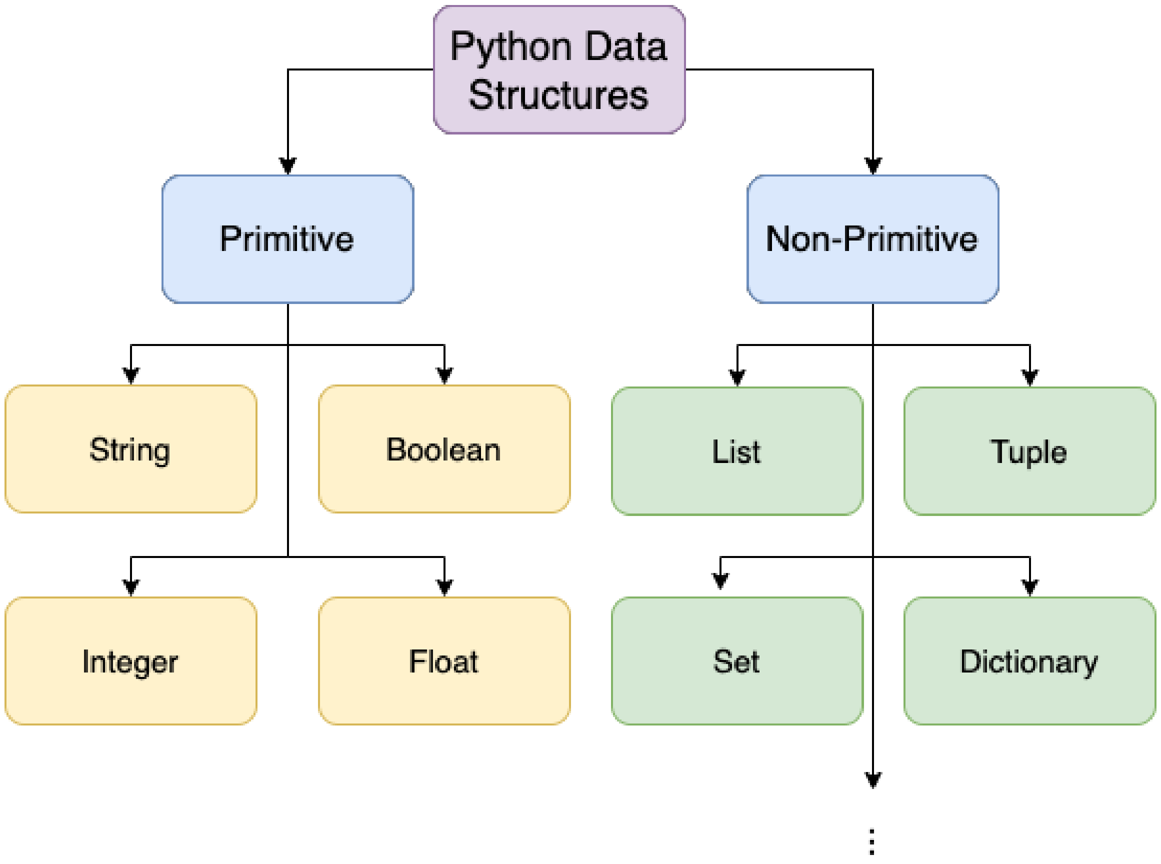
Python Data Structures Cheat Sheet

Types of Data Structures in Python



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.

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[1]`	Get the second item on the list
`fruit_list[-1]`	Get the last item
`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 "-che-rry-" to the end
`fruit_list[--4:-1]`	Get the items from "-ora-nge-" (-4) onwards but exclude "-man-go" (-1)

`if "-app-le" in fruit_list:`
`   print(-"Yes, we have 'apple'")`
	Check if "-app-le" 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.a-ppend(element)`
`clear()`	Remove all the elements from the list	`list1.c-lear()`
`copy()`	Return a copy of the list	`list1.c-opy()`
`count()`	Return the number of elements with the specified value*	`list1.c-oun-t(e-lement)`
`extend()`	Add the elements of a list (or any iterable), to the end of the current list	`list1.e-xt-end-(list2)`
`index()`	Return the index of the first element with the specified value*	`list1.i-nde-x(e-lem-ent-[,s-tar-t[,-end]])`
`insert()`	Add an element at the specified position (position is an integer)	`list1.i-nse-rt(-po-sition, element)`
`pop()`	Remove the element at the specified position	`list1.p-op(-[in-dex])`
`remove()`	Remove the first item with the specified value	`list1.r-emo-ve(-ele-ment)`
`reverse()`	Reverse the order of the list	`list1.re-verse()`

`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 compre-hension 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 != "-app-le"]`	Accept all items except `"-app-le"`
`[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 != "-ban-ana-" else "-ora-nge-" for x in fruits]`	Replace `"-ban-ana-"` with `"-ora-nge-"` in the new list

Accessing Items in Tuples

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

Command	Description
`"-app-le" in fruits` Return: `True`	Check if `"-app-le"` is present in the tuple

`(x, y, z) = fruits`
 	
`# x == "apple"`
`# y == "banana"`
`# z == "cherry"`
 	
`(a, *_) = fruits`
 	
`# a == "apple"`
`# _ == ["banana", "cherry"]`
	Assign variables to take up each item in the tuple, also known as **unpacking** a tuple.
	
	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 = ("ap-ple-", "-ban-ana-", "-che-rry-")`

Code	`new_item = ("or-ang-e",)`
	`original += new_item`

Result	`("ap-ple-", "-ban-ana-", "-che-rry-", "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 = ("ap-ple-", "-ban-ana-", "-che-rry-")`		
→ List	`tempList = list(original)`		
Code	`tempList.appe-nd("-ora-nge-")`	`tempList.remo-ve("-app-le")`	`tempList[1] = "-kiw-i"`
→ Tuple	`newList = tuple(tempList)`		
Result of `newList`	`("ap-ple-", "-ban-ana-", "-che-rry-", "orange")`	`("-ban-ana-", "-che-rry-")`	`("kiwi", "-ban-ana-", "-che-rry-")`

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" }`		
Code	`meta[-"GIF"] = "Giphy"`	`meta.update(new_co)`	`meta = {**meta, **new_co}`
Result of `meta`	`{ "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 di-ct1[-"key1"]`	Remove the item with the specified key name	`del meta[-"WA"] # "WhatsApp"`
`del di-ct1`	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.c-lear()`
`copy()`	Return a copy of the dictionary	`dict1.c-opy()`

`fromkeys()`	Return a dictionary with the specified keys and value	`dict1.f-rom-key-s(keys, value)`
`get()`	Return the value of the specified key	`dictio-nar-y.g-et(-key-_name, value)`
`items()`	Return a list containing a tuple for each key-value pair	`dict1.i-tems()`
`keys()`	Return a list containing the dictio-nary’s keys	`dict1.k-eys()`
`pop()`	Remove the element with the specified key	`dict1.p-op(ke-y_name)`
`popitem()`	Remove the last inserted key-value pair	`dict1.p-opi-tem()`
`setdef-ault()`	Return the value of the specified key. If the key does not exist, add as new key-value pair	`dict1.s-etd-efa-ult-(ke-y_name, value)`
`update()`	Update the dictionary with the specified key-value pairs	`dict1.u-pda-te(-ite-rable)`
`values()`	Return a list of all the values in the dictionary	`dict1.v-alues()`

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.a-dd(-"-ora-nge-")`
`update()`	Add elements from another set into this set	`fruits.a-dd(-{"pi-nea-ppl-e", "-man-go", "durian"})`

`discard()`
`remove()`
	Remove the specified element	`fruits.d-isc-ard-("ba-nan-a")`
		`fruits.r-emo-ve(-"-ban-ana-")`

`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

`differ-ence()`
`-`
	Get the difference of several sets

`differ-enc-e_u-pdate()`	Remove the elements in this set that are also included in another, specified set

`inters-ect-ion()`
`&`
	Get intersection of sets

`inters-ect-ion-_up-date()`	Remove the elements in this set that are not present in other, specified set(s)
`isdisj-oint()`	Return whether two sets have an inters-ection

`issubset()`
`<, <=`
	Check if a set is a (strict <) subset

`issupe-rset()`
`>, >=`
	Check if a set is a (strict >) superset

`symmet-ric-_di-ffe-rence()`
`^`
	Get symmetric differ-ence of two sets

`symmet-ric-_di-ffe-ren-ce_-upd-ate()`	Insert the symmetric differ-ences from this set and another

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

Algorithms and the Complexities

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)) — ∞
Symmetric Difference	`s1^s2`	`O(len(s1)) – O(len(s1)*len(s2))`

Symbol Table

Python keeps track of variables using symbol tables, which you can explore using the following basic commands:

Command	Description
`__doc__`	Program documentation as comments (`#, """, '''`) at the beginning of the code
`__name__`	How you run the Python program.

	Direct execution: `__name__ == "__main__"`
`dir()`	Effective namespace
`global()`	Dictionary of variable names of global scope to variable values
`locals()`	Dictionary of variable names of local scope to variable values
`eval()`	Return the value of the specified variable Example usage: `for x in dir(): print(x, eval(x))`

Python Libraries

The following commands are for setting up Python programs or libraries for use inside your program:

Command	Description
`import module1`	Import program / library `module1` as Python module
`from module1 import obj1, obj2`	Import the objects `obj1, obj2` from `module1`
`from module1 import *`	Import all objects in `module1`
`module1.__dict__`	Return the dictionary containing `module1`'s symbol table, like `dir()` of the main program

Examples of Python libraries containing other non-primitive data structures:

- array: Efficient arrays of numeric values
- collections: Abstract data structures
- dataclasses: For creating user-defined data structures
- datetime: Basic date and time types
- queue: Synchronized queue class

Experience Software Development

Much of the above doesn't make sense to Python newbies but are crucial when developing apps for commercial settings, including freelancing. At StationX, we have self-paced tutorials and labs on software development focusing on cybersecurity. Click here to learn more.

We hope this Python data structures cheat sheet is helpful. To download a PDF version, click here.