Other Set Operations

Below is a list of all set operations available in Python.

Method	Description
union()	**Return a new set containing the union of two or more sets**
update()	**Modify this set with the union of this set and other sets**
intersection()	**Returns a new set which is the intersection of two or more sets**
intersection_update()	**Removes the items from this set that are not present in other sets**
difference()	**Returns a new set containing the difference between two or more sets**
difference_update()	**Removes the items from this set that are also included in another set**
symmetric_difference()	**Returns a new set with the symmetric differences of two or more sets**
symmetric_difference_update()	**Modify this set with the symmetric difference of this set and other set**
isdisjoint()	**Determines whether or not two sets have any elements in common**
issubset()	**Determines whether one set is a subset of the other**
issuperset()	**Determines whether one set is a superset of the other**

Python Set union() Method

Returns a set with items from all the specified sets

Usage

The `union()` **method returns a new set containing all items from all the specified sets, with no duplicates.**

You can specify as many sets **as you want, just separate each set with a comma.**

If you want to modify the original set instead of returning a new one, use update() method.

Syntax

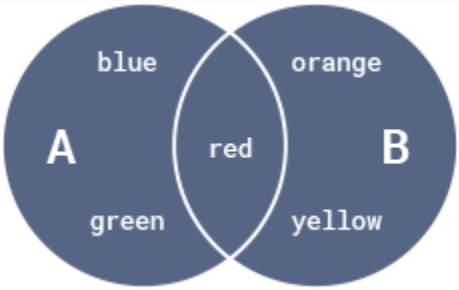
```
set.union(set1,set2...)
```

Parameter	Condition	Description
set1, set2…	**Optional**	**A comma-separated list of one or more sets to merge with.**

Basic Example

```
# Perform union of two sets
A = {'red', 'green', 'blue'}
B = {'yellow', 'red', 'orange'}

print(A.union(B))
# Prints {'blue', 'green', 'yellow', 'orange', 'red'}
```



Equivalent Operator |

Set union can be performed with the | operator as well.

```
A = {'red', 'green', 'blue'}
B = {'yellow', 'red', 'orange'}

# by method
print(A.union(B))
```

```
# Prints {'blue', 'green', 'yellow', 'orange', 'red'}


# by operator
print(A | B)
# Prints {'blue', 'green', 'yellow', 'orange', 'red'}
```

Union of Multiple Sets

Multiple sets can be specified with either the operator or the method.

```
A = {'red', 'green', 'blue'}
B = {'yellow', 'orange', 'red'}
C = {'blue', 'red', 'black'}


# by method
print(A.union(B,C))
# Prints {'blue', 'green', 'yellow', 'orange', 'black', 'red'}


# by operator
print(A | B | C)
# Prints {'blue', 'green', 'yellow', 'orange', 'black', 'red'}
```

Python Set update() Method

Updates the set by adding items from all the specified sets

Usage

The `update()` method updates the original set by adding items from all the specified sets, with no duplicates.

You can specify as many sets as you want, just separate each set with a comma.

If you don't want to update the original set, use union() method.

Syntax

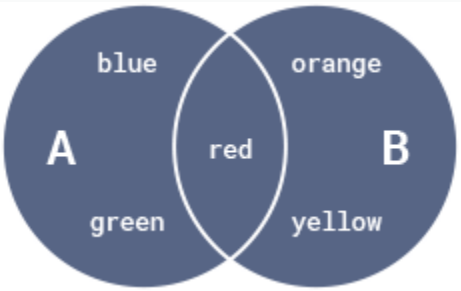
```
set.update(set1,set2…)
```

Parameter	Condition	Description
set1, set2…	**Optional**	**A comma-separated list of one or more sets to merge with.**

Basic Example

```
# Update the set by adding items from other set
A = {'red', 'green', 'blue'}
B = {'yellow', 'red', 'orange'}

A.update(B)
print(A)
# Prints {'blue', 'green', 'yellow', 'orange', 'red'}
```



Equivalent Operator |=

You can achieve the same result by using the |= augmented assignment operator.

```
A = {'red', 'green', 'blue'}
B = {'yellow', 'red', 'orange'}

A |= B
print(A)
# Prints {'blue', 'green', 'yellow', 'orange', 'red'}
```

Update() Method with Multiple Sets

Multiple sets can be specified with either the operator or the method.

```
A = {'red', 'green', 'blue'}
B = {'yellow', 'orange', 'red'}
C = {'blue', 'red', 'black'}

# by method
A.update(B,C)
print(A)
# Prints {'blue', 'green', 'yellow', 'orange', 'black', 'red'}

# by operator
A |= B | C
print(A)
# Prints {'blue', 'green', 'yellow', 'orange', 'black', 'red'}
```

Python Set intersection() Method

Returns a set with items common to all the specified sets

Usage

The intersection() **method returns a new set of items that are common to all the specified sets.**

You can specify as many sets **as you want, just separate each set with a comma.**

If you want to modify the original set instead of returning a new one, use intersection_update() method.

Syntax

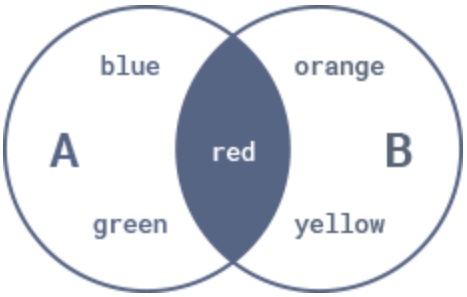
set.intersection(set1,set2…)

Parameter	Condition	Description
set1, set2…	**Optional**	**A comma-separated list of one or more sets to search for common items in**

Basic Example

```
# Perform intersection of two sets
A = {'red', 'green', 'blue'}
B = {'yellow', 'red', 'orange'}

print(A.intersection(B))
# Prints {'red'}
```



Equivalent Operator &

Set intersection can be performed with the `&` operator as well.

```
A = {'red', 'green', 'blue'}
B = {'yellow', 'red', 'orange'}

# by method
print(A.intersection(B))
# Prints {'red'}

# by operator
print(A & B)
# Prints {'red'}
```

Intersection of Multiple Sets

Multiple sets can be specified with either the operator or the method.

```
A = {'red', 'green', 'blue'}
B = {'yellow', 'orange', 'red'}
C = {'blue', 'red', 'black'}

# by method
print(A.intersection(B,C))
# Prints {'red'}
```

```
# by operator
print(A & B & C)
# Prints {'red'}
```

Python Set intersection_update() Method

Updates the set by removing the items that are not common

Usage

The intersection_update() **method updates the set by removing the items that are not common to all the specified sets.**

You can specify as many sets **as you want, just separate each set with a comma.**

If you don't want to update the original set, use intersection() method.

Syntax

set.intersection_update(set1,set2…)

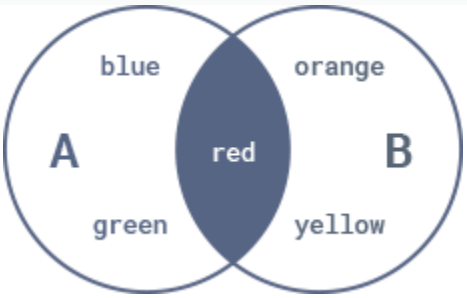
Parameter	Condition	Description

set1, set2…	**Optional**	**A comma-separated list of one or more sets to search for common items in**

Basic Example

```
# Remove items that are not common to both A & B
A = {'red', 'green', 'blue'}
B = {'yellow', 'red', 'orange'}

A.intersection_update(B)
print(A)
# Prints {'red'}
```



Equivalent Operator &=

You can achieve the same result by using the `&=` augmented assignment operator.

```
A = {'red', 'green', 'blue'}
B = {'yellow', 'red', 'orange'}

A &= B
print(A)
# Prints {'red'}
```

intersection_update() Method with Multiple Sets

Multiple sets can be specified with either the operator or the method.

```
A = {'red', 'green', 'blue'}
B = {'yellow', 'orange', 'red'}
C = {'blue', 'red', 'black'}


# by method
A.intersection_update (B,C)
print(A)
# Prints {'red'}


# by operator
A &= B & C
print(A)
# Prints {'red'}
```

Python Set difference() Method

Returns a new set with items in the set that are not in other sets

Usage

The difference() **method returns a new set of items that are in the original set but not in any of the specified sets.**

You can specify as many sets **as you want, just separate each set with a comma.**

If you want to modify the original set instead of returning a new one, use difference_update() method.

Syntax

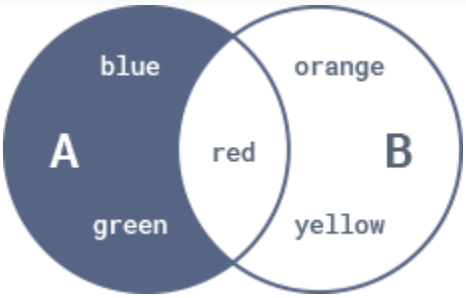
set.difference(set1,set2…)

Parameter	Condition	Description
set1, set2…	**Optional**	**A comma-separated list of one or more sets to find differences in**

Basic Example

```
# Compute the difference between two sets
A = {'red', 'green', 'blue'}
B = {'yellow', 'red', 'orange'}


print(A.difference(B))
# Prints {'blue', 'green'}
```



Equivalent Operator –

Set difference can be performed with the `-` operator as well.

```
A = {'red', 'green', 'blue'}
B = {'yellow', 'red', 'orange'}

# by method
print(A.difference(B))
# Prints {'blue', 'green'}

# by operator
print(A - B)
# Prints {'blue', 'green'}
```

Difference between Multiple Sets

Multiple sets can be specified with either the operator or the method.

```
A = {'red', 'green', 'blue'}
B = {'yellow', 'orange', 'red'}
C = {'blue', 'red', 'black'}

# by method
print(A.difference(B,C))
# Prints {'green'}

# by operator
print(A - B - C)
# Prints {'green'}
```

Python Set difference_update() Method

Updates the set by removing items found in other sets

Usage

The difference_update() **method updates the set by removing items found in specified sets.**

You can specify as many sets **as you want, just separate each set with a comma.**

If you don't want to update the original set, use difference() method.

Syntax

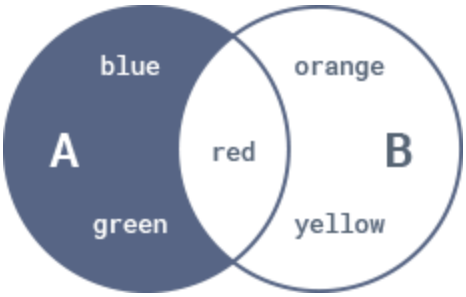
set.difference_update(set1,set2…)

Parameter	Condition	Description
set1, set2…	**Optional**	**A comma-separated list of one or more sets to find differences in**

Basic Example

```
# Remove items from A found in B
A = {'red', 'green', 'blue'}
B = {'yellow', 'red', 'orange'}

A.difference_update(B)
print(A)
# Prints {'blue', 'green'}
```



Equivalent Operator -=

You can achieve the same result by using the `-=` augmented assignment operator.

```
A = {'red', 'green', 'blue'}
B = {'yellow', 'red', 'orange'}


A -= B
print(A)
# Prints {'blue', 'green'}
```

difference_update() Method with Multiple Sets

Multiple sets can be specified with either the operator or the method.

```
A = {'red', 'green', 'blue'}
B = {'yellow', 'orange', 'red'}
C = {'blue', 'red', 'black'}


# by method
A.difference_update(B,C)
print(A)
# Prints {'green'}
```

```
# by operator
A -= B | C
print(A)
# Prints {'green'}
```

Python Set symmetric_difference() Method

Returns a new set with items from all the sets, except common items

Usage

The symmetric_difference() **method returns a new set containing all items from both the sets, except common items.**

If you want to modify the original set instead of returning a new one, use symmetric_difference_update() method.

The symmetric difference is actually the union of the two sets, minus their intersection.

Syntax

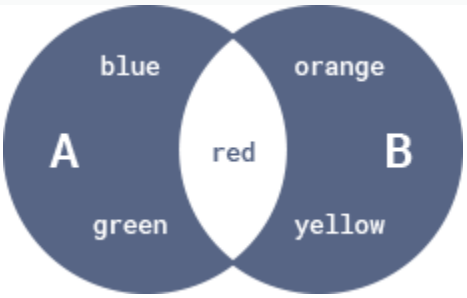
```
set.symmetric_difference(set)
```

Parameter	Condition	Description
set	**Required**	**A set to find difference in**

Basic Example

```
# Compute the symmetric difference between two sets
A = {'red', 'green', 'blue'}
B = {'yellow', 'red', 'orange'}

print(A.symmetric_difference(B))
# Prints {'orange', 'blue', 'green', 'yellow'}
```



Equivalent Operator ^

Set symmetric difference can be performed with the `^` operator as well.

```
A = {'red', 'green', 'blue'}
B = {'yellow', 'red', 'orange'}

# by method
print(A.symmetric_difference(B))
```

```
# Prints {'orange', 'blue', 'green', 'yellow'}


# by operator
print(A ^ B)
# Prints {'orange', 'blue', 'green', 'yellow'}
```

Symmetric Difference between Multiple Sets

The `symmetric_difference()` method doesn't allow multiple sets.

However, using `^` operator, you can find symmetric difference between multiple sets.

```
A = {'red', 'green', 'blue'}
B = {'yellow', 'orange'}
C = {'blue', 'red', 'black'}


print(A ^ B ^ C)
# Prints {'orange', 'black', 'green', 'yellow'}
```

Python Set symmetric_difference_update() Method

Updates the set by keeping only elements found in either set, but not in both

Usage

The `symmetric_difference_update()` **method updates the set by keeping only elements found in either set, but not in both.**

If you don't want to update the original set, use symmetric_difference() method.

The symmetric difference is actually the union of the two sets, minus their intersection.

Syntax

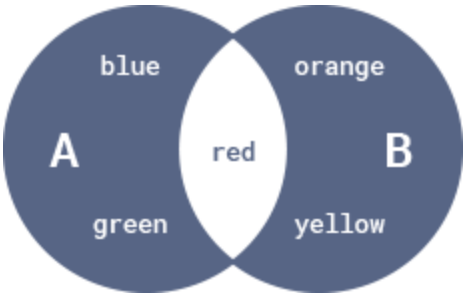
```
set.symmetric_difference_update(set)
```

Parameter	Condition	Description
set	**Required**	**A set to find difference in**

Basic Example

```
# Update A by adding items from B, except common items
A = {'red', 'green', 'blue'}
B = {'yellow', 'red', 'orange'}

A.symmetric_difference_update(B)
print(A)
# Prints {'blue', 'orange', 'green', 'yellow'}
```



Equivalent Operator ^=

You can achieve the same result by using the `^=` augmented assignment operator.

```
A = {'red', 'green', 'blue'}
B = {'yellow', 'red', 'orange'}


A ^= B
print(A)
# Prints {'blue', 'orange', 'green', 'yellow'}
```

Python Set isdisjoint() Method

Determines whether or not two sets have any items in common

Usage

The `isdisjoint()` method returns True if two sets have no items in common, otherwise FALSE.

Sets are disjoint if and only if their intersection is the empty set.

Syntax

set.isdisjoint(set)

Parameter	Condition	Description
set	**Required**	**A set to search for common items in**

Examples

```
# Check if two sets have no items in common
A = {'green', 'blue', 'purple'}
B = {'yellow', 'red', 'orange'}


print(A.isdisjoint(B))
# Prints True
```

The method returns FALSE if the specified sets have any item in common.

```
A = {'red', 'green', 'blue'}
B = {'yellow', 'red', 'orange'}


print(A.isdisjoint(B))
# Prints False
```

Python Set issubset() Method

Determines whether all items in the set are present in the specified set

Usage

The issubset() **method returns True if all items in the set are present in the specified** set**, otherwise FALSE.**

In set theory, every set is a subset of itself.

For example, A.issubset(A) **is True.**

Syntax

```
set.issubset(set)
```

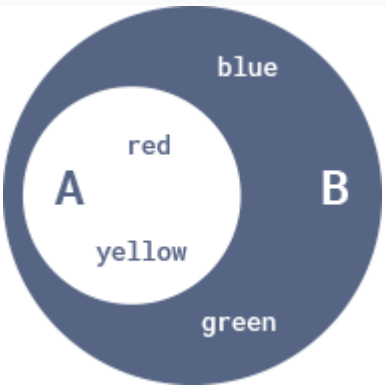
Parameter	Condition	Description
set	**Required**	**A set to search for common items in**

Basic Example

```
# Check if all items in A are present in B
A = {'yellow', 'red'}
```

```
B = {'red', 'green', 'blue', 'yellow'}

print(A.issubset(B))
# Prints True
```



Equivalent Operator <=

You can achieve the same result by using the `<=` comparison operator.

```
A = {'yellow', 'red'}

B = {'red', 'green', 'blue', 'yellow'}

print(A <= B)
# Prints True
```

Find Proper Subset

To test whether the set is a proper subset of other, use `<` comparison operator.

Set A is considered a proper subset of B, if A is a subset of B, but A is not equal to B.

```
# Check if A is a proper subset of B

A = {'yellow', 'red'}

B = {'red', 'green', 'blue', 'yellow'}

print(A < B)

# Prints True
```

```
# Check if A is a proper subset of B
A = {'yellow', 'red'}
B = {'yellow', 'red'}
print(A < B)
# Prints False
```

Python Set issuperset() Method

Determines whether all items in the specified set are present in the original set

Usage

The `issuperset()` **method returns True if all items in the specified** `set` **are present in the original set, otherwise FALSE.**

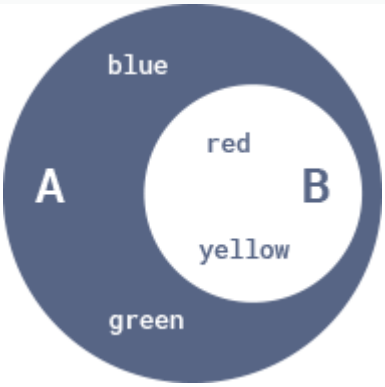
Syntax

```
set.issuperset(set)
```

Parameter	Condition	Description
set	**Required**	**A set to search for common items in**

Basic Example

```
# Check if all items in B are present in A
A = {'red', 'green', 'blue', 'yellow'}
B = {'yellow', 'red'}
print(A.issuperset(B))
# Prints True
```



Equivalent Operator >=

You can achieve the same result by using the `>=` comparison operator.

```
A = {'red', 'green', 'blue', 'yellow'}
B = {'yellow', 'red'}
print(A >= B)
# Prints True
```

Find Proper Superset

To test whether the set is a proper superset of other, use `>` comparison operator.

Set A is considered a proper superset of B, if A is a superset of B, but A is not equal to B.

```
# Check if A is a proper superset of B
A = {'red', 'green', 'blue', 'yellow'}
B = {'yellow', 'red'}
```

```
print(A > B)
# Prints True


# Check if A is a proper superset of B
A = {'yellow', 'red'}
B = {'yellow', 'red'}
print(A > B)
# Prints False
```

Built-in Functions with Set

Below is a list of all built-in functions that you can use with set objects.

Method	Description
all()	**Returns True if all items in a set are true**
any()	**Returns True if any item in a set is true**
enumerate()	**Takes a set and returns an enumerate object**
len()	**Returns the number of items in the set**
max()	**Returns the largest item of the set**
min()	**Returns the smallest item of the set**
sorted()	**Returns a sorted set**
sum()	**Sums items of the set**

Python all() Function

Determines whether all items in an iterable are True

Usage

The `all()` function returns True if all items in an iterable are True. Otherwise, it returns False.

If the iterable is empty, the function returns True.

Syntax

```
all(iterable)
```

Parameter	Condition	Description
iterable	**Required**	**An iterable of type (list, string, tuple, set, dictionary etc.)**

Falsy Values

In Python, all the following values are considered False.

- **Constants defined to be false:** `None` **and** `False`**.**
- **Zero of any numeric type:** `0`**,** `0.0`**,** `0j`**,** `Decimal(0)`**,** `Fraction(0, 1)`
- **Empty sequences and collections:** `''`**,** `()`**,** `[]`**,** `{}`**,** `set()`**,** `range(0)`

Basic Examples

```
# Check if all items in a list are True

L = [1, 1, 1]
print(all(L))   # Prints True

L = [0, 1, 1]
print(all(L))   # Prints False
```

Here are some scenarios where `all()` returns False.

```
L = [True, 0, 1]
print(all(L))   # Prints False


T = ('', 'red', 'green')
print(all(T))   # Prints False


S = {0j, 3+4j}
print(all(S))   # Prints False
```

all() on a Dictionary

When you use `all()` function on a dictionary, it checks if all the keys are true, not the values.

```
D1 = {0: 'Zero', 1: 'One', 2: 'Two'}
print(all(D1))   # Prints False


D2 = {'Zero': 0, 'One': 1, 'Two': 2}
print(all(D2))   # Prints True
```

all() on Empty Iterable

If the iterable is empty, the function returns True.

```
# empty iterable
L = []
print(all(L))   # Prints True


# iterable with empty items
L = [[], []]
print(all(L))   # Prints False
```

Python any() Function

Determines whether any item in an iterable is True

Usage

The `any()` function returns True if any item in an iterable is True. Otherwise, it returns False.

If the iterable is empty, the function returns False.

Syntax

```
any(iterable)
```

Parameter	Condition	Description
iterable	**Required**	**An iterable of type (list, string, tuple, set, dictionary etc.)**

Falsy Values

In Python, all the following values are considered False.

- **Constants defined to be false:** `None` **and** `False`**.**
- **Zero of any numeric type:** `0`**,** `0.0`**,** `0j`**,** `Decimal(0)`**,** `Fraction(0, 1)`

- **Empty sequences and collections:** `''`, `()`, `[]`, `{}`, `set()`, `range(0)`

Basic Examples

```
# Check if any item in a list is True


L = [0, 0, 0]
print(any(L))   # Prints False


L = [0, 1, 0]
print(any(L))   # Prints True
```

Here are some scenarios where `any()` returns True.

```
L = [False, 0, 1]
print(any(L))   # Prints True


T = ('', [], 'green')
print(any(T))   # Prints True


S = {0j, 3+4j, 0.0}
print(any(S))   # Prints True
```

any() on a Dictionary

When you use `any()` function on a dictionary, it checks if any of the keys is true, not the values.

```
D1 = {0: 'Zero', 0: 'Nil'}
print(any(D1))   # Prints False


D2 = {'Zero': 0, 'Nil': 0}
print(any(D2))   # Prints True
```

any() on Empty Iterable

If the iterable is empty, the function returns False.

```
L = []
print(any(L))   # Prints False
```

Python enumerate() Function

Adds a counter to an iterable

Usage

The enumerate() function adds a counter to an iterable and returns it as an enumerate object.

By default, enumerate() starts counting at 0 but if you add a second argument start, it'll start from that number instead.

Syntax

enumerate(iterable,start)

Parameter	Condition	Description
iterable	**Required**	**An iterable (e.g. list, tuple, string etc.)**

start	**Optional**	**A number to start counting from. Default is 0.**

Basic Example

```
# Create a list that can be enumerated
L = ['red', 'green', 'blue']
x = list(enumerate(L))
print(x)
# Prints [(0, 'red'), (1, 'green'), (2, 'blue')]
```

Specify Different Start

By default, `enumerate()` starts counting at 0 but if you add a second argument `start`, it'll start from that number instead.

```
# Start counter from 10
L = ['red', 'green', 'blue']
x = list(enumerate(L, 10))
print(x)
# Prints [(10, 'red'), (11, 'green'), (12, 'blue')]
```

Iterate Enumerate Object

When you iterate an enumerate object, you get a tuple containing (counter, item)

```
L = ['red', 'green', 'blue']
for pair in enumerate(L):
    print(pair)
# Prints (0, 'red')
# Prints (1, 'green')
```

```
# Prints (2, 'blue')
```

You can unpack the tuple into multiple variables as well.

```
L = ['red', 'green', 'blue']

for index, item in enumerate(L):

    print(index, item)

# Prints 0 red

# Prints 1 green
# Prints 2 blue
```

Python len() Function

Returns the number of items of an object

Usage

The `len()` **function returns the number of items of an** object**.**

The object **may be a sequence (such as a string, tuple, list, or range) or a collection (such as a dictionary, set, or frozen set).**

Syntax

```
len(object)
```

Parameter	Condition	Description

object	**Required**	**A sequence or a collection.**

len() on Sequences

```
# number of characters in a string

S = 'Python'

x = len(S)

print(x)
# Prints 6

# number of items in a list

L = ['red', 'green', 'blue']

x = len(L)

print(x)
# Prints 3

# number of items in a tuple

T = ('red', 'green', 'blue')

x = len(T)

print(x)
# Prints 3
```

len() on Collections

```
# number of key:value pairs in a dictionary

D = {'name': 'Bob', 'age': 25}

x = len(D)

print(x)
# Prints 2

# number of items in a set

S = {'red', 'green', 'blue'}

x = len(S)
```

```
print(x)
# Prints 3
```

Python max() Function

Returns the largest item

Usage

The `max()` **function can find**

- **the largest of two or more values (such as numbers, strings etc.)**
- **the largest item in an iterable (such as list, tuple etc.)**

With optional `key` **parameter, you can specify custom comparison criteria to find maximum value.**

Syntax

```
max(val1,val2,val3… ,key)
```

Parameter	Condition	Description
val1,val2,val3…	**Required**	**Two or more values to compare**

key	**Optional**	**A function to specify the comparison criteria. Default value is None.**

– OR –

```
max(iterable,key,default)
```

Parameter	Condition	Description
iterable	**Required**	**Any iterable, with one or more items to compare**
key	**Optional**	**A function to specify the comparison criteria. Default value is None.**
default	**Optional**	**A value to return if the iterable is empty. Default value is False.**

Find Maximum of Two or More Values

If you specify two or more values, the largest value is returned.

```
x = max(10, 20, 30)

print(x)
# Prints 30
```

If the values are strings, the string with the highest value in alphabetical order is returned.

```
x = max('red', 'green', 'blue')

print(x)
# Prints red
```

You have to specify minimum two values to compare. Otherwise, TypeError exception is raised.

Find Maximum in an Iterable

If you specify an Iterable (such as list, tuple, set etc.), the largest item in that iterable is returned.

```
L = [300, 500, 100, 400, 200]

x = max(L)

print(x)
# Prints 500
```

If the iterable is empty, a ValueError is raised.

```
L = []

x = max(L)

print(x)
# Triggers ValueError: max() arg is an empty sequence
```

To avoid such exception, add default **parameter. The** default **parameter specifies a value to return if the provided iterable is empty.**

```
# Specify default value '0'

L = []

x = max(L, default='0')

print(x)
```

```
# Prints 0
```

Find Maximum with Built-in Function

With optional key parameter, you can specify custom comparison criteria to find maximum value. A key parameter specifies a function to be executed on each iterable's item before making comparisons.

For example, with a list of strings, specifying key=len (the built-in len() function) finds longest string.

```
L = ['red', 'green', 'blue', 'black', 'orange']

x = max(L, key=len)

print(x)
# Prints orange
```

Python min() Function

Returns the smallest item

Usage

The min() function can find

- **the smallest of two or more values (such as numbers, strings etc.)**
- **the smallest item in an iterable (such as list, tuple etc.)**

With optional key parameter, you can specify custom comparison criteria to find minimum value.

Syntax

```
min(val1,val2,val3… ,key)
```

Parameter	Condition	Description
val1,val2,val3…	**Required**	**Two or more values to compare**
key	**Optional**	**A function to specify the comparison criteria. Default value is None.**

– OR –

min(iterable,key,default)

Parameter	Condition	Description
iterable	**Required**	**Any iterable, with one or more items to compare**
key	**Optional**	**A function to specify the comparison criteria. Default value is None.**

default	**Optional**	**A value to return if the iterable is empty. Default value is False.**

Find Minimum of Two or More Values

If you specify two or more values, the smallest value is returned.

```
x = min(10, 20, 30)

print(x)
# Prints 10
```

If the values are strings, the string with the lowest value in alphabetical order is returned.

```
x = min('red', 'green', 'blue')

print(x)
# Prints blue
```

You have to specify minimum two values to compare. Otherwise, TypeError exception is raised.

Find Minimum in an Iterable

If you specify an Iterable (such as list, tuple, set etc.), the smallest item in that iterable is returned.

```
L = [300, 500, 100, 400, 200]

x = min(L)

print(x)
# Prints 100
```

If the iterable is empty, a ValueError is raised.

```
L = []
```

```
x = min(L)

print(x)
# Triggers ValueError: min() arg is an empty sequence
```

To avoid such exception, add default parameter. The default parameter specifies a value to return if the provided iterable is empty.

```
# Specify default value '0'

L = []

x = min(L, default='0')

print(x)
# Prints 0
```

Find Minimum with Built-in Function

With optional key parameter, you can specify custom comparison criteria to find minimum value. A key parameter specifies a function to be executed on each iterable's item before making comparisons.

For example, with a list of strings, specifying key=len (the built-in len() function) finds shortest string.

```
L = ['red', 'green', 'blue']

x = min(L, key=len)

print(x)
# Prints red
```

Python sorted() Function

Sorts the items of an iterable

Usage

The sorted() method sorts the items of any iterable

You can optionally specify parameters for sort customization like sorting order and sorting criteria.

Syntax

```
sorted(iterable,key,reverse)
```

The method has two optional arguments, which must be specified as keyword arguments.

Parameter	Condition	Description
iterable	**Required**	**Any iterable (list, tuple, dictionary, set etc.) to sort.**
key	**Optional**	**A function to specify the sorting criteria. Default value is None.**
reverse	**Optional**	**Settting it to True sorts the list in reverse order. Default value is False.**

Return Value

The method returns a new sorted list from the items in iterable**.**

Sort Iterables

`sorted()` **function accepts any** iterable **like list, tuple, dictionary, set, string etc.**

```
# strings are sorted alphabetically
L = ['red', 'green', 'blue', 'orange']
x = sorted(L)
print(x)
# Prints ['blue', 'green', 'orange', 'red']


# numbers are sorted numerically
L = [42, 99, 1, 12]
x = sorted(L)
print(x)
# Prints [1, 12, 42, 99]
```

If you want to sort the list in-place, use built-in sort() method.

sort() **is actually faster than** sorted() **as it doesn't need to create a new list.**

```
# Sort a tuple
L = ('cc', 'aa', 'dd', 'bb')
x = sorted(L)
print(x)
# Prints ['aa', 'bb', 'cc', 'dd']
```

`sorted()` **function sorts a dictionary by keys, by default.**

```
D = {'Bob':30, 'Sam':25, 'Max':35, 'Tom':20}
x = sorted(D)
print(x)
# Prints ['Bob', 'Max', 'Sam', 'Tom']
```

To sort a dictionary by values use the `sorted()` **function along with the values() method.**

```
D = {'Bob':30, 'Sam':25, 'Max':35, 'Tom':20}
```

```
x = sorted(D.values())

print(x)
# Prints [20, 25, 30, 35]
```

Sort in Reverse Order

You can also sort an iterable in reverse order by setting reverse to true.

```
L = ['cc', 'aa', 'dd', 'bb']

x = sorted(L, reverse=True)

print(x)
# Prints ['dd', 'cc', 'bb', 'aa']
```

Sort with Key

Use key parameter for more complex custom sorting. A key parameter specifies a function to be executed on each list item before making comparisons.

For example, with a list of strings, specifying `key=len` (the built-in len() function) sorts the strings by length, from shortest to longest.

```
L = ['orange', 'red', 'green', 'blue']

x = sorted(L, key=len)

print(x)
# Prints ['red', 'blue', 'green', 'orange']
```

Python sum() Function

Sums items of an iterable

Usage

The `sum()` function sums the items of an iterable and returns the total.

If you specify an optional parameter start, it will be added to the final sum.

This function is created specifically for numeric values. For other values, it will raise TypeError.

Syntax

```
sum(iterable,start)
```

Parameter	Condition	Description
iterable	**Required**	**An iterable (such as list, tuple etc.)**
start	**Optional**	**A value to be added to the final sum. Default is 0.**

Examples

```
# Return the sum of all items in a list
L = [1, 2, 3, 4, 5]
x = sum(L)
print(x)
# Prints 15
```

If you specify an optional parameter start, it will be added to the final sum.

```
# Start with '10' and add all items in a list
```

```
L = [1, 2, 3, 4, 5]

x = sum(L, 10)

print(x)
# Prints 25
```