Sets!

A `set` contains unique, unordered objects

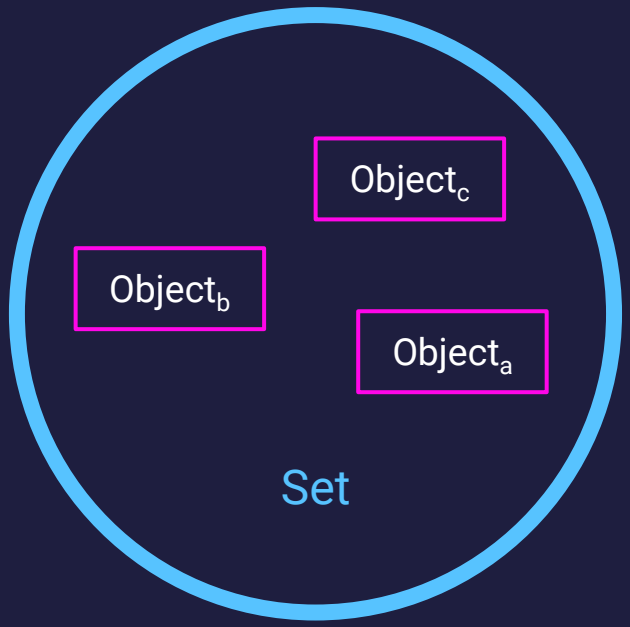
- A `Set` is a **container** data type, like a `List`

- Duplicate objects are not allowed in a Set
  - Adding the same object more than once is *idempotent*, meaning it has no effect.

- Objects do not have a specific ordering in the Set
  - An object is either a member of the set or it isn't
  - At times objects may *appear* ordered, but you cannot assume this!

- Sets are ***mutable*** objects, like Lists
  - You can `add` and `remove` objects from a Set

- Testing *membership* in a Set *significantly more efficient* than in a List

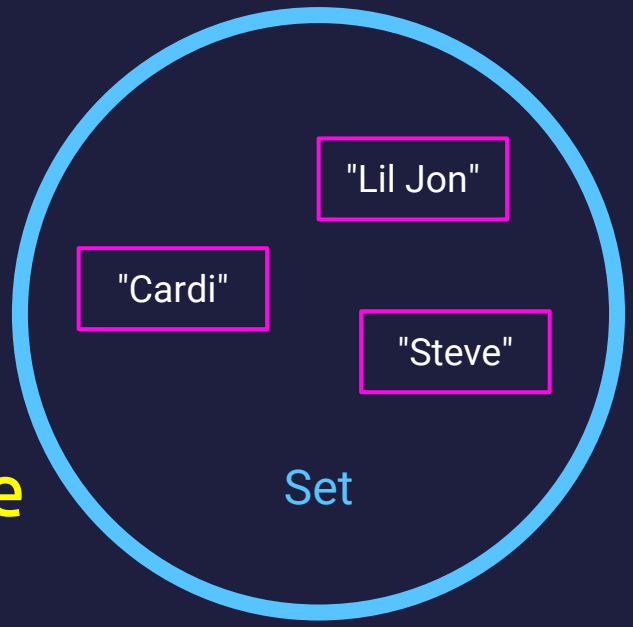


Constructing a `set`

- The `Set` type is defined in the `typing` package
  - `from typing import Set`

- Set Literal Notation surrounds objects in curly braces

```
names: Set[str] = {"Cardi B", "Lil Jon", "Steve Carell"}
```



- The `set` function constructs a `Set` from an `iterable`

```
colors: Set[str] = set(["red", "green", "blue"])
odds: Set[int] = set(range(1, 10, 2))
```

Fundamental Operations on a `set`

- Objects can be `add`ed to a set

```
names: Set[str] = {"Cardi B", "Lil Jon", "Steve Carell"}
names.add("Kaki")
```

- Objects can be `remove`d from a set

```
colors: Set[str] = set(["red", "green", "blue"])
colors.remove("red")
```

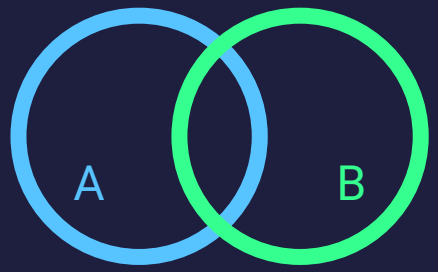
- Test whether an object is `in` a set

```
odds: Set[int] = set(range(1, 10, 2))
1 in odds   # Evaluates to True
2 in odds   # Evaluates to False
```

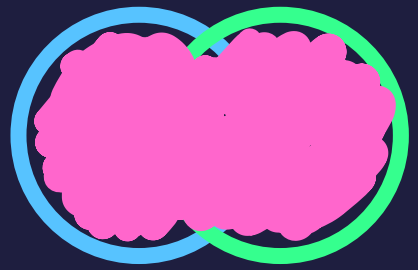
Set Operations

- The Set data structure is inspired by its namesake in discrete math
- Often taught and visualized in terms of Venn diagrams
- Common set operations:
  1. Union
  2. Intersection
  3. Difference

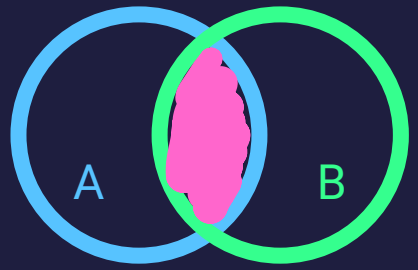
Consider two Sets



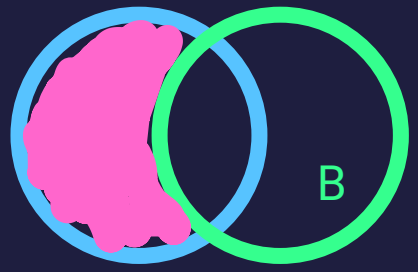
Union



Intersection



Difference
(A - B)



Union

- The union method returns a new set with *every* object included from its operands

```
odds: Set[int] = set(range(1, 10, 2))
evens: Set[int] = set(range(0, 10, 2))
odds.union(evens)   # Evaluates to {1, 2, 3, 4, 5, 6, 7, 8, 9}
evens.union(odds)   # Evaluates to {1, 2, 3, 4, 5, 6, 7, 8, 9}
```

- Python's set type also *overloads* the `|` operator to result in union:

```
odds | evens   # Evaluates to {1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Intersection

- The intersection method returns a set of only the objects shared by **both** of its operands

```
one_to_seven: Set[int] = set(range(1, 8))
five_to_ten: Set[int] = set(range(5, 11))
one_to_seven.intersection(five_to_ten)   # Evaluates to {5, 6, 7}
```

- Python's set type also *overloads* the `&` operator to result in intersection:

```
one_to_seven & five_to_ten   # Evalutes to {5, 6, 7}
```

Difference

- The difference method returns a set of the left-hand set's objects excluding any in the right-hand set's

```
one_to_seven: Set[int] = set(range(1, 8))
five_to_ten: Set[int] = set(range(5, 11))
one_to_seven.difference(five_to_ten)    # Evaluates to {1, 2, 3, 4}
five_to_ten.difference(one_to_seven)    # Evaluates to {8, 9, 10}
```

- Python's set type also *overloads* the `-` operator to result in difference:

```
one_to_seven - five_to_ten  # Evalutes to {5, 6, 7}
five_to_ten - one_to_seven  # Evalutes to {5, 6, 7}
```

Boolean Relationships between Sets

- `issubset` - are each of `set_a`'s members *in* `set_b`?
  - Method: `set_a.issubset(set_b)`
  - Operator overload: `set_a <= set_b`

- `issuperset` - are each of `set_b`'s members *in* `set_a`?
  - Method: `set_a.issuperset(set_b)`
  - Operator overload: `set_a >= set_b`

- `isdisjoint` - do `set_a` and `set_b` share *no* objects in common?
  - Method: `set_a.isdisjoint(set_b)`