

```
1. 1
1. list_name.append(element) about:blank
```

append()
The `append()` method is used to add an element to the end of a list.

Copied!

Example:

```
1. 1
2. 2
```

```
1. fruits = ["apple", "banana", "orange"]
2. fruits.append("mango") print(fruits)
```

Copied!

Example 1:

copy()
The `copy()` method is used to create a shallow copy of a list.

```
1. 1
2. 2
3. 3
```

```
1. my_list = [1, 2, 3, 4, 5]
2. new_list = my_list.copy() print(new_list)
3. # Output: [1, 2, 3, 4, 5]
```

count()
The `count()` method is used to count the number of occurrences of a specific element in a list in Python.

Copied!

Example:

```
1. 1
2. 2
3. 3
```

```
1. my_list = [1, 2, 2, 3, 4, 2, 5, 2]
2. count = my_list.count(2) print(count)
3. # Output: 4
```

Creating a list
A list is a built-in data type that represents an ordered and mutable collection of elements. Lists are enclosed in square brackets [] and elements are separated by commas.

Example:

```
1. 1
```

```
1. fruits = ["apple", "banana", "orange", "mango"]
```

Copied!

del
The `del` statement is used to

Example:

```
1. 1
```

| | | |
|------------------|--|---|
| extend() | The `extend()` method is used to add multiple elements to a list. It takes an iterable (such as another list, tuple, or string) and appends each element of the iterable to the original list. | <div>1. 1</div> <div>1. list_name.extend(iterable)about:blank</div> <div>Copied!</div> |
| | | <div>Example:</div> <div>1. 1</div> <div>2. 2</div> <div>3. 3</div> <div>4. 4</div> <div>1. fruits = ["apple", "banana", "orange"]</div> <div>2. more_fruits = ["mango", "grape"]</div> <div>3. fruits.extend(more_fruits)</div> <div>4. print(fruits)</div> <div>Copied!</div> |
| Indexing | Indexing in a list allows you to access individual elements by their position. In Python, indexing starts from 0 for the first element and goes up to `length_of_list - 1`. | <div>Example:</div> <div>1. 1</div> <div>2. 2</div> <div>3. 3</div> <div>4. 4</div> <div>5. 5</div> <div>1. my_list = [10, 20, 30, 40, 50]</div> <div>2. print(my_list[0])</div> <div>3. # Output: 10 (accessing the first element)</div> <div>4. print(my_list[-1])</div> <div>5. # Output: 50 (accessing the last element using negative indexing)</div> <div>Copied!</div> |
| | | <div>Syntax:</div> <div>1. 1</div> <div>1. list_name.insert(index, element)</div> <div>Copied!</div> |
| insert() | The `insert()` method is used to insert an element. | <div>Example:</div> <div>1. 1</div> <div>2. 2</div> <div>3. 3</div> <div>1. my_list = [1, 2, 3, 4, 5]</div> <div>2. my_list.insert(2, 6)</div> <div>3. print(my_list)</div> <div>Copied!</div> |
| | | <div>Example:</div> <div>1. 1</div> <div>2. 2</div> <div>3. 3</div> |
| Modifying a list | You can use indexing to modify or assign new | |

pop()

``pop()`` method is another way to remove an element from a list in Python. It removes and returns the element at the specified index. If you don't provide an index to the ``pop()`` method, it will remove and return the last element of the list by default

```
3. 3
4. 4
5. 5
6. 6
7. 7

1. my_list = [10, 20, 30, 40, 50]
2. removed_element = my_list.pop(2) # Removes and returns the element at index 2
3. print(removed_element)
4. # Output: 30
5.
6. print(my_list)
7. # Output: [10, 20, 40, 50]
```

Copied!

Example 2:

```
1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7

1. my_list = [10, 20, 30, 40, 50]
2. removed_element = my_list.pop() # Removes and returns the last element
3. print(removed_element)
4. # Output: 50
5.
6. print(my_list)
7. # Output: [10, 20, 30, 40]
```

Copied!

remove()

To remove an element from a list. The ``remove()`` method removes the first occurrence of the specified value.

Example:

```
1. 1
2. 2
3. 3
4. 4

1. my_list = [10, 20, 30, 40, 50]
2. my_list.remove(30) # Removes the element 30
3. print(my_list)
4. # Output: [10, 20, 40, 50]
```

Copied!

reverse()

The ``reverse()`` method is used to reverse the order of elements in a list

Example 1:

```
1. 1
2. 2
3. 3

1. my_list = [1, 2, 3, 4, 5]
2. my_list.reverse() print(my_list)
3. # Output: [5, 4, 3, 2, 1]
```

Copied!

Slicing

You can use slicing to access a range of elements from a list.

```
1. 3
2. 4
3. 5
4. 6
5. 7
6. 8
7. 9
8. 10
9. 11
10. 12
11.
12.

1. my_list = [1, 2, 3, 4, 5]
2. print(my_list[1:4])
3. # Output: [2, 3, 4] (elements from index 1 to 3)
4.
5. print(my_list[:3])
6. # Output: [1, 2, 3] (elements from the beginning up to index 2)
7.
8. print(my_list[2:])
9. # Output: [3, 4, 5] (elements from index 2 to the end)
10.
11. print(my_list[::-2])
12. # Output: [1, 3, 5] (every second element)
```

Copied!

Example 1:

```
1. 1
2. 2
3. 3
4. 4

1. my_list = [5, 2, 8, 1, 9]
2. my_list.sort()
3. print(my_list)
4. # Output: [1, 2, 5, 8, 9]
```

Copied!

sort()

The `sort()` method is used to sort the elements of a list in ascending order. If you want to sort the list in descending order, you can pass the `reverse=True` argument to the `sort()` method.

Example 2:

```
1. 1
2. 2
3. 3
4. 4

1. my_list = [5, 2, 8, 1, 9]
2. my_list.sort(reverse=True)
3. print(my_list)
4. # Output: [9, 8, 5, 2, 1]
```

Copied!

Dictionary

| Package/Method | Description | Code Example |
|------------------|---|-----------------|
| Accessing Values | You can access the values in a dictionary using | Syntax: 1. 1 |

Code Example

Copied!

about:blank

Syntax:

```
1. 1
1. dict_name[key] = value
```

Copied!

Add or modify

Inserts a new key-value pair into the dictionary. If the key already exists, the value will be updated; otherwise, a new entry is created.

Example:

```
1. 1
2. 2

1. person["Country"] = "USA" # A new entry will be created.
2. person["city"] = "Chicago" # Update the existing value for the same key
```

Copied!

Syntax:

```
1. 1
1. dict_name.clear()
```

Copied!

clear()

The 'clear()' method empties the dictionary, removing all key-value pairs within it. After this operation, the dictionary is still accessible and can be used further.

Example:

```
1. 1
1. grades.clear()
```

Copied!

Syntax:

```
1. 1
1. new_dict = dict_name.copy()
```

Copied!

copy()

Creates a shallow copy of the dictionary. The new dictionary contains the same key-value pairs as the original, but they remain distinct objects in memory.

Example:

```
1. 1
2. 2

1. new_person = person.copy()
2. new_person = dict(person) # another way to create a copy of dictionary
```

Copied!

Creating a Dictionary

A dictionary is a built-in data type that represents a collection of key-value pairs. Dictionaries are

Example:

```
1. 1
2. 2

1. dict_name = {} #Creates an empty dictionary
2. person = { "name": "John", "age": 30, "city": "New York"}
```

KeyError: If the key does not exist.

```
1. 1
    about:blank
1. del person["Country"]
```

Copied!

Syntax:

```
1. 1
1. items_list = list(dict_name.items())
```

Copied!

items()

Retrieves all key-value pairs as tuples and converts them into a list of tuples. Each tuple consists of a key and its corresponding value.

Example:

```
1. 1
1. info = list(person.items())
```

Copied!

Example:

key existence

You can check for the existence of a key in a dictionary using the 'in' keyword

```
1. 1
2. 2
1. if "name" in person:
2.     print("Name exists in the dictionary.")
```

Copied!

Syntax:

```
1. 1
1. keys_list = list(dict_name.keys())
```

Copied!

keys()

Retrieves all keys from the dictionary and converts them into a list. Useful for iterating or processing keys using list methods.

Example:

```
1. 1
1. person_keys = list(person.keys())
```

Copied!

update()

The 'update()' method merges the provided dictionary into the existing dictionary, adding or updating key-value pairs.

Syntax:

```
1. 1
1. dict_name.update({key: value})
```

Copied!

Example:

```
1. 1
```

can be used for further processing or analysis.

Example:

```
1. 1
```

about:blank

```
1. person_values = list(person.values())
```

Copied!

Sets

| Package/Method | Description | Code Example |
|----------------|--|---|
| | | Syntax: <pre>1. 1</pre> <pre>1. set_name.add(element)</pre> <p>Copied!</p> <p>Example:</p> <pre>1. 1</pre> <pre>1. fruits.add("mango")</pre> <p>Copied!</p> |
| add() | Elements can be added to a set using the `add()` method. Duplicates are automatically removed, as sets only store unique values. | |
| | | Syntax: <pre>1. 1</pre> <pre>1. set_name.clear()</pre> <p>Copied!</p> <p>Example:</p> <pre>1. 1</pre> <pre>1. fruits.clear()</pre> <p>Copied!</p> |
| clear() | The `clear()` method removes all elements from the set, resulting in an empty set. It updates the set in-place. | |
| | | Syntax: <pre>1. 1</pre> <pre>1. new_set = set_name.copy()</pre> <p>Copied!</p> <p>Example:</p> <pre>1. 1</pre> <pre>1. new_fruits = fruits.copy()</pre> <p>Copied!</p> |
| copy() | The `copy()` method creates a shallow copy of the set. Any modifications to the copy won't affect the original set. | |
| | | Example: <pre>1. 1</pre> <pre>1. new_fruits = fruits.copy()</pre> <p>Copied!</p> |
| Defining Sets | A set is an unordered collection of unique elements. Sets are enclosed in curly braces `{}`. | Example: |

Use the `discard()` method to remove a specific element from the set. Ignores if the element is not found.

Copied!

Example:

```
1. 1  
  
1. fruits.discard("apple")
```

Copied!

Syntax:

```
1. 1  
  
1. is_subset = set1.issubset(set2)
```

Copied!

Example:

```
1. 1  
  
1. is_subset = fruits.issubset(colors)
```

Copied!

Syntax:

```
1. 1  
  
1. is_superset = set1.issuperset(set2)
```

Copied!

Example:

```
1. 1  
  
1. is_superset = colors.issuperset(fruits)
```

Copied!

Syntax:

```
1. 1  
  
1. removed_element = set_name.pop()
```

Copied!

Example:

```
1. 1  
  
1. removed_fruit = fruits.pop()
```

Copied!

issubset()

The `issubset()` method checks if the current set is a subset of another set. It returns True if all elements of the current set are present in the other set, otherwise False.

issuperset()

The `issuperset()` method checks if the current set is a superset of another set. It returns True if all elements of the other set are present in the current set, otherwise False.

pop()

The `pop()` method removes and returns an arbitrary element from the set. It raises a `KeyError` if the set is empty. Use this method to remove elements when the order doesn't matter.


```
1. fruits.remove("banana")
```

about:blank

Copied!

Syntax:

- 1. 1
- 2. 2
- 3. 3
- 4. 4

- 1. union_set = set1.union(set2)
- 2. intersection_set = set1.intersection(set2)
- 3. difference_set = set1.difference(set2)
- 4. sym_diff_set = set1.symmetric_difference(set2)

Copied!

Example:

- 1. 1
- 2. 2
- 3. 3
- 4. 4

- 1. combined = fruits.union(colors)
- 2. common = fruits.intersection(colors)
- 3. unique_to_fruits = fruits.difference(colors)
- 4. sym_diff = fruits.symmetric_difference(colors)

Copied!

Syntax:

- 1. 1

- 1. set_name.update(iterable)

Copied!

Example:

- 1. 1

- 1. fruits.update(["kiwi", "grape"])

Copied!

Set Operations

Perform various operations on sets: `union`, `intersection`, `difference`, `symmetric difference`.

update()

The `update()` method adds elements from another iterable into the set. It maintains the uniqueness of elements.