Contents

1 Data Structures

For any data type, one can see all built-in functions using `dir(list)`.

`list.insert()`	`del`	Iterate	`in`/Equality	Get `list[i]`	`len()`	Pop @ end `.pop(-1)`
O(n)	O(n)	O(n)	O(n)	O(1)	O(1)	O(1)

Table 1: List

Insert/Del are costliest at beginning of array, because they shift all of the remaining elements. Appending can be costly if preallocated memory becomes full (i.e. extending).

Tuples are the same but are immutable.

Use a list as a stack (FIFO): push/pop → `.append()`/`.pop()`. Use a list as a queue (LIFO): enqueue/dequeue → `.append()`/`.pop(0)`. `collections.deque()` is O(1) for `.append()`/`.pop()`/`.popleft()` compared to O(n) for native list.

Get `dict[key]`	Construct	`del`	Iterate
O(1)	O(1)	O(1)	O(n)

Table 2: Dict

An (unordered) dict is a hashmap. When iterating, don't need to call .keys(); just use `for key in dict: ...`

`a in set(A)`	`A-B`	`A \& A` (intersect $\cap$)	`A|B` (union $\cup$)
O(1)	O(`len(A)`)	O(`min( [len(A), len(B)] )`)	O(`n = len(A) + len(B)`)

Table 3: Set

Strings are immutable arrays of chars; changing a string creates a copy.

Bytearrays are mutable sequences of integers in $0 \leq x < 256$; each element representing an 8-bit byte in memory. `bytearray((1,1,3,5,8))`

1.0.1 G4G external cheatsheet

Geeks for Geeks cheatsheet

1.1 Datatypes & memory allocation

1.1.1 Checking size

Check easily with sys package: `sys.getsizeof(object)` (value is in bytes)

1.1.2 Schemes for `list` and `tuple`

For a an immutable tuple, every additional element (homogeneous) is allocated the same amount of space. With an `int`, that would be 8 bytes.

For a mutable list, additional appended elements will increase the allocated memory whenever the length of the list exceeds the number of elements $2^n + 1$. For example, more memory will be allocated at the $2^0 + 1, 2^2 + 1, 2^3 + 1$, etc. thresholds, corresponding to the 1, 5, 9, etc. locations.

Because a tuple has an immutable length, `len(tuple) <= len(list)`, with the difference greatest just after memory is allocated to the list.

2 Misc. Tips

2.1 `filter(function, list)`

Use filter to mask arrays.

```
# Given some list
numbers = range(1,1000)

# Have a filtering function
def isPrime(int) -> bool:
```

```
    return

# Filter will initially return an object to save memory
all_primes = filter(isPrime, list)

# Convert it to a list to print
list(all_primes)
```

3 Interpreter

3.1 `compile()`

Useful for *symbolic* programming, in which each line adds an element to a graph which is executed at the END of a set of instructions. In this way, variables do not hold values until they are executed. Python instructions are *imperative*, meaning that each line executes & variables updated at each instruction.

- String/bytestring i.e. open('script.py', 'r').read()
- Filename from which the code was read; if not read, doesn't matter
- Mode: 'exec', 'eval' or 'single'
  - eval – single expression i.e. `x=50; compile('x==50', '', 'eval')`
  - exec – block of a code that has Python statements, class and functions and so on i.e. file read
  - single – single interactive statement i.e. `x=50; compile('x', '', 'single')`

3.2 Global Interpreter Lock

Python variables which are referenced in multiple multithreaded processes are subject to being 'locked' by one thread or another, to prevent the variable being modified by both *at the same time*. The instructions must be bound for the CPU.

```
# For some CPU intensive function
def countdown(n):
    while n>0:
        n-=1
COUNT = int(5e5)

# Single thread
countdown(COUNT)

# Multithread
from threading import Thread
t1 = Thread(target=countdown, args = (COUNT/2))
```

```
t2 = Thread(target=countdown, args = (COUNT/2))
t1.start()
t2.start()
t1.join()
t2.join()
# Both will perform equally fast (use time.time())

# Use multiprocessing instead
# Each process will get its own interpreter and memory space
from multiprocessing import Pool
pool = Pool(processes = 2)
r1 = pool.apply_async(countdown, [COUNT//2])
r2 = pool.apply_async(countdown, [COUNT//2])
pool.close()
pool.join()
```