TABLE OF CONTENTS

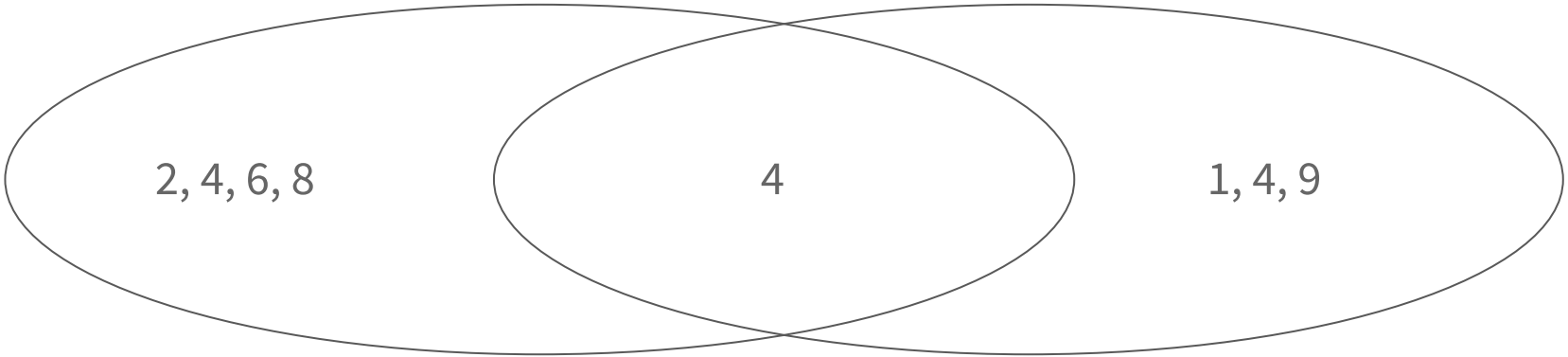
- Section 1: Set Introduction
  a. What is a Set?
  b. Immutable vs. Hashable
  c. Defining a Set in Python
- Section 2: Operating on a Set
  a. Length, Membership and Iteration
  b. Union, Intersection and Difference
  c. Disjoint, Subset and Superset

TABLE OF CONTENTS

- Section 3: Modifying a Set
  a. Add, Remove, Discard, Pop, Clear
  b. Different Update Methods
  c. Augmented Assignment
- Section 4: Frozen Sets and Conclusion
  a. Frozen Sets
  b. Augmented Assignment
  c. Why Sets?
  d. Set Speed Test

What is a Set?

- In mathematics, a **set** is a well-defined collection of distinct objects
  - Set of even positive numbers less than 10: 2, 4, 6, 8
  - Set of perfect squares less than 10: 1, 4, 9
  - Intersection of these two sets: the number 4

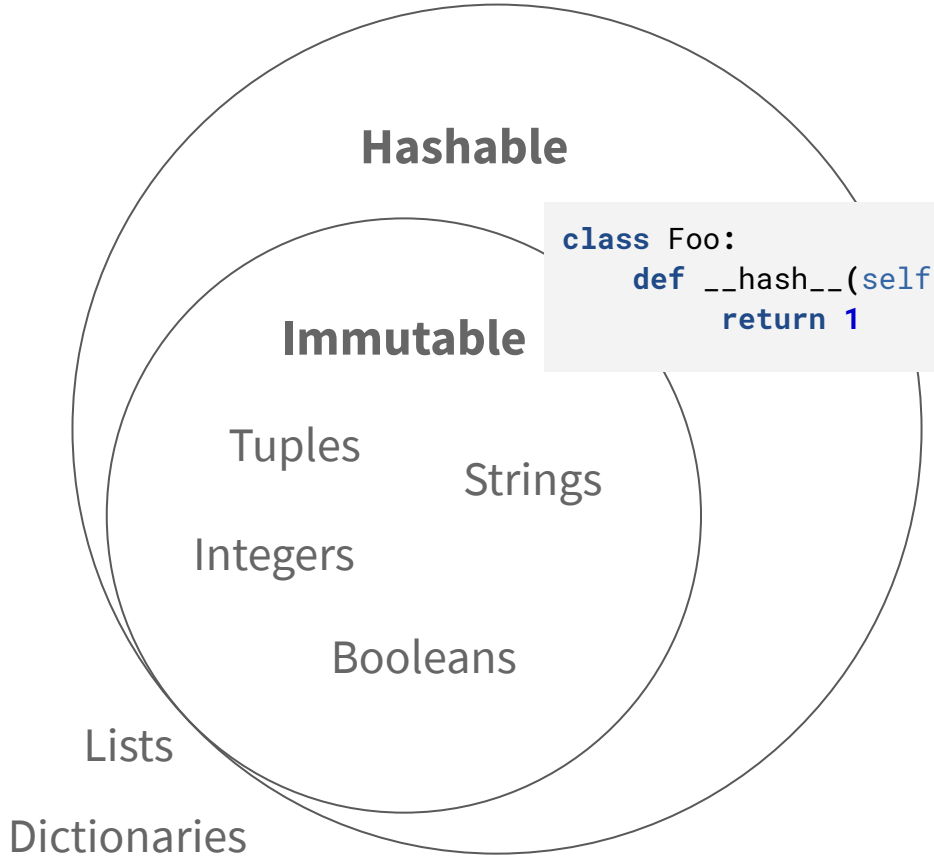


What is a Set in Python?

- Python's built-in **`set`** type has the following characteristics:
  - Sets are unordered
  - Set elements are unique and duplicate elements are not allowed
  - A set itself may be modified, but the elements contained in the set must be hashable

Immutable vs Hashable

- Immutable - A type of object that cannot be modified after it was created.
- Hashable - A type of object that you can call **hash()** on.
- All immutable objects are hashable, but not all hashable objects are immutable.
- Python Sets can only include hashable objects.




```
class Foo:
    def __hash__(self):
        return 1
```

Immutable vs Hashable - Live Coding

Defining a Set in Python - Live Coding

Operating on a Set - Length

- `len(x)`
  - Computes the length of a set
  - Argument need to be an iterable (sets are iterables)

```
>>> a = {1, 1, 2}
>>> len(a)
2

>>> a
{1, 2}

>>> len(set())
0

>>> b = {(1, 2), 2}
>>> len(b)
2
```

Operating on a Set - Membership

- `<elem> in x`
  - Returns a boolean which indicates if an element exists in a set
  - x has to be an iterable (sets are iterables)

```
>>> x = {'foo', 'bar'}
>>> 'foo' in x
True

>>> 'baz' in x
False

>>> 'baz' not in x
True

>>> not 'baz' in x
True
```

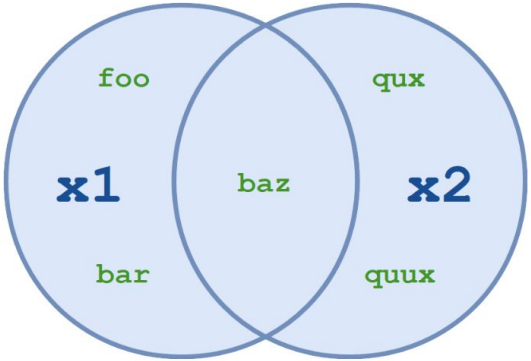
Operating on a Set - Iteration

- You can iterate over a set using the same syntax as you would a list or tuple
  - Note you cannot slice or index sets (they're not ordered)

```
>>> x = {'foo', 'bar', 'baz'}
>>> for elem in x:
...     print(elem)
baz
foo
bar
>>> x[0]
TypeError: 'set' object is not subscriptable
>>> x[1:]
TypeError: 'set' object is not subscriptable
```

Operating on a Set - Union

- Union of multiple sets is the set of all the elements in all sets
  - `x1.union(x2[, x3 ...])`
    - Arguments need to be iterables
    - This is called a method
  - `x1 | x2 [| x3 ...]`
    - Operands need to be sets
    - This is called a operator



Set Union

Operating on a Set - Union

```
>>> x1 = {'foo', 'bar', 'baz'}
>>> x2 = {'baz', 'qux', 'quux'}

>>> x1 | x2
{'bar', 'baz', 'foo', 'quux', 'qux'}

>>> x1.union(x2)
{'bar', 'baz', 'foo', 'quux', 'qux'}

>>> x1.union(('baz', 'qux', 'quux'))
{'bar', 'baz', 'foo', 'quux', 'qux'}

>>> x1 | ('baz', 'qux', 'quux')
TypeError: unsupported operand type(s) for |: 'set' and 'tuple'
```

Operating on a Set - Union

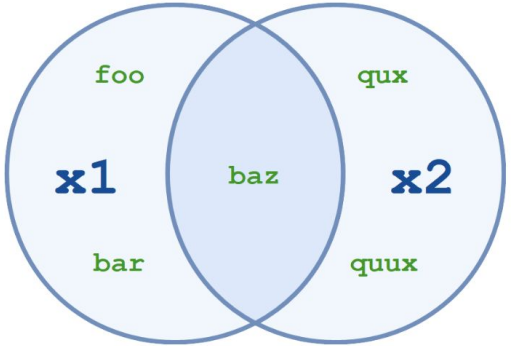
```
>>> a = {1, 2, 3, 4}
>>> b = {2, 3, 4, 5}
>>> c = {3, 4, 5, 6}
>>> d = {4, 5, 6, 7}

>>> a.union(b, c, d)
{1, 2, 3, 4, 5, 6, 7}

>>> a | b | c | d
{1, 2, 3, 4, 5, 6, 7}
```

Operating on a Set - Intersection

- The intersection of multiple sets is the set of only the elements that exist in all sets
  - `x1.intersection(x2[, x3 ...])`
    - Arguments need to be iterables
  - `x1 & x2 [& x3 ...]`
    - Operands need to be sets



Set Intersection

Operating on a Set - Intersection

```
>>> x1 = {'foo', 'bar', 'baz'}
>>> x2 = {'baz', 'qux', 'quux'}

>>> x1.intersection(x2)
{'baz'}

>>> x1 & x2
{'baz'}
```

Operating on a Set - Intersection

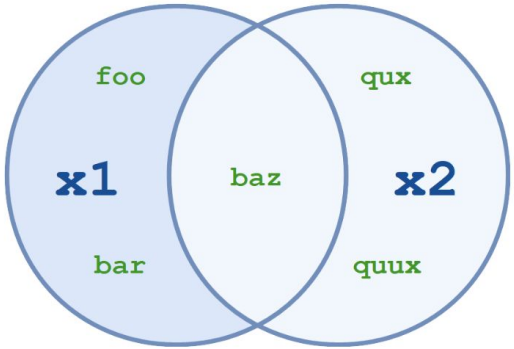
```
>>> a = {1, 2, 3, 4}
>>> b = {2, 3, 4, 5}
>>> c = {3, 4, 5, 6}
>>> d = {4, 5, 6, 7}

>>> a.intersection(b, c, d)
{4}

>>> a & b & c & d
{4}
```

Operating on a Set - Difference

- The difference of multiple sets is the set of only the elements that exist in the first set but not in any of the rest
  - `x1.difference(x2[, x3 ...])`
    - Arguments need to be iterables
  - `x1 - x2 [- x3 ...]`
    - Operands need to be sets



Set Difference

Operating on a Set - Difference

```
>>> x1 = {'foo', 'bar', 'baz'}
>>> x2 = {'baz', 'qux', 'quux'}

>>> x1.difference(x2)
{'foo', 'bar'}

>>> x1 - x2
{'foo', 'bar'}
```

Operating on a Set - Difference

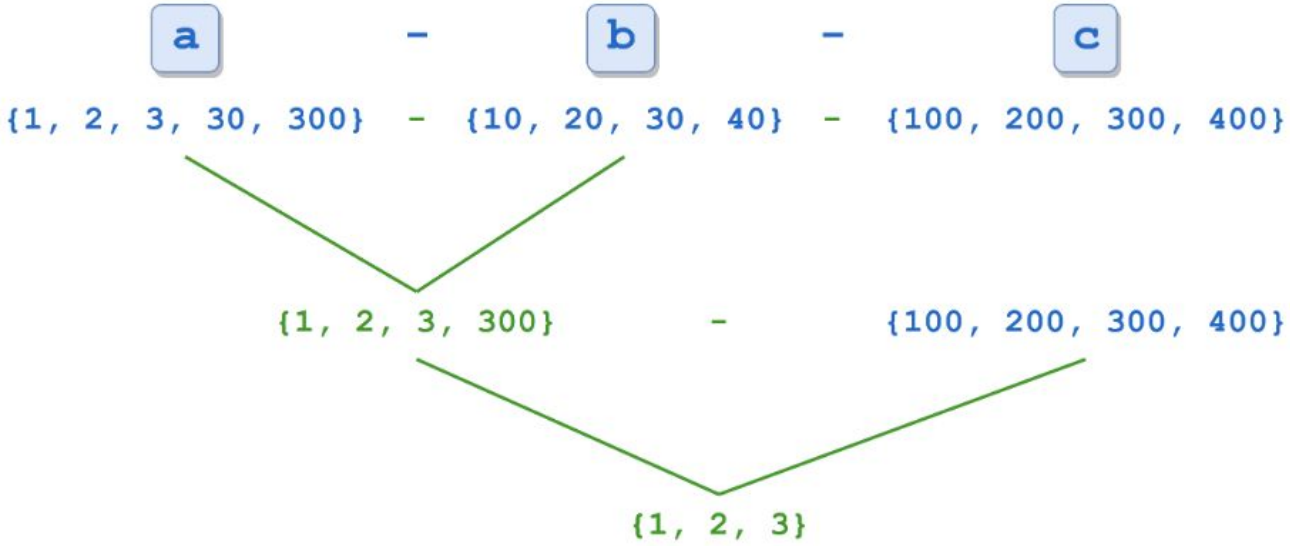
```
>>> a = {1, 2, 3, 30, 300}
>>> b = {10, 20, 30, 40}
>>> c = {100, 200, 300, 400}

>>> a.difference(b, c)
{1, 2, 3}

>>> a - b - c
{1, 2, 3}
```

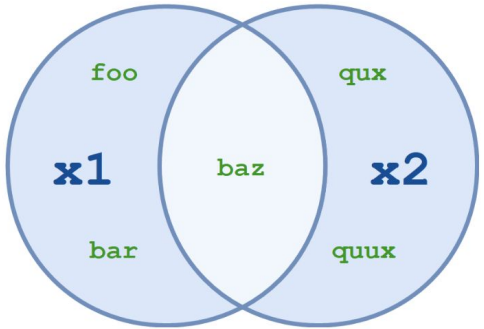
Operating on a Set - Difference

- This operation is evaluated left to right



Operating on a Set - Symmetric Difference

- The symmetric difference of multiple sets is the set of only the elements that exist in a single set, but not in multiple
  - `x1.symmetric_difference(x2)`
    - Argument need to be iterable
    - Only works with one argument
  - `x1 ^ x2 [^ x3 ...]`
    - Operands need to be sets
    - Works with multiple sets



Set Symmetric Difference

Operating on a Set - Symmetric Difference

```
>>> x1 = {'foo', 'bar', 'baz'}
>>> x2 = {'baz', 'qux', 'quux'}

>>> x1.symmetric_difference(x2)
{'foo', 'qux', 'quux', 'bar'}

>>> x1 ^ x2
{'foo', 'qux', 'quux', 'bar'}

>>> x3 = {'bar', 'baz'}

>>> x1.symmetric_difference(x2, x3)
TypeError: symmetric_difference() takes exactly one argument (2 given)
```

Operating on a Set - Symmetric Difference

```
>>> a = {1, 2, 3, 4, 5}
>>> b = {10, 2, 3, 4, 50}
>>> c = {1, 50, 100}

>>> a ^ b ^ c
{100, 5, 10}
```

Operating on a Set - Is Disjoint

- Determines whether or not two sets have any elements in common.
  - `x1.isdisjoint(x2)`
    - Only works with one argument (comparing two sets)
      - Argument needs to be iterable
  - No corresponding operator

Operating on a Set - Is Disjoint

```
>>> x1 = {'foo', 'bar', 'baz'}
>>> x2 = {'baz', 'qux', 'quux'}

>>> x1.isdisjoint(x2)
False

>>> x2 - {'baz'}
{'quux', 'qux'}

>>> x1.isdisjoint(x2 - {'baz'})
True
```

Operating on a Set - Is Disjoint

```
>>> x1 = {1, 3, 5}
>>> x2 = {2, 4, 6}
>>> x1.isdisjoint(x2)
True
>>> x1 & x2 # if x1 and x2 are disjoint, intersection is empty set
set()
```

Operating on a Set - Is Subset

- A set is considered a subset of another set if every element of the first set is in the second
  - `x1.issubset(x2)`
    - Argument need to be an iterable
  - `x1 <= x2 [<= x3 ...]`
    - Operands need to be sets
    - Works with multiple sets and compares if each set is a subset of all the rest of the sets to the right

Operating on a Set - Is Subset

```
>>> x = {'foo', 'bar'}
>>> x.issubset({'foo', 'bar', 'baz'})
True

>>> x.issubset(x)
True
```

Operating on a Set - Is Subset

```
>>> a = {1}
>>> b = {1, 2}
>>> c = {1, 2, 3}
>>> d = {1, 2, 4}
>>> a <= b
True
>>> b <= c
True
>>> a <= b <= c
True
>>> a <= b <= d
True
>>> a <= c <= d
False
```

Operating on a Set - Is Proper Subset

- Proper subset is the same as subset except sets can't be identical
  - No corresponding method
  - `x1 < x2 [< x3 ...]`
    - Operands need to be sets
    - Works with multiple sets and compares if each set is a proper subset of all the rest of the sets to the right
  - `x1 < x1` would return False while `x1 <= x1` would return True.

Operating on a Set - Is Superset

- A set is considered a superset of another set if the first set contains every element of the second set
  - `x1.issuperset(x2)`
    - Argument need to be an iterable
  - `x1 >= x2 [>= x3 ...]`
    - Operands need to be sets
    - Works with multiple sets and compares if each set is a superset of all the rest of the sets to the right

Operating on a Set - Is Superset

```
>>> x = {'foo', 'bar', 'baz'}
>>> x.issuperset({'foo', 'bar'})
True

>>> x.issuperset(x)
True
```

Operating on a Set - Is Superset

```
>>> a = {1}
>>> b = {1, 2}
>>> c = {1, 2, 3}
>>> d = {1, 2, 4}
>>> b >= a
True
>>> c >= b
True
>>> c >= b >= a
True
>>> d >= b >= a
True
>>> d >= c >= a
False
```

Operating on a Set - Is Proper Superset

- Proper superset is the same as superset except sets can't be identical
  - No corresponding method
  - `x1 > x2 [> x3 ...]`
    - Operands need to be sets
    - Works with multiple sets and compares if each set is a proper superset of all the rest of the sets to the right
  - `x1 > x1` would return False while `x1 >= x1` would return True.

Modifying a Set - Add

- `x.add(<elem>)`
  - Adds an element to a set.
  - `<elem>` - must be hashable or else throws error

```
>>> x = {'foo', 'bar', 'baz'}

>>> x.add('qux')
>>> x
{'bar', 'baz', 'foo', 'qux'}

>>> x.add({'quix'})
TypeError: unhashable type: 'set'
```

Modifying a Set - Remove

- Removes an element to a set.
  - `x.remove(<elem>)`
    - `<elem>` must exist in set or else throws error

```
>>> x = {'foo', 'bar', 'baz'}

>>> x.remove('baz')
>>> x
{'bar', 'foo'}

>>> x.remove('qux')
KeyError: 'qux'
```

Modifying a Set - Discard

- Removes an element to a set.
  - `x.discard(<elem>)`
    - Removes `<elem>` from set. If `<elem>` does not exist, do nothing.

```
>>> x = {'foo', 'bar', 'baz'}

>>> x.discard('baz')
>>> x
{'bar', 'foo'}

>>> x.discard('qux')
>>> x
{'bar', 'foo'}
```

Modifying a Set - Pop

- Removes and returns a random element from a set.
  - `x.pop()` - if x is empty, raise an exception

```
>>> x = {'foo', 'bar', 'baz'}

>>> x.pop()
'bar'
>>> x
{'baz', 'foo'}

>>> x.pop()
'baz'
```

Modifying a Set - Pop

```
>>> x
{'foo'}

>>> x.pop()
'foo'

>>> x
set()

>>> x.pop()
KeyError: 'pop from an empty set'
```

Modifying a Set - Clear

- Removes all elements from a set
  - `x.clear()`

```
>>> x = {'foo', 'bar', 'baz'}
>>> x
{'foo', 'bar', 'baz'}

>>> x.clear()
>>> x
set()

>>> x.clear()
>>> x
set()
```

Modifying a Set - Update

- Modify a set by adding any elements that do not already exist. Similar to union
  - `x1.update(x2[, x3...])`
    - Arguments need to be iterables
    - This is called a method (uses dot notation)
  - `x1 |= x2 [| x3 ...]`
    - Operands need to be sets
    - This is called augmented assignment

Modifying a Set - Update

```
>>> x1 = {'foo', 'bar', 'baz'}
>>> x2 = {'foo', 'baz', 'qux'}

>>> x1 |= x2
>>> x1
{'qux', 'foo', 'bar', 'baz'}

>>> x1.update(['corge', 'garply'])
>>> x1
{'qux', 'corge', 'garply', 'foo', 'bar', 'baz'}
```

Modifying a Set - Intersection Update

- Modify a set by retaining only elements found in both sets
  - `x1.intersection_update(x2[, x3...])`
    - Arguments need to be iterables
  - `x1 &= x2 [& x3 ...]`
    - Operands need to be sets

Modifying a Set - Intersection Update

```
>>> x1 = {'foo', 'bar', 'baz'}
>>> x2 = {'foo', 'baz', 'qux'}

>>> x1 &= x2
>>> x1
{'foo', 'baz'}

>>> x1.intersection_update(['baz', 'qux'])
>>> x1
{'baz'}
```

Modifying a Set - Difference Update

- Modify a set by keeping only elements that exist in the first set, not any of the rest
  - `x1.difference_update(x2[, x3...])`
    - Arguments need to be iterables
  - `x1 -= x2 [| x3 ...]`
    - Operands need to be sets
    - Note how there is a `|` between multiple operands. This will mutate `x1` by checking union-ing all sets `x2`, `x3...`, and removing any elements from `x1` that exist in the union.
    - `x1 -= x2 [- x3 ...]`, won't do this...

Modifying a Set - Difference Update

```
>>> a = {1, 2, 3}
>>> b = {2}
>>> c = {3}

>>> a.difference_update(b, c)
>>> a
{1}
```

```
>>> a = {1, 2, 3} # reset a
>>> a -= b | c # need union
>>> a
{1}

>>> a = {1, 2, 3} # reset a
>>> a -= b - c
>>> a
{1, 3}
```

Modifying a Set - Symmetric Difference Update

- Modify a set by keeping only the elements that exist in a single set, but not in multiple
  - `x1.symmetric_difference_update(x2)`
    - Argument needs to be an iterable
    - Only takes in one argument
  - `x1 ^= x2 [^ x3 ...]`
    - Operands need to be sets
    - Works on multiple sets and will mutate `x1` to include all elements found in either any of the sets but not in multiple sets

Modifying a Set - Symmetric Difference Update

```
>>> a = {1, 2, 3}
>>> b = {3, 4, 5}
>>> c = {1, 5, 6}

>>> a.symmetric_difference_update(b)
>>> a
{1, 2, 4, 5}

>>> a = {1, 2, 3} # reset a
>>> a ^= b ^ c
>>> a
{2, 4, 6}
```

Modifying a Set - Augmented Assignment

- Many of the modifying set methods have a corresponding augmented assignment
  - &= or -= or ^= for example
  - These are **not** the same as their expanded out counterparts
  - Ex.
    - `x &= {1}` is **not** the same as `x = x & {1}`

Modifying a Set - Augmented Assignment

```
>>> x = {1}
>>> y = x

>>> x |= {2}
>>> x
{1, 2}

>>> y
{1, 2}
```

Mutates x

vs

```
>>> x = {1}
>>> y = x

>>> x = x | {2}
>>> x
{1, 2}

>>> y
{1}
```

Reassigns x

Frozen Sets - Live Coding

Frozen Sets - Augmented Assignment

- Augmented assignment works differently for normal sets and frozen sets
  - &= or -= or ^= for example
  - For frozen sets these **are** the same as their expanded out counterparts
  - Ex.
    - `x &= {1}` is the same as `x = x & {1}`

Frozen Sets - Augmented Assignment

```
>>> x = frozenset({1})
>>> y = x

>>> x |= {2}
>>> x
frozenset({1, 2})

>>> y
frozenset({1})
```

Reassigns x

vs

```
>>> x = frozenset({1})
>>> y = x

>>> x = x | {2}
>>> x
frozenset({1, 2})

>>> y
frozenset({1})
```

Reassigns x

Conclusion - Why Sets?

- Sometimes you only care about unique values and don’t need your data structure to be ordered
- You can do some cool operations very easily (union, symmetric difference, add, difference update)
- Sets are also very fast
    - You can check for membership almost instantly! Let’s see how it compares with lists and tuples.

Speed Test - Live Coding