Python Data Structures Cheat Sheet: The Essential Guide

December 26, 2023 • Cassandra Lee

Do you want to pick out elements from a list of objects but forgot how to do list comprehension? Or maybe you have a JavaScript Object Notation (JSON) file and need to find a suitable data structure (such as Python's dictionary data type) to process the data in it. Browsing app development forums to find a helpful piece of Python code can be frustrating.

The good news is we've prepared this cheat sheet for people like you. It doubles as a refresher on data structures and algorithms as applied to Python. Keep a copy of this Python data structures cheat sheet on your desk to look up commands or code snippets the next time you need to recall them.

This Python data structures cheat sheet covers the theoretical essentials. Download the PDF version **here**.

Python Data Structures Cheat Sheet Search

Search our Python data structures cheat sheet to find the right cheat for the term you're looking for. Simply enter the term in the search bar and you'll receive the matching cheats available.

Search cheats here

Types of Data Structures in Python

Here's a diagram to illustrate the hierarchy of Python data structures:

Python Primitive Data Structures

These store simple data values.

DESCRIPTION		EXAMPLES
String	Collection of characters surrounded by single or double quotation marks	`'Alice', "Bob"`
Boolean	Logical values	`True, False`
Integer	Whole number of unlimited length	`0, -273`
Float	Floating-point decimal	`1.618, 3.1415926`

Python Built-In Non-Primitive Data Structures

These data structures, which store values and collections of values, are inherent to Python.

DESCRIPTION	ORDERED	ALLOW DUPLICATES	MUTABLE	SYNTAX	EXAMPLES
List	√	√	√	`[ ]`	• `[1, 2.3, True]` • `['John', 'Doe']`

Tuple	√	√	×	`( )`	• `('age', 22)` • `(7.89, False)`
Set 0 is the same as False, as are 1 and True.	×	×	×	`{ }`	• `{6, 4.5}` • `{'nice', True}`
Dictionary Map, storing key-value pairs.	√ > 3.7 × ≤ 3.6	×	√	`{key: value}`	• `{"FB": "Facebook", "WA": "WhatsApp", "IG": "Instagram"}` • `{'name': 'Bob', 'id': 255}`

List and Tuple Operations

Note that Python lists and tuples are **zero-indexed**, meaning the first element has an index of 0. See the command chart below for an example.

Accessing Items in Lists

The table below is based on this list:

`fruit_list = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]`.

COMMAND	DESCRIPTION
`fruit_list[0]`	Get the first item on the list ("apple")
`fruit_list[1]`	Get the second item on the list ("banana")
`fruit_list[-1]`	Get the last item ("mango")
`fruit_list[2:5]`	Get the items from start to end indexes
`fruit_list[:4]`	Get the items from the beginning but exclude "kiwi" and beyond
`fruit_list[2:]`	Get the items from "cherry" to the end
`fruit_list[-4:-1]`	Get the items from "orange" (-4) onwards but exclude "mango" (-1)
`if "apple" in fruit_list:` `  print("Yes, we have 'apple'")`	Check if "apple" is in the list

List (and Tuple) Methods

Commands with an asterisk (*) apply to tuples.

COMMAND	DESCRIPTION	USAGE
`append()`	Add an element at the end of the list	`list1.append(element)`
`clear()`	Remove all the elements from the list	`list1.clear()`
`copy()`	Return a copy of the list	`list1.copy()`
`count()`	Return the number of elements with the specified value*	`list1.count(element)`
`extend()`	Add the elements of a list (or any iterable), to the end of the current list	`list1.extend(list2)`
`index()`	Return the index of the first element with the specified value*	`list1.index(element[,start[,end]])`
`insert()`	Add an element at the specified position (position is an integer)	`list1.insert(position, element)`
`pop()`	Remove the element at the specified position	`list1.pop([index])`
`remove()`	Remove the first item with the specified value	`list1.remove(element)`
`reverse()`	Reverse the order of the list	`list1.reverse()`
`sort()` `sort(reverse = True)`	Sort the list in ascending / descending order	`list1.sort()` `list2.sort(reverse = True)`
`del()`	Delete from the list the item specified with its index	`del list1[index]`

`list1 + list2`	Join two lists	`list1 = ["x", "y"]`
		`list2 = [8, 9]`
		`list3 = list1 + list2`
		
		`# Returns: ["x","y",8,9]`

List Comprehension

List comprehension simplifies the creation of a new list based on the values of an existing list.

COMMAND	DESCRIPTION
`[n for n in range(10) if n < 5]`	Accept only numbers less than 5
`[x for x in fruits if "a" in x]`	Accept items containing "a".
`[x for x in fruits if x != "apple"]`	Accept all items except "apple"
`[x.upper() for x in fruits]`	Make uppercase the values in the new list
`[x + '?' for x in fruits]`	Add a question mark at the end of each item
`['hello' for x in fruits]`	Set all values in the new list to 'hello'
`[x if x != "banana" else "orange" for x in fruits]`	Replace "banana" with "orange" in the new list

Accessing Items in Tuples

Below, the tuple in question is `fruits = ("apple", "banana", "cherry")`.

COMMAND	DESCRIPTION
`"apple" in fruits`	Check if "apple" is present in the tuple. This command returns the value True.

`(x, y, z) = fruits`
`# x == "apple"`
`# y == "banana"`
`# z == "cherry"`
	Assign variables to take up each item in the tuple, also known as **unpacking** a tuple.

`(a, *_) = fruits`
`# a == "apple"`
`# _ == ["banana", "cherry"]`
	Either the number of variables must match the number of values in the tuple, or use an asterisk as shown to put away the unwanted values.

Tuple Manipulation

Adding items

You can add items to a tuple as follows:

Initial	`original = ("apple", "banana", "cherry")`
Code	`new_item = ("orange",)` `original += new_item`
Result	`("apple", "banana", "cherry", "orange")`

Tip: When creating a single-item tuple, remember to include a comma.

Removing items and changing values

Since tuples are immutable, you can't remove or modify their contents directly. The key is converting it into a list and back.

EXAMPLE	ADDITION	REMOVAL	CHANGE
Initial	`original = ("apple", "banana", "cherry")`	`original = ("apple", "banana", "cherry")`	`original = ("apple", "banana", "cherry")`
→ List	`tempList = list(original)`	`tempList = list(original)`	`tempList = list(original)`
Code	`tempList.append("orange")`	`tempList.remove("apple")`	`tempList[1] = "kiwi"`
→ Tuple	`newList = tuple(tempList)`	`newList = tuple(tempList)`	`newList = tuple(tempList)`
Result of newList	`("apple", "banana", "cherry", "orange")`	`("banana", "cherry")`	`("kiwi", "banana", "cherry")`

Dictionary Operations

Adding Items

There are three methods:

EXAMPLE	ADDITION #1 (DIRECT)	ADDITION #2 (UPDATE())	ADDITION #3 (**)

Initial	`meta = { "FB": "Facebook", "WA": "WhatsApp", "IG": "Instagram" }`
	
	`new_co = { "GIF": "Giphy" }`
		`meta = { "FB": "Facebook", "WA": "WhatsApp", "IG": "Instagram" }`
		
		`new_co = { "GIF": "Giphy" }`
			`meta = { "FB": "Facebook", "WA": "WhatsApp", "IG": "Instagram" }`
			
			`new_co = { "GIF": "Giphy" }`

Code	`meta["GIF"] = "Giphy"`	`meta.update(new_co)`	`meta = {**meta, **new_co}`
Result of meta	`{ "FB": "Facebook", "WA": "WhatsApp", "IG": "Instagram", "GIF": "Giphy" }`	`{ "FB": "Facebook", "WA": "WhatsApp", "IG": "Instagram", "GIF": "Giphy" }`	`{ "FB": "Facebook", "WA": "WhatsApp", "IG": "Instagram", "GIF": "Giphy" }`

Warning: duplicate keys will cause the latest values to overwrite earlier values.

General Operations

COMMAND	DESCRIPTION	EXAMPLE
`del dict1["key1"]`	Remove the item with the specified key name	`del meta["WA"] # "WhatsApp"`
`del dict1`	Delete the dictionary	`del meta`
`dict1[key1]`	Access the value of a dictionary `dict1` element using its key `key1`	`meta["FB"]# "Facebook"`
Dictionary method	**Description**	**Usage**
`clear()`	Remove all the elements from the dictionary	`dict1.clear()`
`copy()`	Return a copy of the dictionary	`dict1.copy()`
`fromkeys()`	Return a dictionary with the specified keys and value	`dict1.fromkeys(keys, value)`
`get()`	Return the value of the specified key	`dictionary.get(key-_name, value)`
`items()`	Return a list containing a tuple for each key-value pair	`dict1.items()`
`keys()`	Return a list containing the dictionary's keys	`dict1.keys()`
`pop()`	Remove the element with the specified key	`dict1.pop(key_name)`
`popitem()`	Remove the last inserted key-value pair	`dict1.popitem()`
`setdefault()`	Return the value of the specified key. If the key does not exist, add as new key-value pair	`dict1.setdefault(ke-y_name, value)`
`update()`	Update the dictionary with the specified key-value pairs	`dict1.update(iterable)`
`values()`	Return a list of all the values in the dictionary	`dict1.values()`

Set Operations

Accessing

Although you can't directly access items in a set, you can loop through the items:

EXAMPLE	ACCESSING ITEMS IN A SET (USING LIST COMPREHENSION)
Code	`set1 = {32, 1, 2, 27, 83, 26, 59, 60}` `set1_odd = [i for i in set1 if i % 2 == 1]`
Result	`set1_odd = [1, 27, 83, 59]`

Adding and Removing Items

COMMAND	DESCRIPTION	USAGE
`add()`	Add a single element to the set	`fruits.add("orange")`
`update()`	Add elements from another set into this set	`fruits.add({"pineapple", "-mango", "durian"})`
`discard()` `remove()`	Remove the specified element	`fruits.discard("banana")` `fruits.remove("banana")`
`pop()`	Remove the last element in the set. The return value of bye is the removed element.	`bye = fruits.pop()`
`clear()`	Empty the set	`fruits.clear()`
`copy()`	Return a copy of the set	`fruits.copy()`
`del`	Delete the set	`del fruits`

Mathematical Operations

COMMAND / BINARY OPERATOR(S)	DESCRIPTION
`difference()` `-`	Get the difference of several sets
`difference_update()`	Remove the elements in this set that are also included in another, specified set
`intersection()` `&`	Get intersection of sets
`intersection_update()`	Remove the elements in this set that are not present in other, specified set(s)
`isdisjoint()`	Return whether two sets have an intersection
`issubset()` `<, <=`	Check if a set is a (strict <) subset
`issuperset()` `>, >=`	Check if a set is a (strict >) superset
`symmetric_difference()` `^`	Get symmetric difference of two sets
`symmetric_difference_update()`	Insert the symmetric differences from this set and another

`union()` `|`	Get the union of sets

Algorithms and the Complexities

This section is about the complexity classes of various Python data structures.

List

Tuples have the same operations (non-mutable) and complexities.

COMMAND (L: LIST)	COMPLEXITY CLASS
`L.append(item)`	`O(1)`
`L.clear()`	`O(1)`
`item in/not in L`	`O(N)`
`L.copy()`	`O(N)`
`del L[i]`	`O(N)`
`L.extend(…)`	`O(N)`
`L1==L2, L1!=L2`	`O(N)`
`L[i]`	`O(1)`
`for item in L:`	`O(N)`
`len(L)`	`O(1)`
`k*L`	`O(k*N)`
`min(L), max(L)`	`O(N)`
`L.pop(-1)`	`O(1)`
`L.pop(item)`	`O(N)`
`L.remove(…)`	`O(N)`
`L.reverse()`	`O(N)`
`L[x:y]`	`O(y-x)`
`L.sort()`	`O(N*log(N))`
`L[i]=item`	`O(1)`

Dictionary

COMMAND (D: DICTIONARY)	COMPLEXITY CLASS / RANGE (—)
`d.clear()`	`O(1)`
`dict(…)`	`O(len(d))`
`del d[k]`	`O(1) — O(N)`
`d.get()`	`O(1) — O(N)`
`for item in d:`	`O(N)`
`len(d)`	`O(1)`
`d.pop(item)`	`O(1) — O(N)`
`d.popitem()`	`O(1)`
`d.values()`	`O(1)`
`d.keys()`	`O(1)`
`d.fromkeys(seq)`	`O(len(seq))`

Set

OPERATION	COMMAND (S: SET)	COMPLEXITY CLASS / RANGE (—)
Add	`s.add(item)`	`O(1) — O(N)`
Clear	`s.clear()`	`O(1)`
Copy	`s.copy()`	`O(N)`
Containment	`item in/not in s`	`O(1) — O(N)`
Creation	`set(…)`	`O(len(s))`
Discard	`s.discard(item)`	`O(1) — O(N)`
Difference	`s1-s2`	`O(len(s1))`
Difference Update	`s1.difference_update(s2)`	`O(len(s2)) — ∞`
Equality	`s1==s2, s1!=s2`	`O(min(len(s1), len(s2)))`
Intersection	`s1&s2`	`O(min(len(s1), len(s2)))`
Iteration	`for item in s:`	`O(N)`
Is Subset	`s1<=s2`	`O(len(s1))`
Is Superset	`s1>=s2`	`O(len(s2)) — O(len(s1))`
Pop	`s.pop()`	`O(1) — O(N)`
Union	`s1|s2`	`O(len(s1)+len(s2)) — ∞`