# Sets

WE'LL COVER THE FOLLOWING


- Creating A Set
- Modifying A Set
- Removing Items From A Set
- Common Set Operations
- Sets In A Boolean Context


A set is an unordered "bag" of unique values. A single set can contain values of any immutable datatype. Once you have two sets, you can do standard set operations like union, intersection, and set difference.

## Creating A Set #

First things first. Creating a set is easy.

```
a_set = {1}              #①
print (a_set)
#{1}

print (type(a_set))      #②
#<class 'set'>

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

① To create a set with one value, put the value in curly brackets (`{}`).

② Sets are actually implemented as classes, but don't worry about that for now.

③ To create a set with multiple values, separate the values with commas and wrap it all up with curly brackets.

You can also create a set out of a list.

```
a_list = ['a', 'b', 'mpilgrim', True, False, 42]
a_set = set(a_list)                                 #①
print (a_set )                                      #②
#{False, True, 'mpilgrim', 'b', 42, 'a'}

print (a_list )                                     #③
#['a', 'b', 'mpilgrim', True, False, 42]
```

① To create a set from a list, use the `set()` function. (Pedants who know about how sets are implemented will point out that this is not really calling a function, but instantiating a class. I *promise* you will learn the difference later in this book. For now, just know that `set()` acts like a function, and it returns a set.)

② As I mentioned earlier, a single set can contain values of any datatype. And, as I mentioned earlier, sets are *unordered*. This set does not remember the original order of the list that was used to create it. If you were to add items to this set, it would not remember the order in which you added them.

③ The original list is unchanged.

Don't have any values yet? Not a problem. You can create an empty set.

```
a_set = set()           #①
print (a_set )          #②
#set()

print (type(a_set))     #③
#<class 'set'>

print (len(a_set))      #④
#0

print (not_sure = {})   #⑤
#Traceback (most recent call last):
# File "/usercode/__ed_file.py", line 11, in <module>
# print (not_sure = {}) #\u2464
```

```
#TypeError: 'not_sure' is an invalid keyword argument for this function
```

① To create an empty set, call `set()` with no arguments.

② The printed representation of an empty set looks a bit strange. Were you expecting `{}`, perhaps? That would denote an empty dictionary, not an empty set. You'll learn about dictionaries later in this chapter.

③ Despite the strange printed representation, this *is* a set...

④ ...and this set has no members.

⑤ Due to historical quirks carried over from Python 2, you can not create an empty set with two curly brackets. This actually creates an empty dictionary, not an empty set.

## Modifying A Set #

There are two different ways to add values to an existing set: the `add()` method, and the `update()` method.

```
a_set = {1, 2}
a_set.add(4)          #①
print (a_set)
#{1, 2, 4}

print (len(a_set))    #②
#3

a_set.add(1)          #③
print (a_set)
#{1, 2, 4}

print (len(a_set))    #④
#3
```

① The `add()` method takes a single argument, which can be any datatype, and adds the given value to the set.

② This set now has 3 members.

③ Sets are bags of *unique values*. If you try to add a value that already exists in the set, it will do nothing. It won't raise an error; it's just a no-op.

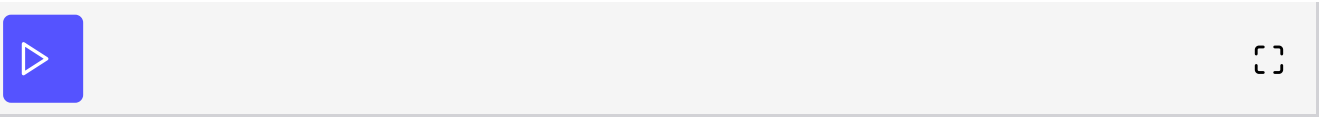
④ This set *still* has 3 members.

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

a_set.update({2, 4, 6})                         #①
print (a_set )                                  #②
#{1, 2, 3, 4, 6}

a_set.update({3, 6, 9}, {1, 2, 3, 5, 8, 13})  #③
print (a_set)
#{1, 2, 3, 4, 5, 6, 8, 9, 13}

a_set.update([10, 20, 30])                      #④
print (a_set)
#{1, 2, 3, 4, 5, 6, 8, 9, 10, 13, 20, 30}
```



① The `update()` method takes one argument, a set, and adds all its members to the original set. It's as if you called the `add()` method with each member of the set.

② Duplicate values are ignored, since sets can not contain duplicates.

③ You can actually call the `update()` method with any number of arguments. When called with two sets, the `update()` method adds all the members of each set to the original set (dropping duplicates).

④ The `update()` method can take objects of a number of different datatypes, including lists. When called with a list, the `update()` method adds all the items of the list to the original set.

## Removing Items From A Set #

There are three ways to remove individual values from a set. The first two, `discard()` and `remove()`, have one subtle difference.

```
a_set = {1, 3, 6, 10, 15, 21, 28, 36, 45}
print (a_set)
#{1, 3, 36, 6, 10, 45, 15, 21, 28}

a_set.discard(10)                           #①
```

```
print (a_set)
#{1, 3, 36, 6, 45, 15, 21, 28}


a_set.discard(10)                          #②
print (a_set)
#{1, 3, 36, 6, 45, 15, 21, 28}

a_set.remove(21)                           #③
print (a_set)
#{1, 3, 36, 6, 45, 15, 28}

print (a_set.remove(21))                   #④
#Traceback (most recent call last):
# File "/usercode/__ed_file.py", line 17, in <module>
# print (a_set.remove(21)) #\u2463
#KeyError: 21
```

① The `discard()` method takes a single value as an argument and removes that value from the set.

② If you call the `discard()` method with a value that doesn't exist in the set, it does nothing. No error; it's just a no-op.

③ The `remove()` method also takes a single value as an argument, and it also removes that value from the set.

④ Here's the difference: if the value doesn't exist in the set, the `remove()` method raises a `KeyError` exception.

Like lists, sets have a `pop()` method.

```
a_set = {1, 3, 6, 10, 15, 21, 28, 36, 45}
print (a_set.pop())                                #①
#1

print (a_set.pop())
#3

print (a_set.pop())
#36

print (a_set)
#{6, 10, 45, 15, 21, 28}

a_set.clear()                                      #②
print (a_set)
#set()

print (a_set.pop() )                               #③
#Traceback (most recent call last):
```

```
# File "/usercode/__ed_file.py", line 18, in <module>
# print (a_set.pop() ) #\u2462
#KeyError: 'pop from an empty set'
```

① The `pop()` method removes a single value from a set and returns the value. However, since sets are unordered, there is no "last" value in a set, so there is no way to control which value gets removed. It is completely arbitrary.

② The `clear()` method removes *all* values from a set, leaving you with an empty set. This is equivalent to `a_set = set()`, which would create a new empty set and overwrite the previous value of the `a_set` variable.

③ Attempting to pop a value from an empty set will raise a `KeyError` exception.

## Common Set Operations #

Python's set type supports several common set operations.

```
a_set = {2, 4, 5, 9, 12, 21, 30, 51, 76, 127, 195}
print (30 in a_set)                                                    #①
#True

print (31 in a_set)
#False

b_set = {1, 2, 3, 5, 6, 8, 9, 12, 15, 17, 18, 21}
print (a_set.union(b_set))                                             #②
#{1, 2, 195, 4, 5, 6, 8, 12, 76, 15, 17, 18, 3, 21, 30, 51, 9, 127}

print (a_set.intersection(b_set))                                      #③
#{9, 2, 12, 5, 21}

print (a_set.difference(b_set) )                                       #④
#{195, 4, 76, 51, 30, 127}

print (a_set.symmetric_difference(b_set))                              #⑤
#{1, 3, 4, 6, 8, 76, 15, 17, 18, 195, 127, 30, 51}
```

① To test whether a value is a member of a set, use the `in` operator. This works the same as lists.

② The `union()` method returns a new set containing all the elements that are

in *either* set.

③ The `intersection()` method returns a new set containing all the elements that are in *both* sets.

④ The `difference()` method returns a new set containing all the elements that are in `a_set` but not `b_set`.

⑤ The `symmetric_difference()` method returns a new set containing all the elements that are in *exactly one* of the sets.

Three of these methods are symmetric.

```
# continued from the previous example
a_set = {2, 4, 5, 9, 12, 21, 30, 51, 76, 127, 195}
b_set = {1, 2, 3, 5, 6, 8, 9, 12, 15, 17, 18, 21}

print (b_set.symmetric_difference(a_set))                                    #①
#{1, 195, 4, 3, 6, 8, 76, 15, 17, 18, 51, 30, 127}

print (b_set.symmetric_difference(a_set) == a_set.symmetric_difference(b_set))  #②
#True

print (b_set.union(a_set) == a_set.union(b_set))                             #③
#True

print (b_set.intersection(a_set) == a_set.intersection(b_set) )              #④
#True

print (b_set.difference(a_set) == a_set.difference(b_set) )                  #⑤
#False
```

① The symmetric difference of `a_set` from `b_set` *looks* different than the symmetric difference of `b_set` from `a_set`, but remember, sets are unordered. Any two sets that contain all the same values (with none left over) are considered equal.

② And that's exactly what happens here. Don't be fooled by the Python Shell's printed representation of these sets. They contain the same values, so they are equal.

③ The union of two sets is also symmetric.

④ The intersection of two sets is also symmetric.

⑤ The difference of two sets is not symmetric. That makes sense; it's analogous to subtracting one number from another. The order of the operands matters.

Finally, there are a few questions you can ask of sets.

```
a_set = {1, 2, 3}
b_set = {1, 2, 3, 4}
print (a_set.issubset(b_set))    #①
#True

print (b_set.issuperset(a_set))  #②
#True

a_set.add(5)                     #③
print (a_set.issubset(b_set))
#False

print (b_set.issuperset(a_set))
#False
```

① `a_set` is a subset of `b_set` — all the members of `a_set` are also members of `b_set`.

② Asking the same question in reverse, `b_set` is a superset of `a_set`, because all the members of `a_set` are also members of `b_set`.

③ As soon as you add a value to `a_set` that is not in `b_set`, both tests return `False`.

## Sets In A Boolean Context #

You can use sets in a boolean context, such as an if statement.

```
def is_it_true(anything):
   if anything:
     print("yes, it's true")
   else:
     print("no, it's false")

print (is_it_true(set()))          #①
#no, it's false
#None
```

```
print (is_it_true({'a'}))          #②
#yes, it's true

#None

print (is_it_true({False}))        #③
#yes, it's true
#None
```

① In a boolean context, an empty set is false.

② Any set with at least one item is true.

③ Any set with at least one item is true. The value of the items is irrelevant.