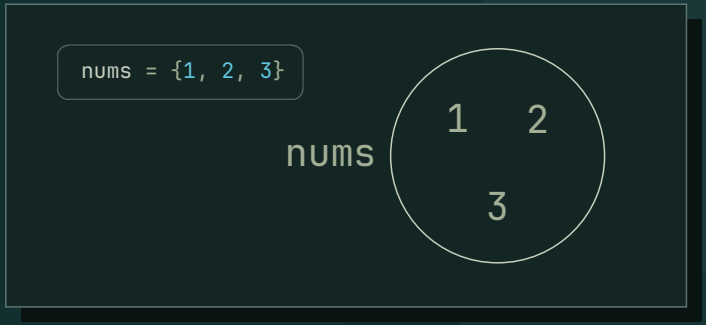


Sets

Data Structures – Intermediate Python

Agenda

- What Are Sets?
  - Properties
  - Benefits
  - Drawbacks
- Set Details
  - Operations
  - Implementation Details
- Working with Sets
  - Math and Syntax
  - Frozen Sets

What Are Sets?

What Is a Set?

Sets are Python's implementation of a data abstraction known as a *set*, which is a collection of distinct items

- Sets represent mathematical sets
- A set with no elements is called the *empty set* and is represented by the symbol ∅
- Sets with single elements are known as *singletons*
- Sets cannot contain themselves

Set Properties

- Items are distinct
- Items are unordered
- Item membership can be checked, but individual items cannot be accessed.
- Items can iterated over
- Set operations include mathematical operations

Benefits of Sets

- Benefits of distinct items
  - Efficiency of certain operations can be increased since the code won't need to worry about possible duplicate elements
  - Sets can be used as a way to deal with duplicate elements in other collections
- Benefits of mathematical operations
  - Sets can be used in mathematical contexts
  - Mathematical algorithms involving sets can be simpler to implement since sets exist in programming languages

Drawbacks of Sets

- Sets aren't contiguous
  - Sets can be slower to use than other data abstractions
  - Sets can be slower to iterate over than other data abstractions
- Sets are unordered
  - If order is important, sets can't be considered, even if their operations are useful for this particular problem

Set Details

Set Operations

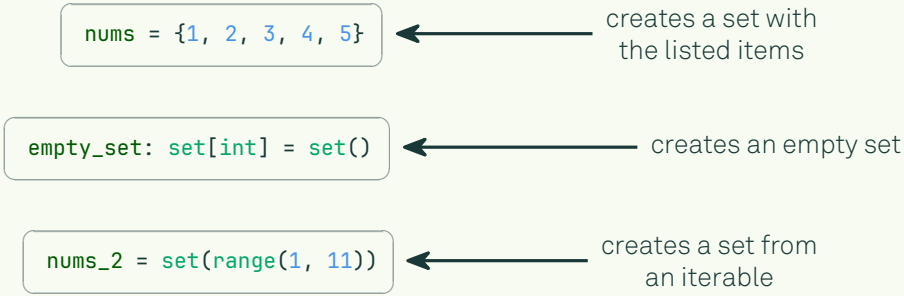
- What operations do sets have?
  - Cardinality (constant time)
  - Membership (time depends on implementation)
  - Add (time depends on implementation)
  - Remove (time depends on implementation)
  - Comparisons to other sets (linear time)
  - Union (linear time)
  - Intersection (linear time)
  - Difference (linear time)
  - Symmetric difference (linear time)

Set Implementation Details

- Python has two implementations of set
- `set`
  - Implemented using a *hash table*
  - All items must be hashable
  - Sets are not hashable
  - Time for membership operation is constant time
  - Time for add and remove operations is amortized constant time
- `frozenset`
  - Implementation is the same as set, but without methods that would mutate the set
  - Frozen sets are hashable

Working with Sets

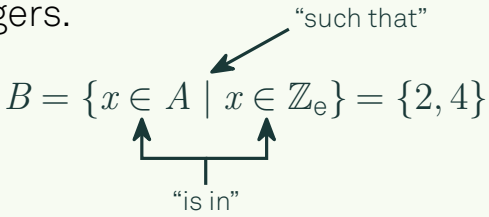
Set Syntax: Creation



Set Syntax: Comprehension

Let $\mathbb{Z}_e$ be the set of even integers.

$$A = \{1, 2, 3, 4, 5\}$$




```
a = {1, 2, 3, 4, 5}
b = {x for x in a if x % 2 == 0}
print(b)
```

```
{2, 4}
```

Set Syntax: Adding

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

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

```
5  a.add(1)
6  print(a)
```

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

Process finished with exit code 0
```

Set Syntax: Cardinality

- The *cardinality* of a set is the number of elements in the set

$$A = \{1, 2, 3, 4, 5\} \qquad |A| = 5$$

```
a = {1, 2, 3, 4, 5}
print(f"|a| = {len(a)}")
```

```
|a| = 5
```

Set Syntax: Membership

- *Membership* of a set means that an element is part of a set

$$A = \{1, 2, 3, 4, 5\} \qquad 3 \in A \qquad 7 \notin A$$

```
a = {1, 2, 3, 4, 5}
print(f"3 in a: {3 in a}")
print(f"7 in a: {7 not in a}")
```

```
3 in a: True
7 not in a: True
```

Set Syntax: Removing

```
1  a = {1, 2, 3, 4, 5}
2  print(a)
3  a.remove(1)
4  print(a)
```

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

```
5  a.remove(7)
```

```
Traceback (most recent call last):
  File "project/src/main.py", line 5, in <module>
    a.remove(7)
    ~~~~~~~~^^^
KeyError: 7
```

Set Syntax: Removing

```python
a = {1, 2, 3, 4, 5}
print(a)
print(a.pop())
print(a)
a.clear()
print(a)
```

```
{1, 2, 3, 4, 5}
1
{2, 3, 4, 5}
set()
```

```python
print(a.pop())
```

```
Traceback (most recent call last):
  File "project/src/main.py", line 7, in <module>
    print(a.pop())
          ~~~~~^^
KeyError: 'pop from an empty set'
```

Set Syntax: Removing

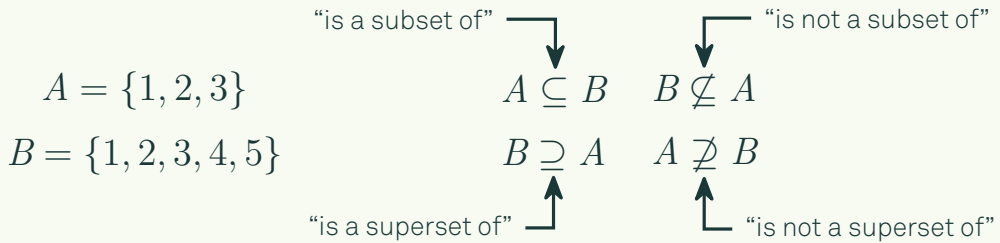
```
a = {1, 2, 3, 4, 5}
print(a)
a.discard(1)
print(a)
a.discard(7)
print(a)
```

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

Process finished with exit code 0
```

Set Syntax: Comparing

- Set A is a *subset* of set B if B contains all the elements of A
- If A is a subset of B, then B is a *superset* of A



- $\varnothing$ is a subset of *every set*

Set Syntax: Comparing

```
a = {1, 2, 3}
b = {1, 2, 3, 4, 5}
print(f"a is a subset of b (method): {a.issubset(b)}")
print(f"a is a subset of b (operator): {a ⩽ b}")
print(f"b is a superset of a (method): {b.issuperset(a)}")
print(f"b is a superset of a (operator): {b ⩾ a}")
```

```
a is a subset of b (method): True
a is a subset of b (operator): True
b is a superset of a (method): True
b is a superset of a (operator): True
```

Why Two Ways?

- Operator version must compare two sets
- Method version compares a set with anything iterable

```
a = {1, 2, 3}
lst = [1, 2, 3, 4, 5]
print(a.issubset(lst))
# print(a <= lst) # TypeError: comparison must be between two sets
```

```
True
```

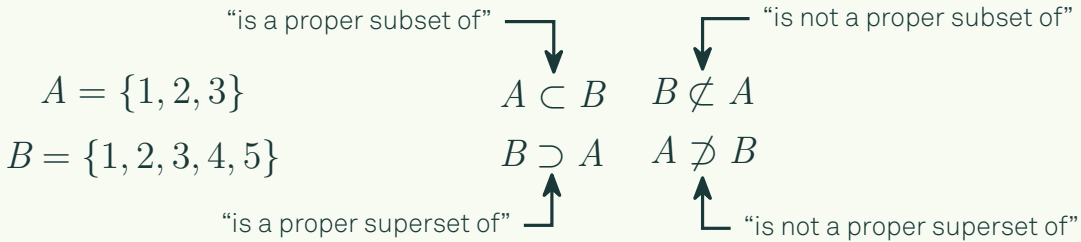
- This is the case for all set comparisons and operations

Why Two Ways?

- The operator version is preferred since it's short and simple
- Forcing the operation version to work with only sets prevents error-prone code caused by the other uses of these operators

Set Syntax: Comparing

- Set A is a *proper subset* of set B if $A \subseteq B$ and $A \neq B$
- If A is a proper subset of B, then B is a *proper superset* of A



Set Syntax: Comparing

```
a = {1, 2, 3}
b = {1, 2, 3, 4, 5}
print(f"a is a proper subset of b: {a < b}")
print(f"b is a proper superset of a: {b > a}")
```

```
a is a proper subset of b: True
b is a proper superset of a: True
```

Set Syntax: Comparing

- Two sets are *disjoint* if they contain no common elements

```
a = {1, 2, 3}
b = {4, 5, 6}
print(a.isdisjoint(b))
```

```
True
```

Set Syntax: Set Operations

- The *union* of sets A and B is a set containing the elements of A and the elements of B

$A = \{1, 2, 3\}$ $B = \{1, 2, 5, 8\}$ $A \cup B = \{1, 2, 3, 5, 8\}$

```
a = {1, 2, 3}
b = {1, 2, 5, 8}
print(a.union(b))
print(a | b)  # operator read as "or"
```

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

Another Example of Why Two Ways

```
nums = {random.randint(1, 100) for _ in range(10)}
print(nums)
# This is clearer than `nums & range(1, 11)`.
if nums.intersection(range(1, 11)):
    print("`nums` has values between 1 and 10")
else:
    print("`nums` doesn't have any values between 1 and 10")
```

```
{66, 70, 8, 43, 76, 13, 88, 26, 94, 31}
`nums` has values between 1 and 10
```

Set Syntax: Set Operations

- The previous union operations didn't modify the sets. We can use the following union operations to modify a set

```
a = {1, 2, 3}
b = {1, 2, 5, 8}
a.update(b)
print(a)
a = {1, 2, 3}
a |= b
print(a)
```

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

Set Syntax: Set Operations

- The *intersection* of sets A and B is a set containing the elements of A that are also elements of B

$A = \{1, 2, 3\}$ $B = \{1, 2, 5, 8\}$ $A \cap B = \{1, 2\}$

```
a = {1, 2, 3}
b = {1, 2, 5, 8}
print(a.intersection(b))
print(a & b)  # operator read as "and"
```

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

Set Syntax: Set Operations

- The previous intersection operations didn't modify the sets. We can use the following intersection operations to modify a set

```
a = {1, 2, 3}
b = {1, 2, 5, 8}
a.intersection_update(b)
print(a)
a = {1, 2, 3}
a &= b
print(a)
```

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

Set Syntax: Set Operations

- The *difference* of sets A and B is a set containing the elements of A that are not elements of B

$$A = \{1, 2, 3\} \quad B = \{1, 2, 5, 8\} \qquad A \setminus B = \{3\} \quad B \setminus A = \{5, 8\}$$

```
a = {1, 2, 3}
b = {1, 2, 5, 8}
print(a.difference(b))
print(b.difference(a))
print(a - b)
print(b - a)
```

```
{3}
{8, 5}
{3}
{8, 5}
```

Set Syntax: Set Operations

- The previous difference operations didn't modify the sets. We can use the following difference operations to modify a set

```
a = {1, 2, 3}
b = {1, 2, 5, 8}
a.difference_update(b)
print(a)
a = {1, 2, 3}
b -= a
print(b)
```

```
{3}
{5, 8}
```

Set Syntax: Set Operations

- The *symmetric difference* of sets A and B is a set containing the elements that are unique to A and unique to B

$$A = \{1, 2, 3\} \quad B = \{1, 2, 5, 8\}$$

$$A \oplus B = (A \cup B) \setminus (A \cap B) = \{3, 5, 8\}$$

```
a = {1, 2, 3}
b = {1, 2, 5, 8}
print(a.symmetric_difference(b))
print(a ^ b)
```

```
{3, 5, 8}
{3, 5, 8}
```

Set Syntax: Set Operations

- The previous symmetric difference operations didn't modify the sets. We can use the following symmetric difference operations to modify a set

```
a = {1, 2, 3}
b = {1, 2, 5, 8}
a.symmetric_difference_update(b)
print(a)
a = {1, 2, 3}
a ^= b
print(a)
```

```
{3, 5, 8}
{3, 5, 8}
```

Set Syntax: Frozen Set

- Frozen sets are created using the **`frozenset`** function

```
a = frozenset((1, 2, 3))
print(a)
b = frozenset(['a', 'b', 'c'])
print(b)
unique_letters = frozenset('hello')
print(unique_letters)
```

```
frozenset({1, 2, 3})
frozenset({'c', 'b', 'a'})
frozenset({'o', 'e', 'l', 'h'})
```

Set Syntax: Frozen Set

- Frozen sets can be stored in other sets, unlike regular sets

```
a = {frozenset((1, 2, 3)), frozenset((4, 5, 6))}
print(a)
```

```
{frozenset({1, 2, 3}), frozenset({4, 5, 6})}
```

Set Syntax: Frozen Set

- Frozen sets can do everything regular sets can do, but they can't be modified
- We can, however, still use all the set operators

```
a = frozenset((1, 2, 3))
b = frozenset((1, 5, 6))
a |= b
print(a)
```

```
frozenset({1, 2, 3, 5, 6})
```

Set Syntax: Mixing Set Types

- We can use both sets and frozen sets together, but we have to be careful in some places

```
a = frozenset((1, 2, 3))
b = {1, 4, 7}
print(a < b)
print(a | b)  # returns a frozen set
print(b | a)  # returns a set
```

```
False
frozenset({1, 2, 3, 4, 7})
{1, 2, 3, 4, 7}
```

Any Questions?