Python Data Structures Cheat Sheet

List

Package/Method	Description	Code Example
append()	The `append()` method is used to add an element to the end of a list.	Syntax: `list_name.append(element)` Example: `fruits = ["apple", "banana", "orange"]` `fruits.append("mango") print(fruits)`
copy()	The `copy()` method is used to create a shallow copy of a list.	Example 1: `my_list = [1, 2, 3, 4, 5]` `new_list = my_list.copy() print(new_list)` `# 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.	Example: `my_list = [1, 2, 2, 3, 4, 2, 5, 2]` `count = my_list.count(2) print(count)` `# 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: `fruits = ["apple", "banana", "orange", "mango"]`
del	The `del` statement is used to remove an element from list. `del` statement removes the element at the specified index.	Example: `my_list = [10, 20, 30, 40, 50]` `del my_list[2] # Removes the element at index 2 print(my_list)` `# Output: [10, 20, 40, 50]`
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.	Syntax: `list_name.extend(iterable)` Example: `fruits = ["apple", "banana", "orange"]` `more_fruits = ["mango", "grape"]` `fruits.extend(more_fruits)` `print(fruits)`
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`.	Example: `my_list = [10, 20, 30, 40, 50]` `print(my_list[0])` `# Output: 10 (accessing the first element)` `print(my_list[-1])` `# Output: 50 (accessing the last element using negative indexing`

insert()	The `insert()` method is used to insert an element.	Syntax:
		`list_name.insert(index, element)`
		
		Example:
		`my_list = [1, 2, 3, 4, 5]`
		`my_list.insert(2, 6)`
		`print(my_list)`

Modifying a list	You can use indexing to modify or assign new values to specific elements in the list.	Example:
		`my_list = [10, 20, 30, 40, 50]`
		`my_list[1] = 25 # Modifying the second element`
		`print(my_list)`
		`# Output: [10, 25, 30, 40, 50]`

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	Example 1:
		`my_list = [10, 20, 30, 40, 50]`
		`removed_element = my_list.pop(2) # Removes and returns the eleme`
		`print(removed_element)`
		`# Output: 30`
		
		`print(my_list)`
		`# Output: [10, 20, 40, 50]`
		
		Example 2:
		`my_list = [10, 20, 30, 40, 50]`
		`removed_element = my_list.pop() # Removes and returns the last e`
		`print(removed_element)`
		`# Output: 50`
		
		`print(my_list)`
		`# Output: [10, 20, 30, 40]`

remove()	To remove an element from a list. The `remove()` method removes the first occurrence of the specified value.	Example:
		`my_list = [10, 20, 30, 40, 50]`
		`my_list.remove(30) # Removes the element 30`
		`print(my_list)`
		`# Output: [10, 20, 40, 50]`

reverse()	The `reverse()` method is used to reverse the order of elements in a list	Example 1:
		`my_list = [1, 2, 3, 4, 5]`
		`my_list.reverse() print(my_list)`
		`# Output: [5, 4, 3, 2, 1]`

Slicing	You can use slicing to access a range of elements from a list.	Syntax:
		`list_name[start:end:step]`
		
		Example:
		`my_list = [1, 2, 3, 4, 5]`
		`print(my_list[1:4])`
		`# Output: [2, 3, 4] (elements from index 1 to 3)`
		
		`print(my_list[:3])`
		`# Output: [1, 2, 3] (elements from the beginning up to index 2)`
		
		`print(my_list[2:])`
		`# Output: [3, 4, 5] (elements from index 2 to the end)`

		`print(my_list[::2])`
		`# Output: [1, 3, 5] (every second element)`

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

Dictionary

Package/Method	Description	Code Example

Accessing Values	You can access the values in a dictionary using their corresponding `keys`.	Syntax:
		`Value = dict_name["key_name"]`
		
		Example:
		`name = person["name"]`
		`age = person["age"]`

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.	Syntax:
		`dict_name[key] = value`
		
		Example:
		`person["Country"] = "USA" # A new entry will be created.`
		`person["city"] = "Chicago" # Update the existing value fc`

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.	Syntax:
		`dict_name.clear()`
		
		Example:
		`grades.clear()`

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.	Syntax:
		`new_dict = dict_name.copy()`
		
		Example:
		`new_person = person.copy()`
		`new_person = dict(person) # another way to create a copy`

Creating a Dictionary	A dictionary is a built-in data type that represents a collection of key-value pairs. Dictionaries are enclosed in curly braces `{}`.	Example:
		`dict_name = {} #Creates an empty dictionary`
		`person = { "name": "John", "age": 30, "city": "New York"}`

del	Removes the specified key-value pair from the dictionary. Raises a `KeyError` if the key does not exist.	Syntax: `del dict_name[key]` Example:

		`del person["Country"]`

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.	Syntax:
		`items_list = list(dict_name.items())`
		Example:
		`info = list(person.items())`

key existence	You can check for the existence of a key in a dictionary using the `in` keyword	Example:
		`if "name" in person:`
		`    print("Name exists in the dictionary.")`

keys()	Retrieves all keys from the dictionary and converts them into a list. Useful for iterating or processing keys using list methods.	Syntax:
		`keys_list = list(dict_name.keys())`
		Example:
		`person_keys = list(person.keys())`

update()	The `update()` method merges the provided dictionary into the existing dictionary, adding or updating key-value pairs.	Syntax:
		`dict_name.update({key: value})`
		Example:
		`person.update({"Profession": "Doctor"})`

values()	Extracts all values from the dictionary and converts them into a list. This list can be used for further processing or analysis.	Syntax:
		`values_list = list(dict_name.values())`
		Example:
		`person_values = list(person.values())`

Sets

Package/Method	Description	Code Example

add()	Elements can be added to a set using the `add()` method. Duplicates are automatically removed, as sets only store unique values.	Syntax:
		`set_name.add(element)`
		Example:
		`fruits.add("mango")`

clear()	The `clear()` method removes all elements from the set, resulting in an empty set. It updates the set in-place.	Syntax:
		`set_name.clear()`
		Example:
		`fruits.clear()`

copy()	The `copy()` method creates a shallow copy of the set. Any modifications to the copy won't affect the original set.	Syntax:
		`new_set = set_name.copy()`
		Example:
		`new_fruits = fruits.copy()`

Defining Sets	A set is an unordered collection of unique elements. Sets are enclosed in curly braces `{}`. They are useful for storing distinct values and performing set operations.	Example:

		`empty_set = set() #Creating an E`
		`fruits = {"apple", "banana", "or`

discard()	Use the `discard()` method to remove a specific element from the set. Ignores if the element is not found.	Syntax:
		`set_name.discard(element)`
		Example:
		`fruits.discard("apple")`

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.	Syntax:
		`is_subset = set1.issubset(set2)`
		Example:
		`is_subset = fruits.issubset(colc`

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.	Syntax:
		`is_superset = set1.issuperset(se`
		Example:
		`is_superset = colors.issuperset(`

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.	Syntax:
		`removed_element = set_name.pop()`
		Example:
		`removed_fruit = fruits.pop()`

remove()	Use the `remove()` method to remove a specific element from the set. Raises a `KeyError` if the element is not found.	Syntax:
		`set_name.remove(element)`
		Example:
		`fruits.remove("banana")`

Set Operations	Perform various operations on sets: `union`, `intersection`, `difference`, `symmetric difference`.	Syntax:
		`union_set = set1.union(set2)`
		`intersection_set = set1.intersec`
		`difference_set = set1.difference`
		`sym_diff_set = set1.symmetric_di`
		Example:
		`combined = fruits.union(colors)`
		`common = fruits.intersection(col`
		`unique_to_fruits = fruits.differ`
		`sym_diff = fruits.symmetric_diff`

update()	The `update()` method adds elements from another iterable into the set. It maintains the uniqueness of elements.	Syntax:
		`set_name.update(iterable)`
		Example:
		`fruits.update(["kiwi", "grape"]`