Topic 3b: Sets

Unique Collections

Learning Outcomes:

- Understand the purpose of sets (uniqueness, no order)
- Apply set syntax and operations
- Use set mathematical operations (union, intersection, difference)
- Distinguish when to use sets vs lists vs tuples

Prerequisites: Topic 2 (Lists), Topic 3a (Tuples)

Opening Problem

You're building a visitor tracking system.

You need to count unique visitors to a website.

```
visitors = []

def log_visitor(user_id):
    if user_id not in visitors:  # Check every time
        visitors.append(user_id)

log_visitor(101)
log_visitor(102)
log_visitor(101)  # Duplicate — must check before adding
```

This works, but it's inefficient. Checking `if user_id not in visitors` takes longer as the list grows.

The Problem: Manual Duplicate Checking

```
visitors = [101, 102, 103, 101, 102, 104, 101, 105]

# How many unique visitors?
unique = []
for v in visitors:
    if v not in unique:
        unique.append(v)

print(len(unique))  # 5
```

Problems:

- Checking `if v not in unique` scans the entire list
- As data grows, this becomes slow
- Extra code just to handle duplicates

The Solution: Sets

Sets automatically reject duplicates.

```
visitors = {101, 102, 103, 101, 102, 104, 101, 105}
print(visitors)
```

Output:

```
{101, 102, 103, 104, 105}
```

Duplicates are removed automatically. No checking needed.

Sets: Fast Membership Testing

Checking if an element exists is extremely fast:

```
visitors = {101, 102, 103, 104, 105}

print(101 in visitors)  # True — instant lookup
print(999 in visitors)  # False — instant lookup
```

Lists: `in` checks every element (slow for large lists)

Sets: `in` uses hash lookup (fast regardless of size)

Where Sets Fit

Python provides 4 built-in data structures:

Type	Symbol	Mutable?	Ordered?	Duplicates?
List	`[ ]`	Yes	Yes	Yes
Tuple	`( )`	No	Yes	Yes
Set	`{ }`	**Yes**	**No**	**No**
Dictionary	`{ : }`	Yes	Yes	No (keys)

Key difference: Sets have no order and no duplicates.

Sets vs Lists vs Tuples

Need	Use
Ordered collection that can change	List
Ordered collection that cannot change	Tuple
Unique items, order doesn't matter	**Set**

```
# List: ordered, duplicates allowed
items = [1, 2, 2, 3]

# Tuple: ordered, immutable
point = (10, 20)

# Set: unordered, no duplicates
unique = {1, 2, 3}
```

Structure of a Set

```
my_set = {10, 20, 30, "Python", 3.14}
```

Components:

Component	Value
Set name	`my_set`
Elements	`10` , `20` , `30` , `"Python"` , `3.14`
Number of elements	5

Syntax: Curly braces `{ }` with comma-separated elements.

The Empty Set Trap

Creating an empty set is NOT what you expect:

```
empty = {}            # This is a DICTIONARY, not a set!
print(type(empty))    # <class 'dict'>

empty = set()         # This is a set
print(type(empty))    # <class 'set'>
```

Rule: Use `set()` to create an empty set. `{}` creates an empty dictionary.

5 Characteristics of Sets

1. **Unordered** — No indexing, no slicing
2. **Mutable** — Can add/remove elements
3. **No Duplicates** — Automatically removes duplicates
4. **Elements Must Be Immutable** — Can contain strings, numbers, tuples (not lists)
5. **Fast Membership Testing** — `in` is very fast

Characteristic 1: Unordered

Sets do not maintain order. You cannot access by index.

```
colors = {"red", "green", "blue"}
print(colors[0])  # TypeError!
```

Output:

```
TypeError: 'set' object is not subscriptable
```

If you need order, use a list.

Characteristic 2: Mutable

You can add and remove elements:

```
fruits = {"apple", "banana"}

fruits.add("cherry")
print(fruits)  # {'apple', 'banana', 'cherry'}

fruits.remove("banana")
print(fruits)  # {'apple', 'cherry'}
```

Characteristic 3: No Duplicates

Duplicates are automatically removed:

```
numbers = {1, 2, 2, 3, 3, 3, 4}
print(numbers)  # {1, 2, 3, 4}
```

Use case: Removing duplicates from a list

```
data = [1, 2, 2, 3, 3, 3, 4]
unique = list(set(data))
print(unique)  # [1, 2, 3, 4]
```

Characteristic 4: Elements Must Be Immutable

Sets can only contain immutable types:

Allowed	Not Allowed
Numbers (`1` , `3.14`)	Lists `[1, 2]`
Strings (`"hello"`)	Dictionaries `{"a": 1}`
Tuples (`(1, 2)`)	Sets (nested)

Characteristic 4: Why? (Hashing)

Step 1: Adding elements — Python calculates hash for each

```
my_set = {"apple", "banana"}

Element      Hash         Stored at
──────────────────────────────────
"apple"   →  hash = 5   →  slot 5
"banana"  →  hash = 2   →  slot 2
```

Characteristic 4: Why? (Searching)

Step 2: Searching — Python uses hash to find instantly

```
"apple" in my_set?

"apple"  →  hash = 5  →  check slot 5  →  Found!
```

No need to check every element. Just go directly to the slot.

Characteristic 4: Why? (The Problem)

Step 3: What if an element could change?

```
Imagine if lists were allowed:

my_set = {[1, 2]}       # [1, 2] stored at slot 7 (based on [1, 2])

[1, 2] changes to [1, 2, 3]   # Value changed!

"[1, 2, 3]" in my_set?
  →  hash = 9  →  check slot 9  →  Empty!  →  Not found!

But the element IS in the set... just at the wrong slot.
```

This is why Python rejects mutable types in sets.

Immutable types (strings, numbers, tuples) never change, so their address stays correct.

Characteristic 4: Example

Valid:

```
valid = {1, 2.5, "hello", (1, 2)}  # Numbers, strings, tuples
```

Invalid:

```
invalid = {[1, 2]}  # Lists cannot be in sets
```

Output:

```
TypeError: unhashable type: 'list'
```

Solution: Use a tuple instead of a list.

Creating Sets

Method 1: Using curly braces

```
fruits = {"apple", "banana", "cherry"}
```

Method 2: Using `set()` function

```
numbers = set([1, 2, 3, 4, 5])  # From list
chars = set("hello")            # From string: {'h', 'e', 'l', 'o'}
```

Method 3: Empty set

```
empty = set()  # NOT {}
```

Accessing Elements: Loops and `in`

Sets do not support indexing. Use loops or membership testing:

```
colors = {"red", "green", "blue"}

# Loop through elements
for color in colors:
    print(color)

# Check membership
if "red" in colors:
    print("Red exists")
```

Adding Elements: `.add()` and `.update()`

```
fruits = {"apple", "banana"}

# Add single element
fruits.add("cherry")

# Add multiple elements
fruits.update(["date", "elderberry"])

print(fruits)  # {'apple', 'banana', 'cherry', 'date', 'elderberry'}
```

Removing Elements: `.remove()`, `.discard()`, `.pop()`, `.clear()`

Method	Behavior
`.remove(x)`	Remove x. Raises KeyError if not found
`.discard(x)`	Remove x. No error if not found
`.pop()`	Remove and return an arbitrary element
`.clear()`	Remove all elements

Removing Elements: Examples

```
fruits = {"apple", "banana", "cherry", "date"}

# remove() – error if not found
fruits.remove("banana")
# fruits.remove("xyz")  # KeyError!

# discard() – safe, no error
fruits.discard("xyz")  # No error

# pop() – remove arbitrary element
removed = fruits.pop()
print(removed)  # Could be any element

# clear() – remove all
fruits.clear()
print(fruits)  # set()
```

Functions: Same as Lists/Tuples

Function	Example	Result
`len()`	`len({1, 2, 3})`	`3`
`min()`	`min({5, 2, 8})`	`2`
`max()`	`max({5, 2, 8})`	`8`
`sum()`	`sum({1, 2, 3})`	`6`

These work identically on lists, tuples, and sets.

Set Mathematical Operations

Sets support mathematical set operations:

Operation	Symbol	Method	Result
Union	`|`	`.union()`	All elements from both
Intersection	`&`	`.intersection()`	Common elements only
Difference	`-`	`.difference()`	In A but not in B
Symmetric Diff	`^`	`.symmetric_difference()`	In either, not both

Union: Combine All Elements

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

print(A | B)         # {1, 2, 3, 4, 5}
print(A.union(B))    # {1, 2, 3, 4, 5}
```

Use case: Combining user lists, merging data sources

Intersection: Common Elements

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

print(A & B)                # {3, 4}
print(A.intersection(B))    # {3, 4}
```

Use case: Finding users who visited both pages, common skills

Difference: In A but Not in B

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

print(A - B)              # {1, 2}
print(A.difference(B))    # {1, 2}
```

Use case: Finding new users, items not yet processed

Symmetric Difference: In Either, Not Both

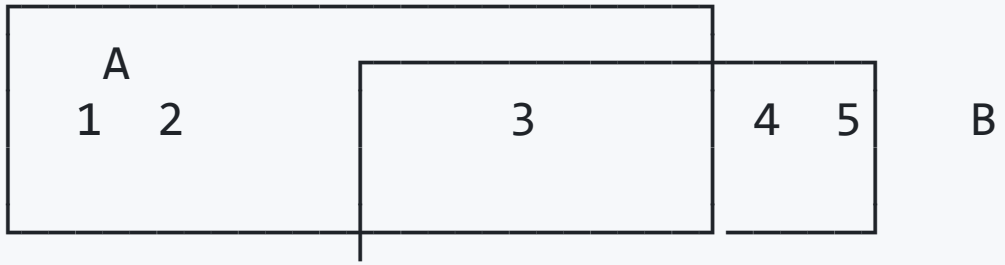
```
A = {1, 2, 3}
B = {3, 4, 5}

print(A ^ B)                          # {1, 2, 4, 5}
print(A.symmetric_difference(B))      # {1, 2, 4, 5}
```

Element 3 is in both, so it's excluded.

Visual: Set Operations

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




```
Union (A | B) = {1, 2, 3, 4, 5}
A - B = {1, 2}
B - A = {4, 5}
A ^ B = {1, 2, 4, 5}
```

Common Mistakes

1. Using `{}` for empty set:

```
empty = {}        # Wrong: creates dict
empty = set()     # Correct: creates set
```

2. Trying to index:

```
my_set = {1, 2, 3}
print(my_set[0])  # TypeError!
```

Common Mistakes (Continued)

3. Adding mutable elements:

```
my_set = set()
my_set.add([1, 2])  # TypeError: list is mutable
my_set.add((1, 2))  # Correct: tuple is immutable
```

4. Using `.remove()` without checking:

```
my_set.remove(99)   # KeyError if 99 not in set
my_set.discard(99)  # Safe: no error
```

Summary

What we covered:

1. **Sets store unique elements** — duplicates auto-removed
2. **Sets are unordered** — no indexing
3. **Syntax:** Curly braces `{}`, empty set with `set()`
4. **CRUD:** `.add()`, `.remove()`, `.discard()`, `.update()`
5. **Mathematical operations:** `|`, `&`, `-`, `^`
6. **Fast membership:** `in` is very efficient

Sets vs Tuples: Quick Comparison

Feature	Tuple	Set
Symbol	`( )`	`{ }`
Ordered	Yes	No
Mutable	No	Yes
Duplicates	Allowed	Not allowed
Indexing	Yes	No
Use case	Fixed grouped data	Unique collections

What's Next

Topic 3c: Dictionaries

- Sets store unique keys only
- Dictionaries store key-value pairs

```
# Set: just keys
users = {"alice", "bob", "charlie"}

# Dictionary: keys with values
users = {"alice": 25, "bob": 30, "charlie": 28}
```

Preview: Dictionaries let you associate data with each key.