Python Data Structures Quick Reference

Eckel, THJSST AI1, Fall 2020

Some data structures in Python are **primitive data types**. These are passed by **value**. There are not many.

- integers and floats and Booleans are exactly what you expect, except integers automatically resize so overflows aren't a thing (don't you love Python?)
- there is also a surprisingly convenient "NoneType", which happens when you write `x = None`
  - all NoneType variables are identical pointers to a single memory location marked as "None"
  - this is useful to indicate errors, like "`if x is None:`" to catch an error state
- all primitive data types are immutable (see chart)

The remaining data structures are **complex data types.** These are passed by **reference**.

The next page contains a reference chart of salient features of all the Python complex data types. You may want to add additional notes as we discuss!

Sidenote: if you've forgotten what "pass by **value**" and "pass by **reference**" mean, consider this code (copy/paste somewhere and run if necessary):

```
def a(x):
    print("x is:", x)
    x = x + 1
    print("x is:", x)

y = 4
a(y)
print("y is:", y)


def b(x):
    print("x is:", x)
    x.append(4)
    print("x is:", x)

z = [2, 3]
b(z)
print("z is:", z)
```

Sidenote: "==" vs "is"

In Python, "==" always compares by *value*. The "is" operator checks to see if two objects are *literally the same object* (memory location). This can get confusing to experiment with because often Python sees literals in your code, makes them, and then points different references to the same literal to save memory. Like this:

```
s = "str"
t = "str"
print(s is t)
```

…is likely to print "True" because Python noticed you used the same string literal twice and so it just pointed both s and t to the same one. This won't happen with mutable structures, so to see the difference more clearly, try this:

```
v = [1, 2, 3]
w = [1, 2, 3]
print(v is w)
print(v == w)
```

Type	Iterable?	Mutable?	Can contain…?	Ordered?	Item Access?	Item Modification?	Insertion or add speed?	Removal speed?	Membership test speed?	Construct & use

List
(like Java ArrayList)
	YES	YES	Anything	YES	YES
					
					for example:
					`print(x[4])`
						YES
						
						for example:
						`x[4] = 5`
							O(n)	O(n)	O(n)
									
									for example:
									`if 3 in x:`
										`x = list()`
										`x = []`
										`x = [2, 3]`
										`x.append()`
										`x.insert()`
										`x.remove()`

Tuple
(immutable list)
	YES	NO*	Anything*	YES	YES	NO
						(not mutable)
							NO ADD	NO REMOVE	O(n)	`x = ()`
										`x = (2, 3)`
										`x = (2,)`

Stack
(just use a list)
	YES	YES	Anything	YES	YES
					(but why would you?)
						YES
						(but, uh, don't)
							Use .append()
							for O(1) add on right
								Use .pop()
								for O(1) remove on right
									O(n)	`x = list()`
										`x = []`
										`x.append()`
										`x.pop()`

Queue**	YES	YES	Anything	YES	YES
					(but why would you?)
						YES
						(but, uh, don't)
							Use .append()
							for O(1) add on right
								Use .popleft()
								for O(1) remove on left
									O(n)	`x = deque()`
										`x.append()`
										`x.popleft()`

String	YES	NO	Length 1 strings***	YES	YES	NO (not mutable)	NO ADD	NO REMOVE	O(n)	`x = ""` `x = 'str'`

Set
(like Java HashSet)
	YES	YES	Only immutable objects	NO
				(iteration order random)
					NO	NO	O(1)	O(1)	O(1)	`x = set()`
										`x = {2, 3}`
										`x.add()`
										`x.remove()`

Dictionary or Dict (like Java HashMap)	YES	YES	Only immutable keys; any values	YES****	Access any value by key `d["key"] =`	Modify any value by key (see left)	O(1) to add a new key	O(1) to remove a key	O(1) to check keys; O(n) to check values	*****
Heap (use a list; see Unit 0 page for details)	YES	YES	Anything	YES	YES (but why would you?)	YES (but, uh, don't)	O(log n) using heappush	O(log n) using heappop	O(n)	******

* Tuples are immutable if and only if they contain only immutable things. A tuple with lists, floats, integers, and strings is immutable. Put a list inside a tuple and it is no longer immutable. This matters (see lower in the chart) because a tuple like (3, 4) could be placed in a set or be a key in a dict, but a tuple like (3, [1, 2]) could not.

** You don't want to use a list here; lists only efficiently operate on the right. To either insert or pop off the left, efficiency is O(n). We'll use a deque class in the python library collections, so start your code with `from collections import deque`.

*** There is no character class in Python. When you iterate over a string, or access an element in a string, you get a length 1 string.

**** This is a fairly new and very unusual feature of Python dictionaries; when iterating over keys, iteration is in the order keys were added to the dict. By default, iterating over a dictionary iterates over keys. You can change that behavior using attributes. So, iterating over a dictionary might look like any of these:

- `for key in dict:`
- `for val in dict.values():`
- `for key, val in dict.items():`

***** Using a dictionary is too complex to fit in this box.

To construct a dictionary:

- `x = dict()`
- `x = {}`
- `x = {2: 3, "key": "value", True: "you can use a Bool as a key but um whyyy", 4: "stuff", "stuff": 2}`

To access a value in a dictionary by key:

- `x[2] = 4`
- `x["key"] = "new value"`
- `print(x[True])`

To add a new key, *just pretend like it already exists and access it*

- `x["new_key"] = "I just added a new key"`

****** See Unit 0 page for more details, but briefly:

- To use heaps, at the top of your code type "`from heapq import heappush, heappop, heapify`"
- Then, make a blank list, say for instance "`heap_list = []`"
- Each time an element is added to the list, use the command "`heappush(heap_list, new_item)`"
- To remove the minimum-valued element, use the command "`temp = heappop(heap_list)`"
- To simply look at the minimum-valued element, just look at `heap_list[0]`!
- You can use the command "`heapify(some_list)`", which will heapify a list that isn't a heap already in place