

```
Initialize a Set by passing a list to set()  
set_one = set(['one', 'two', 'three'])
```

Python Sets and Set Theory

Learn about Python sets: what they are, how to create them, when to use them, built-in functions, and their relationship to set theory operations.

[Michael Galarnyk](#)

[Jun 6, 2018](#)

Sets vs Lists and Tuples

A Set is a mutable collection of unique, immutable, unordered values.

Lists and tuples are standard Python data types that store values in a sequence. Sets are another standard Python data type that also store values. The major difference is that sets, unlike lists or tuples, cannot have multiple occurrences of the same element and store unordered values.

Advantages of Python Sets

Set are:

1. Only unique values
2. No unordered values

Because sets cannot have multiple occurrences of the same element, it makes sets highly useful to efficiently remove duplicate values from a list or tuple and to perform common math operations like unions and intersections.

This tutorial will introduce you a few topics about Python sets and set theory:

- How to initialize empty sets and sets with values.
- How to add and remove values from sets
- How to use efficiently use sets for tasks such as membership tests and removing duplicate values from a list.
- How to perform common set operations like unions, intersections, difference, and symmetric difference.
- The difference between a set and a frozenset

With that, let's get started.

Initialize a Set

Sets are a mutable collection of distinct (unique) immutable values that are unordered.

You can initialize an empty set by using `set()`.

```
emptySet = set()
```

To initialize a set with values, you can pass in a list to `set()`.

```
dataScientist = set(['Python', 'R', 'SQL', 'Git', 'Tableau', 'SAS'])
```

```
dataEngineer = set(['Python', 'Java', 'Scala', 'Git', 'SQL', 'Hadoop'])
```

Set Creation

Empty:

```
mtSet = set()
```

Sets with Values:

```
var = set(['list'])
```

or

```
var = set{'val1', 'etc'}
```

Note:

```
var = dict{}
```

with empty curly brackets

```
dataScientist = set(['Python', 'R', 'SQL', 'Git', 'Tableau', 'SAS'])  
dataEngineer = set(['Python', 'Java', 'Scala', 'Git', 'SQL', 'Hadoop'])
```

```
dataScientist
```

```
{'Git', 'Python', 'R', 'SAS', 'SQL', 'Tableau'}
```

```
dataEngineer
```

```
{'Git', 'Hadoop', 'Java', 'Python', 'SQL', 'Scala'}
```

Notice Auto-Orders

If you look at the output of `dataScientist` and `dataEngineer` variables above, notice that the values in the set are not in the order added in. This is because sets are unordered.

Sets containing values can also be initialized by using curly braces.

```
dataScientist = {'Python', 'R', 'SQL', 'Git', 'Tableau', 'SAS'}
dataEngineer = {'Python', 'Java', 'Scala', 'Git', 'SQL', 'Hadoop'}
```

```
dataScientist = {'Python', 'R', 'SQL', 'Git', 'Tableau', 'SAS'}
dataEngineer = {'Python', 'Java', 'Scala', 'Git', 'SQL', 'Hadoop'}
```

```
dataScientist
```

```
{'Git', 'Python', 'R', 'SAS', 'SQL', 'Tableau'}
```

```
dataEngineer
```

```
{'Git', 'Hadoop', 'Java', 'Python', 'SQL', 'Scala'}
```

Keep in mind that curly braces can only be used to initialize a set containing values. The image below shows that using curly braces without values is one of the ways to initialize a dictionary and not a set.

```
# Initialize empty set      # Initialize empty dictionary
emptySet = set()           emptyDict = dict()
                           emptyDict = {}
```

Choice 1

Choice 2

Add and Remove Values from Sets

To add or remove values from a set, you first have to initialize a set.

```
# Initialize set with values
```

```
graphicDesigner = {'InDesign', 'Photoshop', 'Acrobat', 'Premiere', 'Bridge'}
```

Add Values to a Set

You can use the method `.add` to add a value to a set.

```
graphicDesigner.add('Illustrator')
```

```
graphicDesigner.add('Illustrator')
```

```
graphicDesigner
```

```
{'Acrobat', 'Bridge', 'Illustrator', 'InDesign', 'Photoshop', 'Premiere'}
```

It is important to note that you can only add a value that is immutable (like a string or a tuple) to a set. For example, you would get a `TypeError` if you try to add a list to a set.

```
graphicDesigner.add(['Powerpoint', 'Blender'])
```

```
graphicDesigner.add(['Powerpoint', 'Blender'])
```

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-4-66abf4996769> in <module>()
----> 1 graphicDesigner.add(['Powerpoint', 'Blender'])

TypeError: unhashable type: 'list'
```

Difference between unordered & ordered?

1. unordered -
2. ordered -

Set-Remove-Value-Methods

.remove(' ')
.discard(' ')
.pop()
.clear()

Remove Values from a Set

There are a couple ways to remove a value from a set.

Option 1: You can use the `remove` method to remove a value from a set.

```
graphicDesigner.remove('Illustrator')
```

.remove(' ')

```
graphicDesigner
{'Acrobat', 'Bridge', 'Illustrator', 'InDesign', 'Photoshop', 'Premiere'}

graphicDesigner.remove('Illustrator')

graphicDesigner
{'Acrobat', 'Bridge', 'InDesign', 'Photoshop', 'Premiere'}
```

The drawback of this method is that if you try to remove a value that is not in your set, you will get a `KeyError`.

```
graphicDesigner.remove('Muse')

-----
KeyError                                Traceback (most recent call last)
<ipython-input-8-b78c3505d8ef> in <module>()
----> 1 graphicDesigner.remove('Muse')

KeyError: 'Muse'
```

Option 2: You can use the `discard` method to remove a value from a set.

```
graphicDesigner.discard('Premiere')
```

.discard(' ')

```
graphicDesigner
{'Acrobat', 'Bridge', 'InDesign', 'Photoshop', 'Premiere'}

graphicDesigner.discard('Premiere')

graphicDesigner
{'Acrobat', 'Bridge', 'InDesign', 'Photoshop'}
```

The benefit of this approach over the `remove` method is if you try to remove a value that is not part of the set, you will not get a `KeyError`. If you are familiar with dictionaries, you might find that this works similarly to the [dictionary method get](#).

Option 3: You can also use the `pop` method to **remove and return** an arbitrary value from a set.

```
graphicDesigner.pop()
```

.clear()

```
graphicDesigner
{'Acrobat', 'Bridge', 'InDesign', 'Photoshop'}

graphicDesigner.pop()
'InDesign'

graphicDesigner
{'Acrobat', 'Bridge', 'Photoshop'}
```

It is important to note that the method raises a `KeyError` if the set is empty.

Remove All Values from a Set

You can use the `clear` method to remove all values from a set.

```
graphicDesigner.clear()
```

.pop()

```
graphicDesigner
{'Acrobat', 'Bridge', 'Photoshop'}

graphicDesigner.clear()

graphicDesigner
set()
```

Iterate through a Set

Like many standard python data types, it is possible to iterate through a set.

```
# Initialize a set
dataScientist = {'Python', 'R', 'SQL', 'Git', 'Tableau', 'SAS'}
for skill in dataScientist:
    print(skill)
```

```
for skill in dataScientist:
    print(skill)
```

```
SQL
Tableau
Git
Python
R
SAS
```

If you look at the output of printing each of the values in dataScientist, notice that the values printed in the set are not in the order they were added in. This is because sets are unordered.

Transform Set into Ordered Values

This tutorial has emphasized that sets are unordered. If you find that you need to get the values from your set in an ordered form, you can use the sorted function which outputs a list that is ordered.

```
type(sorted(dataScientist))
```

```
type(sorted(dataScientist, reverse = True))
```

```
list
```

'reverse = True'
gives 'z' to 'a'

The code below outputs the values in the set dataScientist in descending alphabetical order (Z-A in this case).

```
sorted(dataScientist, reverse = True)
```

```
dataScientist
```

```
{'Git', 'Python', 'R', 'SAS', 'SQL', 'Tableau'}
```

```
sorted(dataScientist, reverse = True)
```

```
['Tableau', 'SQL', 'SAS', 'R', 'Python', 'Git']
```

'reverse = False'
gives 'a' to 'z'

Remove Duplicates from a List

Part of the content in this section was previously explored in the tutorial [18 Most Common Python List Questions](#), but it is important to emphasize that **sets are the fastest way to remove duplicates from a list**. To show this, let's study **the performance difference between two approaches**.

Approach 1: Use a set to remove duplicates from a list.

```
print(list(set([1, 2, 3, 1, 7])))
```

Approach 2: Use a list comprehension to remove duplicates from a list (If you would like a refresher on list comprehensions, see this [tutorial](#)).

```
def remove_duplicates(original):  
    unique = []  
    [unique.append(n) for n in original if n not in unique]  
    return(unique)print(remove_duplicates([1, 2, 3, 1, 7]))
```

The **performance difference can be measured using the the timeit library** which allows you to time your Python code. The code below runs the code for each approach 10000 times and outputs the overall time it took in seconds.

```
import timeit  
# Approach 1: Execution time  
print(timeit.timeit('list(set([1, 2, 3, 1, 7]))', number=10000))  
# Approach 2: Execution time  
print(timeit.timeit('remove_duplicates([1, 2, 3, 1, 7])', globals=globals(),  
number=10000))
```

```
# Approach 1: Execution time  
print(timeit.timeit('list(set([1, 2, 3, 1, 7]))', number=10000))
```

```
0.0067351709949434735
```

```
# Approach 2: Execution time  
print(timeit.timeit('remove_duplicates([1, 2, 3, 1, 7])', globals=globals(), number=10000))
```

```
0.012259711998922285
```

Comparing these two approaches shows that **using sets to remove duplicates is more efficient**. While it may seem like a small difference in time, **it can save you a lot of time if you have very large lists**.

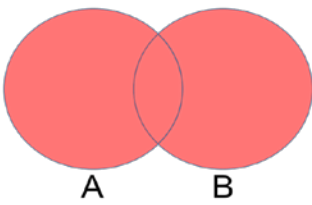
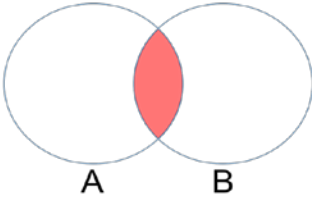
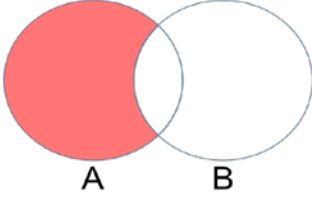
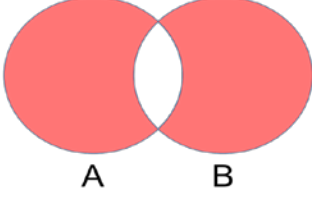
Set Operation Methods (4 methods)

A common use of sets in Python is computing standard math operations such as:

- union,
- intersection,
- difference, and
- symmetric difference.

The image below shows a couple standard math operations on two sets A and B.

The red part of each Venn diagram is the resulting set of a given set operation.

Set Operation	Venn Diagram	Interpretation
Union		$A \cup B$, is the set of all values that are a member of A, or B, or both.
Intersection		$A \cap B$, is the set of all values that are members of both A and B.
Difference		$A \setminus B$, is the set of all values of A that are not members of B
Symmetric Difference		$A \triangle B$, is the set of all values which are in one of the sets, but not both.

Python sets have methods that allow you to perform these mathematical operations as well as operators that give you equivalent results.

Before exploring these methods, let's start by initializing two sets *dataScientist* and *dataEngineer*.

```
dataScientist = set(['Python', 'R', 'SQL', 'Git', 'Tableau', 'SAS'])
dataEngineer = set(['Python', 'Java', 'Scala', 'Git', 'SQL', 'Hadoop'])
```

```
dataScientist
unique:
-R
-Tableau
-SAS
dataEngineer
unique:
-Java
-Scala
-Hadoop
shared in
common:
-Python
-SQL
-Git
```

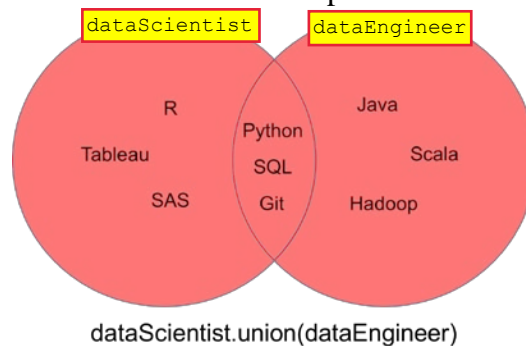
1) Union

A union, denoted **dataScientist** \cup **dataEngineer**, is the set of all values that are values of dataScientist, or dataEngineer, or both. You can use the union method to find out all the unique values in two sets.

```
# set built-in function union
dataScientist.union(dataEngineer)
# Equivalent Result
dataScientist | dataEngineer
```

```
dataScientist.union(dataEngineer)
{'Git', 'Hadoop', 'Java', 'Python', 'R', 'SAS', 'SQL', 'Scala', 'Tableau'}
```

The set returned from the union can be visualized as the red part of the Venn diagram below.



2) Intersection

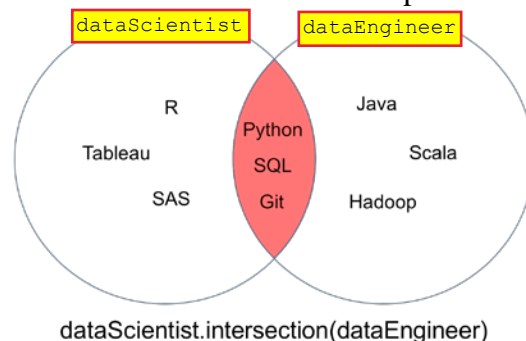
An intersection of two sets dataScientist and dataEngineer, denoted **dataScientist** \cap **dataEngineer**, is the set of all values that are values of both dataScientist and dataEngineer.

```
# Intersection operation
dataScientist.intersection(dataEngineer)
# Equivalent Result
dataScientist & dataEngineer
```

same result if reverse sets?

```
dataScientist.intersection(dataEngineer)
{'Git', 'Python', 'SQL'}
```

The set returned from the intersection can be visualized as the red part of the Venn diagram below.



You may find that you come across a case where you want to make sure that two sets have no value in common. In other words, you want two sets that have an intersection that is empty.

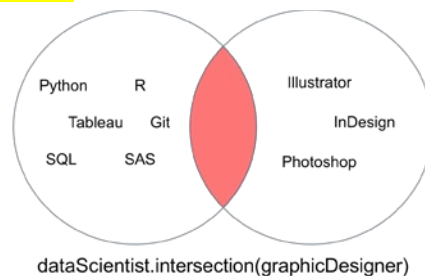
These two sets are called disjoint sets.

You can test for disjoint sets by using the `isdisjoint` method.

```
# Initialize a set
graphicDesigner = {'Illustrator', 'InDesign', 'Photoshop'}
# These sets have elements in common so it would return False
dataScientist.isdisjoint(dataEngineer)
# These sets have no elements in common so it would return True
dataScientist.isdisjoint(graphicDesigner)

graphicDesigner = {'Illustrator', 'InDesign', 'Photoshop'}
dataScientist.isdisjoint(dataEngineer)
False
dataScientist.isdisjoint(graphicDesigner)
True
```

You can notice in the intersection shown in the Venn diagram below that the disjoint sets `dataScientist` and `graphicDesigner` have no values in common.



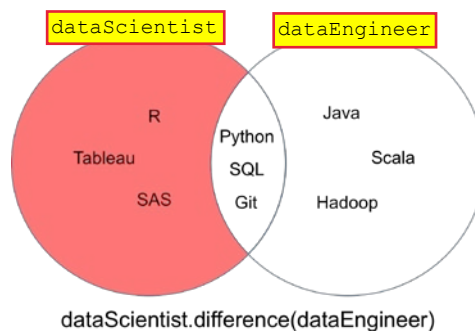
3) Difference

A difference of two sets `dataScientist` and `dataEngineer`, denoted `dataScientist \ dataEngineer`, is the set of all values of `dataScientist` that are not values of `dataEngineer`.

```
# Difference Operation
dataScientist.difference(dataEngineer)
# Equivalent Result
dataScientist - dataEngineer
```

```
dataScientist.difference(dataEngineer)
{'R', 'SAS', 'Tableau'}
```

The set returned from the difference can be visualized as the red part of the Venn diagram below.



4) Symmetric_Difference

gives unique to each set

A symmetric difference of two sets `dataScientist` and `dataEngineer`, denoted `dataScientist Δ dataEngineer`, is the set of all values that are values of exactly one of two sets, but not both.

Symmetric Difference Operation

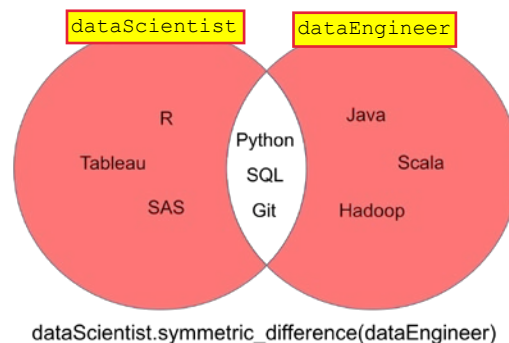
```
dataScientist.symmetric_difference(dataEngineer)
```

Equivalent Result

```
dataScientist ^ dataEngineer
```

```
dataScientist.symmetric_difference(dataEngineer)
{'Hadoop', 'Java', 'R', 'SAS', 'Scala', 'Tableau'}
```

The set returned from the symmetric difference can be visualized as the red part of the Venn diagram below.



Set Comprehension

need explained

You may have previously have learned about [list comprehensions](#), [dictionary comprehensions](#), and generator comprehensions. There is also set comprehensions. Set comprehensions are very similar. Set comprehensions in Python can be constructed as follows:

```
{skill for skill in ['SQL', 'SQL', 'PYTHON', 'PYTHON']}
```

```
{skill for skill in ['SQL', 'SQL', 'PYTHON', 'PYTHON']}
{'PYTHON', 'SQL'}
```

The output above is a set of 2 values because sets cannot have multiple occurrences of the same element.

The idea behind using set comprehensions is to let you write and reason in code the same way you would do mathematics by hand.

```
{skill for skill in ['GIT', 'PYTHON', 'SQL'] if skill not in {'GIT', 'PYTHON', 'JAVA'}}
```

```
{skill for skill in ['GIT', 'PYTHON', 'SQL'] if skill not in {'GIT', 'PYTHON', 'JAVA'}}
{'SQL'}
```

The code above is similar to a set difference you learned about earlier. It just looks a bit different.

Membership Tests

Membership tests check whether a specific element is contained in a sequence, such as strings, lists, tuples, or sets. One of the main advantages of using sets in Python is that they are highly optimized for membership tests. For example, sets do membership tests a lot more efficiently than lists. In case you are from a computer science background, this is because the [average case time complexity of membership tests in sets are \$O\(1\)\$ vs \$O\(n\)\$ for lists](#).

The code below shows a membership test using a list.

```
# Initialize a list
possibleList = ['Python', 'R', 'SQL', 'Git', 'Tableau', 'SAS', 'Java', 'Spark', 'Scala']
# Membership test
'Python' in possibleList
```

```
# Initialize a list
possibleList = ['Python', 'R', 'SQL', 'Git', 'Tableau', 'SAS', 'Java', 'Spark', 'Scala']

# Membership test
'Python' in possibleList

True
```

Something similar can be done for sets. Sets just happen to be more efficient.

```
# Initialize a set
possibleSet = {'Python', 'R', 'SQL', 'Git', 'Tableau', 'SAS', 'Java', 'Spark', 'Scala'}
# Membership test
'Python' in possibleSet
```

```
# Initialize a set
possibleSet = {'Python', 'R', 'SQL', 'Git', 'Tableau', 'SAS', 'Java', 'Spark', 'Scala'}

# Membership test
'Python' in possibleSet

True
```

Since possibleSet is a set and the value 'Python' is a value of possibleSet, this can be denoted as 'Python' \in possibleSet.

If you had a value that wasn't part of the set, like 'Fortran', it would be denoted as 'Fortran' \notin possibleSet.

Subset

A practical application of understanding membership is subsets.

Let's first initialize two sets.

```
possibleSkills = {'Python', 'R', 'SQL', 'Git', 'Tableau', 'SAS'}
mySkills = {'Python', 'R'}
```

If every value of the set mySkills is also a value of the set possibleSkills, then mySkills is said to be a subset of possibleSkills, mathematically written $\text{mySkills} \subseteq \text{possibleSkills}$.

You can check to see if one set is a subset of another using the method issubset.

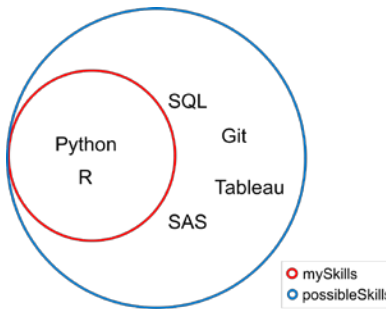
```
mySkills.issubset(possibleSkills)
```

```
mySkills.issubset(possibleSkills)
```

```
True
```

Since the method returns True in this case, it is a subset.

In the Venn diagram below, notice that every value of the set `mySkills` is also a value of the set `possibleSkills`.



Frozensets

immutable (not changable) set

You have have encountered nested lists and tuples.

Nested Lists and Tuples

```
nestedLists = [['the', 12], ['to', 11], ['of', 9], ['and', 7], ['that', 6]]
```

```
nestedTuples = (('the', 12), ('to', 11), ('of', 9), ('and', 7), ('that', 6))
```

```
# Nested Lists and Tuples
```

```
nestedLists = [['the', 12], ['to', 11], ['of', 9], ['and', 7], ['that', 6]]
nestedTuples = (('the', 12), ('to', 11), ('of', 9), ('and', 7), ('that', 6))
```

```
nestedLists
```

```
[['the', 12], ['to', 11], ['of', 9], ['and', 7], ['that', 6]]
```

```
nestedTuples
```

```
(( 'the', 12), ('to', 11), ('of', 9), ('and', 7), ('that', 6))
```

The problem with nested sets is that you cannot normally have nested sets as sets cannot contain mutable values including sets.

changing

```
nestedSets = set([set()])
```

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-22-13ea2492fef> in <module>()
----> 1 nestedSets = set([set()])
```

```
TypeError: unhashable type: 'set'
```

This is one situation where you may wish to use a frozenset. A frozenset is very similar to a set except that a frozenset is immutable.

You make a frozenset by using `frozenset()`.

```
# Initialize a frozenset
immutableSet = frozenset()
```

```
# Initialize a frozenset
immutableSet = frozenset()
```

```
immutableSet

frozenset()
```

You can make a nested set if you utilize a frozenset similar to the code below.

```
nestedSets = set([frozenset()])
```

```
nestedSets = set([frozenset()])
```

```
nestedSets

{frozenset()}
```

It is important to keep in mind that a major disadvantage of a frozenset is that since they are immutable, it means that you cannot add or remove values.

Conclusion

The Python sets are highly useful to efficiently remove duplicate values from a collection like a list and to perform common math operations like unions and intersections.

Some of the challenges people often encounter are when to use the various data types. For example, if you feel like you aren't sure when it is advantageous to use a dictionary versus a set, I encourage you to check out DataCamp's [daily practice mode](#).

If you any questions or thoughts on the tutorial, feel free to reach out in the comments below or through [Twitter](#).

Originally published at www.datacamp.com.

[Michael Galarnyk](#)

Data Scientist at Scripps Research Institute

<https://towardsdatascience.com/python-sets-and-set-theory-2ace093d1607>