

But a regular set cannot contain a list:

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

This causes an error because lists are mutable and therefore unhashable.

Tuples can be set elements if their contents are hashable:

```
points = {
    (10, 20),
    (30, 40)
}
```

---

40. Set vs List vs Tuple

Feature	**List**	**Tuple**	**Set**
Ordered sequence	Yes	Yes	No
Mutable	Yes	No	Yes
Duplicates	Allowed	Allowed	Not stored
Indexing	Yes	Yes	No
Slicing	Yes	Yes	No
Add items	Yes	No	Yes
Remove items	Yes	No	Yes
Main purpose	Changing sequence	Fixed sequence	Unique values / set operations

Think of them like this:

```
LIST
"I need a collection whose order and contents can change."

TUPLE
"I need a fixed collection of related values."

SET
"I care about unique values and membership."
```

---

41. Common Mistakes

Mistake 1: Using `{}` for an Empty Set

Incorrect:

```
items = {}
```

This creates a dictionary.

Correct:

```
items = set()
```

Mistake 2: Trying to Index a Set

Incorrect:

```
items = {"Python", "Java"}

print(items[0])
```

Sets do not support indexing.

Mistake 3: Expecting Duplicates

```
items = {"Python", "Python", "Java"}
```

There will only be one `"Python"`.

Mistake 4: Confusing `remove()` and `discard()`

```
items.remove("Python")
```

can raise `KeyError` if the value is missing.

```
items.discard("Python")
```

does nothing if the value is missing.

Mistake 5: Assuming Set Order

Do not write code that depends on:

```
items = {"A", "B", "C"}
```

always being iterated as:

```
A
B
C
```

Sets are not designed for positional ordering.

---

42. Set Problem Solving Framework

When you see a problem, ask:

Do I need unique values?

Use:

```
set()
```

Do I need to remove duplicates?

```
set(items)
```

Do I need to check membership?

```
value in my_set
```

Do I need all values from two groups?

```
A | B
```

Do I need common values?

```
A & B
```

Do I need values only in A?

```
A - B
```

Do I need values in either group but not both?

```
A ^ B
```

Do I need to check whether one group is contained in another?

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

This mental model makes set problems much easier to recognize.

---

43. Mini Project: Unique Resource Tags

Create a simple tag manager:

```
tags = [
    "python",
    "backend",
    "python",
    "api",
    "web",
    "backend"
]
```

Convert the list to a set:

```
unique_tags = set(tags)
```

Add a new tag:

```
unique_tags.add("programming")
```

Remove a tag safely:

```
unique_tags.discard("old")
```

Check for a specific tag:

```
if "python" in unique_tags:
    print("Python resources available")
```

Display sorted tags:

```
for tag in sorted(unique_tags):
    print(tag)
```

This small project demonstrates:

- duplicate removal
- adding

- removing
- membership checking
- sorting for display

---

44. 5 Minute Challenge

Create two sets:

```
python_students = {
    "Ali",
    "Hafsa",
    "Sara",
    "Usman"
}

javascript_students = {
    "Hafsa",
    "Sara",
    "Ahmed",
    "Bilal"
}
```

Your program should find:

1. All students.
2. Students learning both languages.
3. Students learning Python only.
4. Students learning JavaScript only.
5. Students learning exactly one of the two languages.
6. Check whether `"Hafsa"` is learning Python.
7. Check whether every Python student is also a JavaScript student.

Use set operators and methods rather than manually comparing each name.

---

45. Practice Exercises

Exercise 1

Create a set containing five programming languages.

Print the set.

Exercise 2

Create a list containing duplicate numbers.

Convert it to a set to remove duplicates.

Exercise 3

Create an empty set correctly.

Verify its type.

Exercise 4

Add three values using `add()`.

Exercise 5

Use `update()` to add multiple values.

Exercise 6

Create two sets and find their union.

Exercise 7

Create two sets and find their intersection.

Exercise 8

Find values that exist in the first set but not the second.

Exercise 9

Find the symmetric difference between two sets.

Exercise 10

Create two sets of developer skills and find their common skills.

Exercise 11

Create a set of permissions and check whether the required permissions are a subset of the user's permissions.

Exercise 12

Create a list of usernames and determine whether duplicate usernames exist.

Hint:

```
len(usernames) != len(set(usernames))
```

---

46. Mini Quiz

Question 1

What is the main feature of a set?

A. It stores values by index.

B. It automatically stores only unique values.

C. It cannot be changed.

D. It stores only strings.

Question 2

How do you create an empty set?

A.

```
{}
```

B.

```
[]
```

C.

```
set()
```

D.

```
()
```

Question 3

Which operator finds common values?

A.

```
A | B
```

B.

```
A & B
```

C.

```
A - B
```

D.

```
A ^ B
```

Question 4

What is the difference between `remove()` and `discard()`?

Question 5

Can you access a set using an index like `my_set[0]`?

Answers

1. B
2. C
3. B
4. `remove()` can raise `KeyError` when the value is missing; `discard()` does not.
5. No. Sets do not support indexing.

---

47. Interview Questions

1. What is a set in Python?
2. Why are sets useful?
3. How are sets different from lists?
4. How are sets different from tuples?
5. How do you create an empty set?
6. Why does `{}` create a dictionary instead of a set?
7. Do sets allow duplicate values?
8. Are sets ordered?
9. Can you access a set using an index?
10. What does `add()` do?
11. What is the difference between `add()` and `update()`?
12. What does `remove()` do?
13. What is the difference between `remove()` and `discard()`?
14. What does `pop()` do on a set?
15. Why can't you specify an index with set `pop()`?
16. What does `clear()` do?
17. What is a union?

18. What is an intersection?
19. What is the difference between `A - B` and `B - A`?
20. What is symmetric difference?
21. What is a subset?
22. What is a superset?
23. What does `isdisjoint()` check?
24. How can you remove duplicates from a list using a set?
25. Why can't a list be stored directly inside a set?
26. What is a `frozenset`?
27. When would you choose a set instead of a list?
28. Why are sets useful for membership testing?
29. Can a tuple be stored inside a set?
30. What is the difference between `set()` and `{}`?

---

48. Set Cheat Sheet

Create

```
languages = {"Python", "JavaScript", "Java"}
```

Empty Set

```
languages = set()
```

Add One

```
languages.add("C++")
```

Add Multiple

```
languages.update(["C++", "Go"])
```

Remove

```
languages.remove("Java")
```

Safe Remove

```
languages.discard("Java")
```

Remove Arbitrary Item

```
languages.pop()
```

Remove Everything

```
languages.clear()
```

Membership

```
"Python" in languages
```

Union

```
A | B
```

or:

```
A.union(B)
```

Intersection

```
A & B
```

or:

```
A.intersection(B)
```

Difference

```
A - B
```

or:

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

Symmetric Difference

```
A ^ B
```

or:

```
A.symmetric_difference(B)
```

Subset

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

Superset

```
A.issuperset(B)
```

Disjoint

```
A.isdisjoint(B)
```

Remove Duplicates

```
unique = set(items)
```

---

49. Key Takeaways

- A set is a collection of **unique values**.
- Sets automatically remove duplicates.
- Sets are mutable.
- Sets do not support indexing or slicing.
- Use `add()` for one item.
- Use `update()` for multiple items.
- `remove()` raises `KeyError` if the value is missing.
- `discard()` safely removes a value without raising an error.
- `pop()` removes an arbitrary item.
- `clear()` removes all items.
- Sets are excellent for membership checking.
- Union combines unique values.
- Intersection finds common values.
- Difference finds values unique to one set.
- Symmetric difference finds values that are unique to either set.
- `issubset()` and `issuperset()` compare groups.
- Sets are extremely useful for duplicate removal and data comparison.
- `frozenset` provides an immutable set.

Final Mental Model

Think of the three data structures you have learned:

```
LIST
[Python, Python, Java]
        ↓
Ordered
Duplicates allowed
```

```
Can change


TUPLE
(Python, Java)
        ↓
Ordered
Duplicates allowed
Cannot change


SET
{Python, Java}
        ↓
Unique values
No positional indexing
Can change
Great for membership and comparisons
```

The simplest rule to remember is:

> **List = sequence**
>
> **Tuple = fixed sequence**
>
> **Set = unique collection**

Once you understand sets, operations such as finding common users, removing duplicate data, comparing permissions, and identifying unique categories become much easier.

Visit haas.dev for more resources and guides.

https://dev-roast-app.vercel.app/resources