

Fundamentals

Review

- [Basic Python](#)
- [Data Types](#)
- [Loops](#)
- [Container Comprehension](#)
- [Recursion](#)
- [Data Structures](#)
- [Basics Cheatsheet](#)
- [Serialization Formats](#)
- [Style Guides](#)
- [Documentation Practices](#)
- [Pathlib](#)
- [Functions](#)

Basic Python

<https://neetcode.io/courses/lessons/python-for-coding-interviews>

Variables

```
# Variables are dynamicly typed
```

```
n = 0
```

```
print('n =', n)
```

```
>>> n = 0
```

```
n = "abc"
```

```
print('n =', n)
```

```
>>> n = abc
```

```
# Multiple assignments
```

```
n, m = 0, "abc"
```

```
n, m, z = 0.125, "abc", False
```

```
# Increment
```

```
n = n + 1 # good
```

```
n += 1    # good
```

```
n++      # bad
```

```
# None is null (absence of value)
```

```
n = 4
```

```
n = None
```

```
print("n =", n)
```

```
>>> n = None
```

If-statements

```
# If statements don't need parentheses
# or curly braces.

n = 1
if n > 2:
    n -= 1
elif n == 2:
    n *= 2
else:
    n += 2

# Parentheses needed for multi-line conditions.
# and = &&
# or  = ||
n, m = 1, 2
if ((n > 2 and
    n != m) or n == m):
    n += 1
```

Loops

```
n = 5
while n < 5:
    print(n)
    n += 1

# Looping from i = 0 to i = 4
for i in range(5):
    print(i)

# Looping from i = 2 to i = 5
for i in range(2, 6):
    print(i)

# Looping from i = 5 to i = 2
for i in range(5, 1, -1):
    print(i)
```

Math

```
# Division is decimal by default
```

```
print(5 / 2)
```

```
# Double slash rounds down
```

```
print(5 // 2)
```

```
# CAREFUL: most languages round towards 0 by default
```

```
# So negative numbers will round down
```

```
print(-3 // 2)
```

```
# A workaround for rounding towards zero
```

```
# is to use decimal division and then convert to int.
```

```
print(int(-3 / 2))
```

```
# Modding is similar to most languages
```

```
print(10 % 3)
```

```
# Except for negative values
```

```
print(-10 % 3)
```

```
# To be consistent with other languages modulo
```

```
import math
```

```
from multiprocessing import heap
```

```
print(math.fmod(-10, 3))
```

```
# More math helpers
```

```
print(math.floor(3 / 2))
```

```
print(math.ceil(3 / 2))
```

```
print(math.sqrt(2))
```

```
print(math.pow(2, 3))
```

```
# Max / Min Int
```

```
float("inf")
```

```
float("-inf")
```

```
# Python numbers are infinite so they never overflow
print(math.pow(2, 200))

# But still less than infinity
print(math.pow(2, 200) < float("inf"))
```

Arrays

```
# Arrays (called lists in python)
arr = [1, 2, 3]
print(arr)
#output: [1, 2, 3]

# Can be used as a stack. .pop takes off end of list, .insert adds to beginning
arr.append(4)
arr.append(5)
print(arr)
#output: [1, 2, 3, 4, 5]

num = arr.pop()
print(num)
#output: 5

print(arr)
#output: [1, 2, 3, 4]

arr.insert(1, 7)
print(arr)
#output: [1, 7, 2, 3, 4]

arr[0] = 0
arr[3] = 0
print(arr)

# Initialize arr of size n with default value of 1
```

```
n = 5
arr = [1] * n
print(arr)
print(len(arr))

# Careful: -1 is not out of bounds, it's the last value
arr = [1, 2, 3]
print(arr[-1])
#Output: 3

# Indexing -2 is the second to last value, etc.
print(arr[-2])
#Output: 2

# Sublists (aka slicing)
arr = [1, 2, 3, 4]
print(arr[1:3])
# output: [2, 3] be aware that index starts at 0 in array and last index is non-inclusive

# Similar to for-loop ranges, last index is non-inclusive
print(arr[0:4])

# But no out of bounds error. Easy to make mistake if no error, be aware
print(arr[0:10])
# output: [2, 3, 4] returns the final values as well if outside of index

# Unpacking
a, b, c = [1, 2, 3]
print(a, b, c)

# Be careful though
# a, b = [1, 2, 3]

# Loop through arrays
nums = [1, 2, 3]

# Using index
for i in range(len(nums)):
    print(f'index i:{i}, nums[i]:{nums[i]}')
```

```
#output:
#index i:0, nums[i]:1
#index i:1, nums[i]:2
#index i:2, nums[i]:3

# Without index
for n in nums:
    print(n)

# With index and value
for i, n in enumerate(nums):
    print(i, n)
#output:
#index i:0, nums[i]:1
#index i:1, nums[i]:2
#index i:2, nums[i]:3

# Loop through multiple arrays simultaneously with unpacking
nums1 = [1, 3, 5]
nums2 = [2, 4, 6]
for n1, n2 in zip(nums1, nums2):
    print(n1, n2)

# Reverse
nums = [1, 2, 3]
nums.reverse()
print(nums)

# Sorting
arr = [5, 4, 7, 3, 8]
arr.sort()
print(arr)

arr.sort(reverse=True)
print(arr)

arr = ["bob", "alice", "jane", "doe"]
arr.sort()
print(arr)
```

```
# Custom sort (by length of string)
arr.sort(key=lambda x: len(x))
print(arr)
```

```
# List comprehension
arr = [i for i in range(5)]
print(arr)
```

```
# 2-D lists
arr = [[0] * 4 for i in range(4)]
print(arr)
print(arr[0][0], arr[3][3])
```

```
# This won't work
# arr = [[0] * 4] * 4
```

Strings

```
# Strings are similar to arrays
s = "abc"
print(s[0:2])
```

```
# But they are immutable
# s[0] = "A"
```

```
# So this creates a new string
s += "def"
print(s)
```

```
# Valid numeric strings can be converted
print(int("123") + int("123"))
```

```
# And numbers can be converted to strings
print(str(123) + str(123))
```



```
# In rare cases you may need the ASCII value of a char
print(ord("a"))
print(ord("b"))

# Combine a list of strings (with an empty string delimiter)
strings = ["ab", "cd", "ef"]
print("".join(strings))
```

Queues

```
# Queues (double ended queue)
from collections import deque

queue = deque()
queue.append(1)
queue.append(2)
print(queue)

queue.popleft()
print(queue)

queue.appendleft(1)
print(queue)

queue.pop()
print(queue)
```

HashSets

```
# HashSet
mySet = set()

mySet.add(1)
mySet.add(2)
print(mySet)
print(len(mySet))
```

```
print(1 in mySet)
print(2 in mySet)
print(3 in mySet)

mySet.remove(2)
print(2 in mySet)

# list to set
print(set([1, 2, 3]))

# Set comprehension
mySet = { i for i in range(5) }
print(mySet)
```

HashMaps

```
# HashMap (aka dict)
myMap = {}
myMap["alice"] = 88
myMap["bob"] = 77
print(myMap)
print(len(myMap))

myMap["alice"] = 80
print(myMap["alice"])

print("alice" in myMap)
myMap.pop("alice")
print("alice" in myMap)

myMap = { "alice": 90, "bob": 70 }
print(myMap)

# Dict comprehension
myMap = { i: 2*i for i in range(3) }
print(myMap)
```

```
# Looping through maps
myMap = { "alice": 90, "bob": 70 }
for key in myMap:
    print(key, myMap[key])

for val in myMap.values():
    print(val)

for key, val in myMap.items():
    print(key, val)
```

Tuples

```
# Tuples are like arrays but immutable
tup = (1, 2, 3)
print(tup)
print(tup[0])
print(tup[-1])

# Can't modify
# tup[0] = 0

# Can be used as key for hash map/set
myMap = { (1,2): 3 }
print(myMap[(1,2)])

mySet = set()
mySet.add((1, 2))
print((1, 2) in mySet)

# Lists can't be keys
# myMap[[3, 4]] = 5
```

Heaps

```
import heapq

# under the hood are arrays
minHeap = []
heapq.heappush(minHeap, 3)
heapq.heappush(minHeap, 2)
heapq.heappush(minHeap, 4)

# Min is always at index 0
print(minHeap[0])

while len(minHeap):
    print(heapq.heappop(minHeap))

# No max heaps by default, work around is
# to use min heap and multiply by -1 when push & pop.
maxHeap = []
heapq.heappush(maxHeap, -3)
heapq.heappush(maxHeap, -2)
heapq.heappush(maxHeap, -4)

# Max is always at index 0
print(-1 * maxHeap[0])

while len(maxHeap):
    print(-1 * heapq.heappop(maxHeap))

# Build heap from initial values
arr = [2, 1, 8, 4, 5]
heapq.heapify(arr)
while arr:
    print(heapq.heappop(arr))
```

Functions

```
def myFunc(n, m):
    return n * m
```

```
print(myFunc(3, 4))

# Nested functions have access to outer variables
def outer(a, b):
    c = "c"

    def inner():
        return a + b + c
    return inner()

print(outer("a", "b"))

# Can modify objects but not reassign
# unless using nonlocal keyword
def double(arr, val):
    def helper():
        # Modifying array works
        for i, n in enumerate(arr):
            arr[i] *= 2

    # will only modify val in the helper scope
    # val *= 2

    # this will modify val outside helper scope
    nonlocal val
    val *= 2
    helper()
    print(arr, val)

nums = [1, 2]
val = 3
double(nums, val)
```

Classes

```
class MyClass:
    # Constructor
```

```
def __init__(self, nums):  
    # Create member variables  
    self.nums = nums  
    self.size = len(nums)  
  
    # self key word required as param  
def getLength(self):  
    return self.size  
  
def getDoubleLength(self):  
    return 2 * self.getLength()  
  
myObj = MyClass([1, 2, 3])  
print(myObj.getLength())  
print(myObj.getDoubleLength())
```

Data Types

Built-in Data Types

In programming, data type is an important concept.

Variables can store data of different types, and different types can do different things.

Python has the following data types built-in by default, in these categories:

Text Type:	<code>str</code>
Numeric Types:	<code>int</code> , <code>float</code> , <code>complex</code>
Sequence Types:	<code>list</code> , <code>tuple</code> , <code>range</code>
Mapping Type:	<code>dict</code>
Set Types:	<code>set</code> , <code>frozenset</code>
Boolean Type:	<code>bool</code>
Binary Types:	<code>bytes</code> , <code>bytearray</code> , <code>memoryview</code>
None Type:	<code>NoneType</code>

Type Hinting

```
greeting = "Hello, {}, you're {} years old"
```

```
def greet(user: str, age: int) -> str:  
    return greeting.format(user, age)
```

```
name = input("Your name?")  
age = int(input("How old are you?"))
```

```
print(greet(name, age))
```


Loops

In Python, there are several loop constructs commonly used for solving array problems. Here are the most common ones:

1. **For Loop:** The `for` loop is widely used for iterating over elements in an array or any iterable object.

It allows you to execute a block of code for each element in the array.

```
arr = [1, 2, 3, 4, 5]
for num in array:
    print(num)
```

2. **While Loop:**

The `while` loop is used when you need to execute a block of code repeatedly as long as a condition is true.

It's often used when you don't know in advance how many iterations are needed.

```
i = 0
while i < len(array):
    print(array[i])
    i += 1
```

3. **Nested Loops:** Nested loops involve placing one loop inside another loop. They are used when you need to iterate over elements of a multi-dimensional array or perform operations that require multiple levels of iteration.

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
for row in matrix:
    for num in row:
        print(num)
```

Typically used in brute force solutions where need to compare every permutation.

```
nums=[1,2,3,4,4]
for i in range(len(nums)): # Outer loop iterates over the rows
    print(f'loop {i}')
    for j in range(i+1, len(nums)): # Inner loop iterates over the columns
        print(nums[i],nums[j])
```

```
loop 0
```

```
1 2
```

```
1 3
```

```
1 4
```

```
1 4
```

```
loop 1
```

```
2 3
```

```
2 4
```

```
2 4
```

```
loop 2
```

```
3 4
```

```
3 4
```

```
loop 3
```

```
4 4
```

```
loop 4
```

4. **Enumerate:** The `enumerate()` function is used to iterate over both the elements and their indices in an array. It returns tuples containing the index and the corresponding element. For example:

```
for i, num in enumerate(arr):  
    print(f"Element at index {i} is {num}")
```

5. **Range:** In Python, the `range()` function is used to generate a sequence of numbers. It's commonly used in loops to iterate over a specific range of numbers.
- If called with one argument, it generates numbers starting from `0` up to (but not including) the specified number.
 - If called with two arguments, it generates numbers starting from the first argument up to (but not including) the second argument.
 - If called with three arguments, it generates numbers starting from the first argument up to (but not including) the second argument, with the specified step size (i.e., the difference between consecutive numbers).

1. Generating numbers from 0 to 5 (exclusive):

```
for i in range(6):  
    print(i)  
  
# Output: 0, 1, 2, 3, 4, 5
```

2. Generating numbers from 2 to 8 (exclusive):

```
for i in range(2, 9):  
    print(i)  
# Output: 2, 3, 4, 5, 6, 7, 8
```

3. Generating numbers from 1 to 10 (exclusive), with a step size of 2:

```
for i in range(1, 11, 2):  
    print(i)  
# Output: 1, 3, 5, 7, 9
```

4. Considering that the end of the range is excluded, when iterating entire array it needs to be -1

```
for i in range(len(array) - 1):  
    print(i)
```

It's important to note that `range()` returns a range object, which is a memory-efficient representation of the sequence of numbers. To actually see the numbers in the sequence, you typically use it within a loop or convert it to a list using `list(range(...))`.

Container Comprehension

List Comprehension

Dictionary Comprehension

Recursion

Data Structures

Array

1	3	3	7
0	1	2	3

```
arr = [1, 3, 3, 7]

for val in arr:
    # 1,3,3,7

7 in arr      # True, takes O(n)

arr[3]        # 7
arr[3] = 8
arr.pop()     # delete last element
arr.append(9) # insert last element

del arr[1]    # delete, takes O(n)
arr.insert(1, 9) # insert, takes O(n)
```

An Array is an ordered list of elements. Each element in an Array exists at an index 0, 1, 2, and so on.

Arrays are called "lists" in Python.

Lists can have duplicate values

Read/Write Time **$O(1)$** .
Insert/Delete Time **$O(n)$** .

Hashmap

1	3	3	7
"a"	"b"	"c"	"d"

HashMaps are Arrays, but where you can look up values by string, tuple, number, or other data, called a "key".

HashMaps are called "dictionaries" or "dicts" in Python.

We talk about how to implement a HashMap from scratch [here](#).

```
h = {} # empty hashmap
h = { # hashmap of the above image
    "a": 1,
    "b": 3,
    "c": 3,
    "d": 7
}

# look for a key, and if it doesn't exist, default to 0
h.get("a", 0) # 1
h.get("e", 0) # 0

for key in h:
    print(key) # "a" "b" "c" "d"

for k,v in h.items():
    print(k) # "a" "b" "c" "d"
    print(v) # 1 3 3 7

h["b"]      # read value (3)
h["b"] = 9  # write value (9)
del h["b"]  # delete value
```

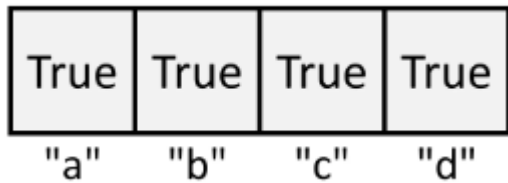
```
"a" in h # True, O(1)
```

Keys have to be unique

Read/Write Time **$O(1)$** .

Insert/Delete Time **$O(1)$** .

Set



Sets are essentially HashMaps where all the values are True. Python implements them as a new data structure for convenience.

Sets let you quickly determine whether an element is present. Here's the syntax for using a Set.

```
s = set() # empty set
s = {"a", "b", "c", "d"} # set of the above image

s.add("z")    # add to set
s.remove("c") # remove from set
s.pop()       # remove and return an element

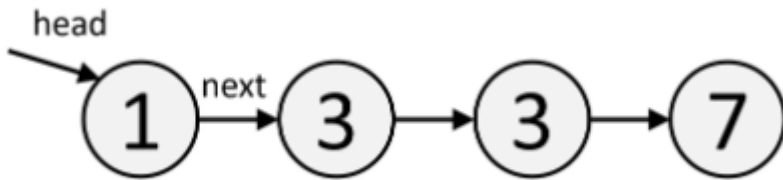
for val in s:
    print(val) # "a" "b" "d" "z"

"z" in s # True, O(1)
```

Read/Write Time **$O(1)$** .

Add/Remove Time **$O(1)$** .

Linked List

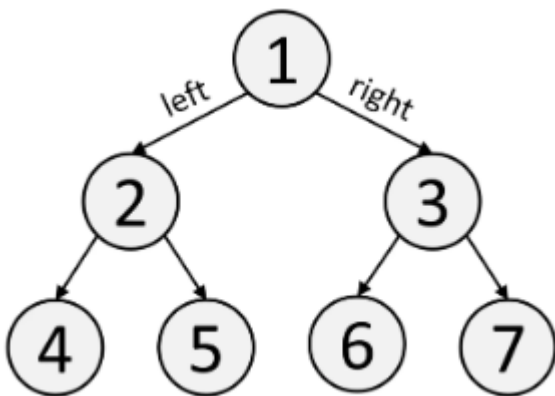


A Linked List is a custom data structure that you implement yourself. The first node in the Linked List is called the "head". Each node has a value `val`, and points to the next node `next`.

```
# create nodes, and change next pointers
head = ListNode(1)
b = ListNode(3)
c = ListNode(3)
d = ListNode(7)
head.next = b
b.next = c
c.next = d

# you can also create the list using the ListNode(val, next) function
head = (
    ListNode(1,
        ListNode(3,
            ListNode(3, ListNode(7))
        )
    )
)
```

Tree



Trees are a custom data structure you implement yourself. Usually, "Tree" means Binary Tree. In a Binary Tree, each node has a value `val`, a left child `left`, and a right child `right`.

```
# you can create the above tree by creating TreeNode objects
```

```
root = TreeNode(1)
```

```
l = TreeNode(2)
```

```
r = TreeNode(3)
```

```
ll = TreeNode(4)
```

```
lr = TreeNode(5)
```

```
rl = TreeNode(6)
```

```
rr = TreeNode(7)
```

```
root.left = l
```

```
root.right = r
```

```
l.left = ll
```

```
l.right = lr
```

```
r.left = rl
```

```
r.right = rr
```

```
# an easier way to write that code is to nest TreeNodes
```

```
root = TreeNode(1,
```

```
    TreeNode(2,
```

```
        TreeNode(4),
```

```
        TreeNode(5)
```

```
    ),
```

```
    TreeNode(3,
```

```
        TreeNode(6),
```

```
        TreeNode(7)
```

```
    )
```

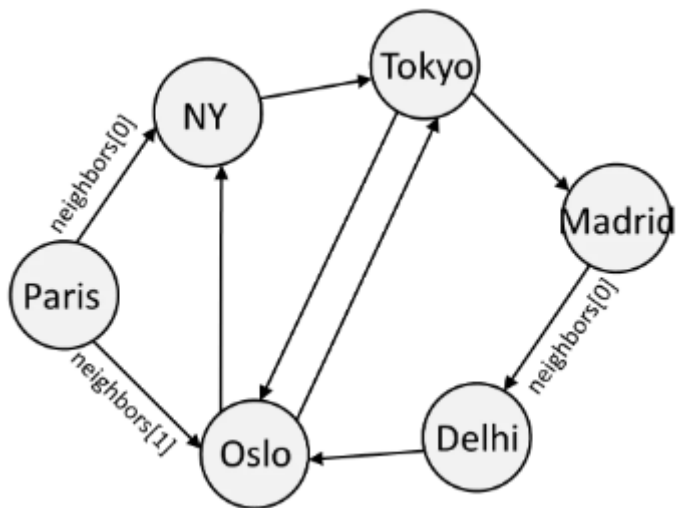
```
)
```

Trees generally have a height of $O(n)$, but balanced trees are spread wider and have a height of $O(\log n)$, which makes reading/writing faster.

Read/Write Time **$O(n)$** .

Read/Write Time **$O(\log n)$** for balanced trees like Binary Search Trees and Heaps.

Graph



A graph is made of nodes that point to each other.

There are two ways you can store a graph. It doesn't matter which one you pick.

```

# Option 1: create nodes, and point every node to its neighbors
Paris = Node()
NY  = Node()
Tokyo = Node()
Madrid= Node()
Delhi = Node()
Oslo = Node()
Paris.neighbors = [NY, Oslo]
NY.neighbors  = [Tokyo]
Tokyo.neighbors = [Madrid, Oslo]
Madrid.neighbors= [Delhi]
Delhi.neighbors = [Oslo]
Oslo.neighbors = [NY, Tokyo]

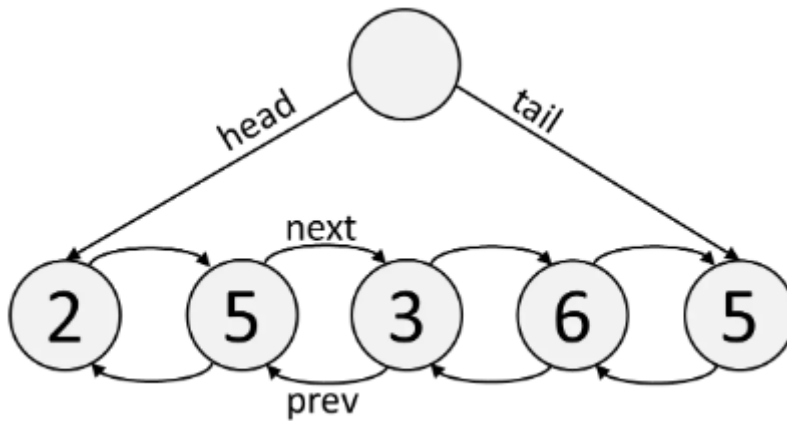
# Option 2: create an "Adjacency Matrix" that stores all the connections
adjacencyMatrix = {
  'Paris': ['NY', 'Oslo'],
  'NY':  ['Tokyo'],
  'Tokyo': ['Madrid', 'Oslo'],
  'Madrid': ['Delhi'],
  'Delhi': ['Oslo'],
  'Oslo': ['NY', 'Tokyo'],
}

```

Read/Write Time Varies.

Insert/Delete Time Varies.

DeQueue (Double-Ended)



A Double-Ended Queue (Deque) is a Linked List that has pointers going both ways, `next` and `prev`. It's a built-in data structure, and the syntax is similar to an Array.

The main benefit of Deques is that reading and writing to the ends of a Deque takes $O(1)$ time.

The tradeoff is that reading and writing in general takes $O(n)$ time.

```
from collections import deque
dq = deque([2, 5, 3, 6, 5])

# These are O(1)
dq.pop()      # remove last element
dq.popleft()  # remove first element
dq.append("a") # add last element
dq.appendleft("b") # add first element

#These are O(n)
dq[3]
dq[3] = 'c'

for val in dq:
    print(val) # 2 5 3 6 5

5 in dq # True, takes O(n)
```

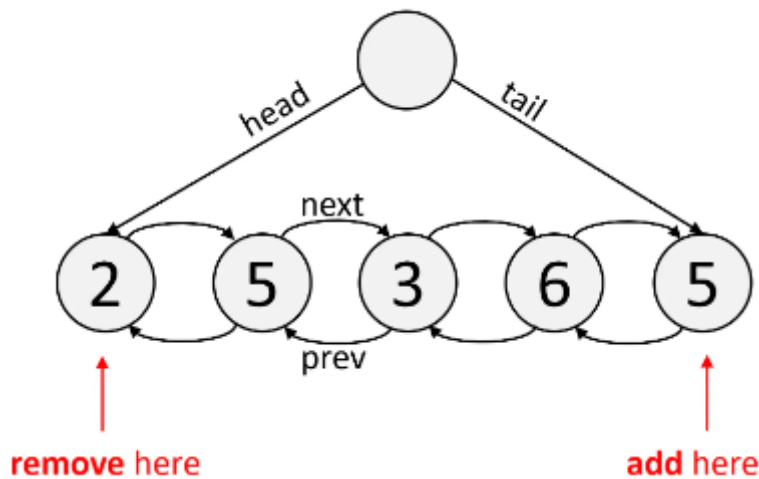
Read/Write Time **$O(n)$** .

Insert/Delete Time **$O(n)$** .

Append/Pop Time **$O(1)$** .

AppendLeft/PopLeft Time **$O(1)$** .

Queue



A Queue adds elements from one side, and remove elements from the opposite side. This means you remove elements in the order they're added, like a line or "queue" at grocery store. Typically you add to the right and pop from the left.

You can implement a Queue using a Deque, or implement it yourself as a custom data structure.

```
from collections import deque
q = deque([1, 3, 3, 7])

val = q.popleft() # remove 0th element
print(val)       # 1
print(q)         # [3, 3, 7]

q.append(5)      # add as last element
print(q)        # [3, 3, 7, 8, 5]
```

Read/Write Time **$O(1)$** .

Append/Pop Time **$O(1)$** .

Sorted Array

1	2	3	4	5	6	7
0	1	2	3	4	5	6

A Sorted Array is just an Array whose values are in increasing order.

Sorted Arrays are useful because you can use Binary Search, which takes $O(\log n)$ time to search for an element. Here are some built-in Python methods for sorting an Array.

```
# sorted() creates a new sorted array:
array = sorted([4,7,2,1,3,6,5])

# .sort() modifies the original array:
array2 = [4,7,2,1,3,6,5]
array2.sort()

print(array) # [1,2,3,4,5,6,7]
print(array2) # [1,2,3,4,5,6,7]
```

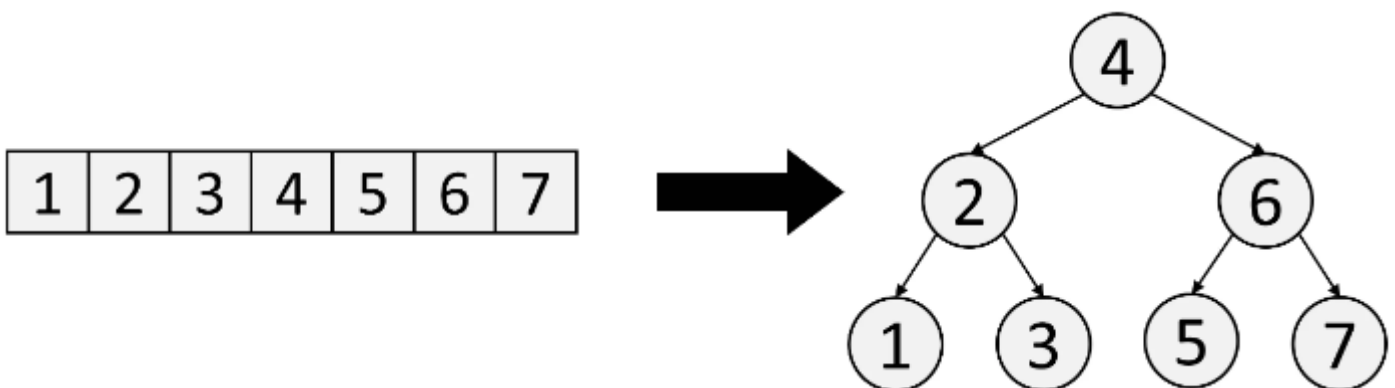
Python's bisect library contains Binary Search methods you can use, although often you'll have to implement Binary Search yourself in an interview.

Search Time **$O(\log n)$** .

Insert/Delete Time **$O(n)$** .

Sort Time **$O(n \log n)$** .

Binary Search Tree



A Binary Search Tree (BST) is just a Sorted Array but in tree form, so that inserting/deleting is faster than if you used a Sorted Array. It's a custom data structure - there is no built-in BST in

Python.

Stepping left gives you smaller values, and stepping right gives you larger values. A useful consequence of this is that an inorder traversal visits values in a BST in increasing order.

You won't need these in an interview, but here are some BST libraries

%pip install bintrees # (recommended)

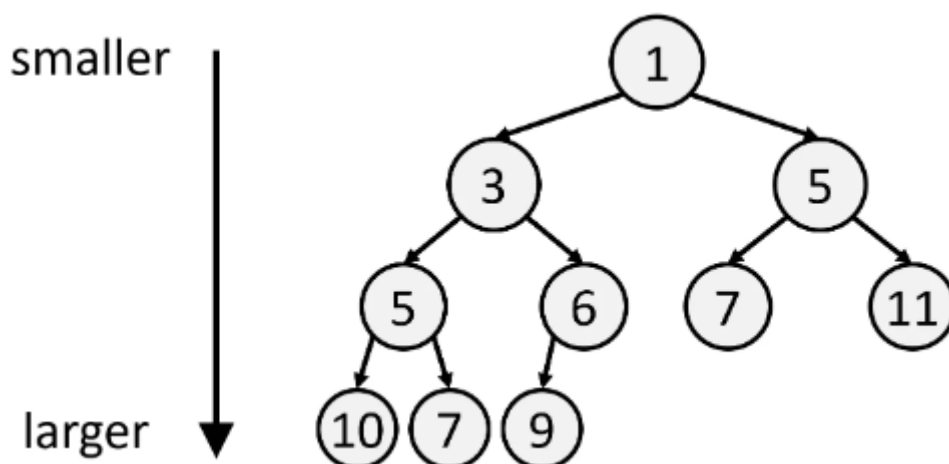
%pip install sortedcontainers

Search Time **$O(\log n)$** .

Insert/Delete Time **$O(\log n)$** .

Create Time **$O(n \log n)$** .

Heap



A Heap is a Tree, where the values increase as you step downwards.

The purpose of a Heap is to have immediate access to the smallest element.

The smallest element is always at the top. Heaps are always balanced, and they're stored as Arrays in the order of their BFS traversal. We introduced Heaps [here](#).

```
import heapq
h = [5,4,3,2,1]
heapq.heapify(h) # turns into heap, h=[1,2,3,5,4]
h[0]             # find min
```

```
heapq.heappop(h)    # h=[2,4,3,5]
heapq.heappush(h, 3) # h=[2,3,3,5,4]
h[0]                # find min
```

Python only allows Min-Heaps (smallest values on top). If you want a Max-Heap, you can implement it with a Min-Heap by negating all your values when you push and pop.

Read Time **$O(1)$** .

Push/Pop Time **$O(\log n)$** .

Create Time **$O(n)$** .

Basics Cheatsheet

Built-in

```
# primitives
x = 1          # integer
x = 1.2        # float
x = True       # bool
x = None       # null
x = float('inf') # infinity (float)

# objects
x = [1, 2, 3] # list (Array)
x = (1, 2, 3) # tuple (Immutable Array)
x = {1: "a"}  # dict (HashMap)
x = {1, 2, 3} # set
x = "abc123"  # str (Immutable String)
```

Casting

```
n = 7
str(n)  # '7'
int('7') # 7

x = [1,2,3]
set(x)  # {1,2,3}
tuple(x) # (1,2,3)

s = {1,2,3}
list(s) # [3,1,2] (sets don't store order)
```

Basic Operations

```
a = [1] # array with the number 1 in it
b = [1]

a == b # True, compares value
a is b # False, compares memory location (exact comparison)

1 / 2  # .5 (division)
1 % 2  # 1 (remainder)
1 // 2 # 0 (division, rounding down)
2 ** 5 # 32 (power, 2^5)
```

Falsy Values

In Python, you can use anything as a boolean. Things that "feel false" like None, 0, and empty data structures evaluate to False.

```
a = None
b = []
c = [15]

if a:
    print('This does not print')
if b:
    print('This does not print')
if c:
    print('This prints')
```

Loops with `range`

```
for i in range(4):  
    # 0 1 2 3  
  
for i in range(1, 4):  
    # 1 2 3  
  
for i in range(1, 6, 2): # loop in steps of 2  
    # 1 3 5  
  
for i in range(3, -1, -1): # loop backwards  
    # 3 2 1 0
```

Loops over Data Structures

```
arr = ["a", "b"]  
for x in arr:  
    # "a" "b"  
  
hmap = {"a": 4, "b": 5}  
for x in hmap:  
    # "a" "b"
```

List Features

```
x = [1,2] + [3,4] # [1,2,3,4]  
x = [0, 7]*2      # [0,7,0,7], don't use this syntax for 2D arrays  
  
x = [0,1,2,3]  
x[2:]  # [2,3]  
x[:2]  # [0,1]  
x[1:4] # [1,2,3]
```

```
x[3::-1] # [3,2,1,0]
x[-1]    # 3

y = reversed(x) # reversed array of x
x.reverse()     # reverses x in-place using no extra memory
sorted(x)       # sorted array of x
x.sort()        # sorts x in-place using no extra memory
```

List Comprehensions

Python has nice syntax for creating Arrays. Here's an example:

```
x = [2*n for n in range(4)]
#  [0, 2, 4, 6]

x = [a for a in [1,2,3,4,5,6] if a % 2 == 0]
#  [2, 4, 6]

# 2 x 3 matrix of zeros
x = [[0 for col in range(3)] for row in range(2)]
#  [[0, 0, 0],
#   [0, 0, 0]]
```

Here's the general form of list comprehension, and what it's actually doing:

```
x = [a*b  for a in A  for b in B  if a > 5]
# Is equivalent to:
x = []
for a in A:
    for b in B:
        if a > 5:
            x.append(a*b)
```

Generator Comprehensions

Generator Comprehensions are List Comprehensions, but they generate values lazily and can stop early. To do a generator comprehension, just use `()` instead of `[]`.

```
# stops early if finds True
any(SlowFn(i) for i in range(5))

# does not run the loop yet
computations = (SlowFn(i) for i in range(5))

# runs the loop - might stop early
for x in computations:
    if not x:
        break
```

Note that `()` can mean a tuple, or it can mean a generator. It just depends on context. You can think of Generator Comprehensions as being implemented exactly the same as List Comprehensions, but replacing the word `return` with `yield` (you don't have to know about this for an interview).

String Features

```
# you can use " or "", there's no difference
x = 'abcde'
y = "abcde"
x[2] # 'c'

for letter in x:
    # "a" "b" "c" "d" "e"

x = 'this,is,awesome'
y = x.split(',')
print(y) # ['this', 'is', 'awesome']

x = ['this', 'is', 'awesome']
y = '!'.join(x)
print(y) # 'this!is!awesome'
```

```
# convert between character and unicode number  
ord("a") # 97  
chr(97) # 'a'
```

Set Features

```
x = {"a", "b", "c"}  
y = {"c", "d", "e"}  
y = x | y # merge sets, creating a new set {"a", "b", "c", "d", "e"}
```

Functions

You declare a function using the `def` keyword:

```
def my_function():  
    # do things here  
  
my_function() # runs the code in the function
```

All variables you declare inside a function in Python are local. In most cases you need to use `nonlocal` if you want to set variables outside a function, like below with `x` and `y`

```
x = 1  
y = 1  
z = [1]  
  
def fn():  
    nonlocal y  
    y = 100    # global  
    x = 100    # local  
    z[0] = 100 # global (this would normally give an error. to avoid this, python refers to a more globally scoped variable)  
fn()
```

```
x # 1
y # 100
z[0] # 100
```

Anonymous Functions

You can also declare a function in-line, using the keyword `lambda`. This is just for convenience. These two statements are both the same function:

```
def fn(x,y):
    return x + y

lambda x,y: x + y
```

Boolean Operators

You can use the `any` function to check if any value is true, and the `all` to check if all values are true.

```
any([True, False, False]) # True
all([True, False, False]) # False

x = [1,2,3]
# checks if any value is equal to 3
any(True if val == 3 else False for val in x) # True
```

Ternary Operator

Most languages have a "Ternary operator" that gives you a value based on an if statement. Here's Python's:

```
0 if x == 5 else 1 # gives 0 if x is equal to 5, else gives 1
```

```
# Many other languages write this as (x == 5 ? 0 : 1)
```

Newlines

You can write something on multiple lines by escaping the newline, or just using parentheses.

```
x = 5 \  
    + 10 \  
    + 6
```

```
x = (  
    5  
    + 10  
    + 6  
)
```

Object Destructuring

You can assign multiple variables at the same time. This is especially useful for swapping variables. Here are a few examples:

```
# sets a=1, b=2
```

```
a,b = 1,2
```

```
# sets a=b, b=a, without needing a temporary variable
```

```
a,b = b,a
```

```
# sets a=1, b=2, c=3, d=4, e=[5, 6]
```

```
[a, b, [c, d], e] = [1, 2, [3, 4], [5, 6]]
```


Python Reference

Here's a reference to the official Python docs.

<https://docs.python.org/3/library/index.html>

The Built-in Functions and Built-in Types sections are the most useful parts to skim, although it's totally optional reading. The docs are not formatted in a very readable way.

Serialization Formats

JSON

```
import json

import requests

def main():
    data = {
        'username': 'james',
        'active': True,
        'subscribers': 10,
        'order_total': 39.99,
        'order_ids': ['ABC123', 'QQQ422', 'LOL300'],
    }

    print(data)

    # printing object as json string
    s = json.dumps(data)
    print(s)

    # getting python object from json string
    data2 = json.loads(s)
    assert data2 == data

    # writing data to file
    with open('test_data.json', 'w') as f:
        json.dump(data, f)

    # reading data from file
    with open('test_data.json') as f:
        data3 = json.load(f)
    assert data3 == data
```

```
r = requests.get('https://jsonplaceholder.typicode.com/users')  
print(type(r.json()))
```

```
if __name__ == '__main__':  
    main()
```

Style Guides

Pep8

Google Style Guide

<https://google.github.io/styleguide/pyguide.html>

Documentation Practices

Pathlib

```
from os import chdir

from pathlib import Path


def main() -> None:

    # current working directory and home directory
    cwd = Path.cwd()
    home = Path.home()
    print(f"Current working directory: {cwd}")
    print(f"Home directory: {home}")


    # creating paths
    path = Path("/usr/bin/python3")


    # using backslashes on Windows
    path = Path(r"C:\Windows\System32\cmd.exe")


    # using forward slash operator
    path = Path("/usr") / "bin" / "python3"


    # using joinpath
    path = Path("/usr").joinpath("bin", "python3")


    # reading a file from a path
    path = Path.cwd() / "settings.yaml"
    with path.open() as file:
        print(file.read())


    # reading a file from a path using read_text
    print(path.read_text())


    # resolving a path
    path = Path("settings.yaml")
    print(path)
    full_path = path.resolve()
```

```
print(full_path)

# path member variables
print(f"Path: {full_path}")
print(f"Parent: {full_path.parent}")
print(f"Grandparent: {full_path.parent.parent}")
print(f"Name: {full_path.name}")
print(f"Stem: {full_path.stem}")
print(f"Suffix: {full_path.suffix}")

# testing whether a path is a directory or a file
print(f"Is directory: {full_path.is_dir()}")
print(f"Is file: {full_path.is_file()}")

# testing whether a path exists
print(f"Full path exists: {full_path.exists()}")
wrong_path = Path("/usr/does/not/exist")
print(f"Wrong path exists: {wrong_path.exists()}")

# creating a file
new_file = Path.cwd() / "new_file.txt"
new_file.touch()

# writing to a file
new_file.write_text("Hello World!")

# deleting a file
new_file.unlink()

# creating a directory
new_dir = Path.cwd() / "new_dir"
new_dir.mkdir()

# changing to the new directory
chdir(new_dir)
print(f"Current working directory: {Path.cwd()}")

# deleting a directory
new_dir.rmdir()
```

```
if __name__ == "__main__":  
    main()
```


Functions

Positional and Keyword Arguments

Positional Arguments:

Positional arguments are the most common type of arguments in Python. They are passed to a function in the same order as they are defined in the function's parameter list.

```
def greet(name, age):  
    print(f"Hello, {name}! You are {age} years old.")  
  
# Calling the function with positional arguments  
greet("Alice", 30)
```

Keyword Arguments:

Keyword arguments are passed to a function with a specific keyword identifier. They do not rely on the order of parameters defined in the function.

```
def greet(name, age):  
    print(f"Hello, {name}! You are {age} years old.")  
  
# Calling the function with keyword arguments  
greet(age=25, name="Bob")
```

Default Values:

You can also provide default values for function parameters, which allows the function to be called with fewer arguments.

```
def greet(name="Anonymous", age=18):  
    print(f"Hello, {name}! You are {age} years old.")  
  
# Calling the function with default values  
greet()
```

Mixing Positional and Keyword Arguments:

You can mix positional and keyword arguments in a function call, but positional arguments must come before keyword arguments.

```
def greet(name, age):  
    print(f"Hello, {name}! You are {age} years old.")  
  
# Mixing positional and keyword arguments  
greet("Charlie", age=35)
```

Understanding these concepts will help you effectively pass arguments to functions in Python, providing flexibility and clarity in your code.

* args

In Python, when `*` is used as a prefix for a parameter in a function definition, it indicates that the parameter is a variable-length argument list, often referred to as "arbitrary positional arguments" or "varargs". This means that the function can accept any number of positional arguments, and they will be packed into a tuple.

```
def my_function(*args):  
    print(args)  
  
my_function(1, 2, 3, 4)
```

In this example, `*args` is used to collect all the positional arguments passed to `my_function()` into a tuple named `args`. When you call `my_function(1, 2, 3, 4)`, the output will be `(1, 2, 3, 4)`.

You can also combine `*args` with other regular parameters:

```
def my_function(a, b, *args):
    print("Regular parameters:", a, b)
    print("Extra positional arguments:", args)

my_function(1, 2, 3, 4, 5)
```

Here, `a` and `b` are regular parameters, while `*args` collects any additional positional arguments into a tuple.

This feature is particularly useful when you want to create functions that can handle a variable number of arguments, providing flexibility in your code.

Putting `*` as the first argument force manual typing of the argument when called

```
def place_order(*, item, price, quantity)
    print(f" {quantity} unitys of {item} at {price} price")

def what_could_go_wrong():
    place_order(item="SPX", price=4500, quantity=10000)
```