Conquer the

Coding Interview in Python

From Zero to Mastery

75 coding questions and answers to prepare you for coding interviews in weeks

Cathy Qian, Ph.D.

About the Author

Cathy Qian is currently a Senior Machine Learning Engineer at Pinterest, with expertise in developing large scale end-to-end machine learning solutions. She has previously worked at Twitter and Expedia, where she helped build and deploy machine learning applications in ads and search ranking systems. She has received a Ph.D. degree from the University of Pennsylvania and published 20+ research papers in high-impact peer-reviewed journals before transitioning to the high-tech industry. She enjoys writing and knowledge sharing in her spare time. Follow her on LinkedIn.

Table of Contents

Introduction **4**
Basic Python Syntaxes **5**
- List 5
- Set 6
- Heap 6
- Dictionary 7
- Collections 8
- Deque 9
- String 9
- Bitwise Operator 11
- Random Number Operation 11
- Misc 11
- Python Class 14
- Time Complexity of Basic Python Data Structures 15
  - List 16
  - Deque 17
  - Dictionary 17
  - Set 18
  - Tuples 18
  - String 19

Core Data Structures **20**
- Linear Data Structures 20
  - Array and String 20
    - Common problems and solutions 20
    - Code examples 20
  - Linked List 23
    - Common problems & solutions 24
    - Code examples 24
- Tree Data Structures 28
  - Binary Tree & Binary Search Tree 28
    - Common problems & Solutions 29
  - Heap 38
    - Implement a minHeap 38
    - Common problems & Solutions 41
  - Trie 43
    - Implement a Trie 44
    - Common problems and solutions 45
  - Graph 46

Implement a Graph 46
Common problems & Solutions 49
Core Algorithms 59
Divide and Conquer 59
Typical problems 59
Master Theorem 59
Related data structures 59
Sample Solutions 59
Binary Search 61
Typical problems 61
Related data structures 61
Sample Solutions 62
Breadth-First Search 68
Typical problems 68
Related data structures 68
Sample Solutions 68
Depth-First Search 74
Typical problems 74
Related data structures 74
Sample Solutions 75
Dynamic Programming 97
Typical problems 97
Related data structures 97
Sample Solutions 97
Union Find 104
Typical problems 104
Related data structure 104
Sample Solutions 104
Core Machine Learning Algorithms From Scratch 108
Linear Regression 108
Logistic Regression 109
K-means clustering 110
K-nearest neighbor 111
Decision tree 112
Reservoir sampling 115
Summary 116

Introduction

Coding interviews are an essential part of standard engineering interviews in the high-tech industry nowadays. However, preparing for coding interviews can be quite time-consuming for both junior and experienced engineers. It may take many junior engineers a couple of months of practice to get ready for interviews. And the more senior you are, the more you wish you could spend your precious time elsewhere.

The goal of this book is to help you conquer the coding interviews efficiently, hopefully in days or weeks instead of months, regardless of whether you are a junior or senior engineer. **All code snippets in this book are written in Python**. I try to cover as many details to help you understand the fundamentals while not too much to be overwhelming, as you may find in many other online tutorials.

The book includes the following sections:

- **Basic python syntaxes,** which summarizes python syntaxes commonly used in coding interviews including data structures such as list, set, heap, dictionary, collections, deque, string, bitwise operations and random number operations.
- **Core data structures**, which summarized common problems, solutions and code examples for
    - linear data structures: array, string, linkedlist
    - tree data structures: binary tree and binary search tree, heap, trie, graph
- **Core algorithms**, which includes
    - divide and conquer
    - binary search
    - breadth-first search
    - depth-first search
    - dynamic programming
    - union find
- **Machine learning algorithms**, which include coding up the following algorithms from scratch:
    - linear regression
    - logistic regression
    - k-nearest neighbor
    - k-means clustering
    - decision tree
    - sampling

Basic Python Syntaxes

This section summarizes basic python syntaxes commonly used in coding interviews.

List

A list is a collection of items stored in a single variable which is ordered and indexed. Note a list is unhashable, so it can not be a key of a dictionary.

```
l = [] # initialize an empty list
l.sort(key=lambda x: func(x), reverse=True/False) # sort in-place, return N.A.; if the key is not provided, it will sort based on the first item of the list items, then the second and third item if needed.
sorted(l, key=lambda x: func(x), reverse=True/False) # return sorted array, no change in original list; same for set, dictionary, tuple
l.reverse() # reverse elements in place, no return; l[::-1] return reversed list
l.pop() # pop out the last item of the list and return that item, equivalent to l.pop(-1)
l.pop(0) # pop out the first item of the list and return
l.extend(iterable) # append another list to the existing list, return N.A.
l.count(x) # return the number of x in the list
l.append(x) # add x to the end of the list in place, return N.A.
l.insert(i, x) # insert x at position i of the list in place, return N.A.
l.remove(x) # remove x from the list, if x does not exist, it throws ValueError
l.index(x) # return index of x in the list (only the first occurance); if x does not exist, it throws ValueError
l.index(x, start, end) # same as above; start and end are optional

l.copy() # return a shallow copy of the list, won't clone child objects
a = copy.deepcopy(l) # return a deep copy of the list, fully clone child objects
a, b = l[:], l[:] # shallow copy of list, not a, b = l, l (simply add pointer)

bisect.bisect_left(l, x, lo=0, hi=len(l)) # return left insertion index of x in l to maintain sorted order; parameter low and hi are optional; by default the entire list is used
bisect.bisect_right(l, x, lo=0, hi=len(l)) # return right insertion index of x in l to maintain sorted order
bisect.bisect(l, x, lo=0, hi=len(l)) # similar to bisect.bisect_left()
```

```
bisect.insort_left(l, x, lo=0, hi=len(l)) # return list with x left
inserted to maintain sorted order, search: O(log n), insertion: O(n), as
all items after the insertion points needs to be moved up
bisect.insort_right(l, x, lo=0, hi=len(l)) # return list with x right
inserted to maintain sorted order
bisect.insort(l, x, lo=0, hi=len(l)) # similar to bisect.insort_left()

zip(iterator1, iterator2, ...) # return an iterator with item pairs from
iterator1 and iterator2…
```

Set

A set is a collection of distinct items stored in a single variable which is unordered and unindexed. Note elements of a set need to be hashable, i.e., **you cannot have a set of lists because a list is unhashable.**

```
s = set() # initialize an empty set
s = {1,2,3,4} # initialize a set
s = set([1,2,3,4]) # initialize a set
s.add(x)  # add x to a set s
s.remove(x) # remove x from a set s
s.update(iterable) # add items from an iterable to an existing set s
s.pop() # pop out a random item
a & b # return intersection of two sets a and b
a.intersection(b) or b.intersection(a) # same as a & b
a.difference(b) # return a set of items that only exist in a but not in b
a^b # return a set of items in either a or b
a -= b # remove items in set b from set a
a.union(b) # return a set that contains all items from both sets a and b
a.isdisjoint(b) # return True if set a has no elements in common with b
a.issubset(b) # return True if a is subset of b
a <= b # test whether every element in a is in b or not
a < b # test whether every element in a is in b or not
```

Heap

Heaps are binary trees for which every parent node has a value less than or equal to any of its children. It's usually implemented using arrays for which heap[k] <= heap[2*k+1] and heap[k] <= heap[2*k+2] for all k, counting elements from zero. See **Core Data Structures** for its implementation in Python.
Time complexity of Python Heap implementation:

- push O(log n)
- pop O(log n)
- heapify O(n)
- getmin O(1)

```
import heapq
heapq.heapify(l) # change a list l to a min heap in place; items in the list need to be comparable to allow ranking, i.e., ListNode is not comparable, but ListNode.val is comparable; l[0] is now the top and minimum item in the min heap
heapq.heapify_max(l) # change a list l to a max heap in place; l[0] is now the maximum item in the max heap
heapq.heappush(h, x) # push a new item x into a heap h
heapq.heappop(h) # Pop and return the smallest item from the heap, maintaining the heap invariant. If the heap is empty, IndexError is raised.
heapq.nlargest(n, iterable, key=None) # Return a list with the n largest elements (sorted from largest to smallest) from the dataset defined by iterable (the iterable may or may not be sorted). key, if provided, specifies a function of one argument that is used to extract a comparison key from each element in iterable (for example, key=str.lower). Equivalent to: sorted(iterable, key=key, reverse=True)[:n].
heapq.nsmallest(n, iterable, key=None) # Return a list with the n smallest elements from the dataset defined by iterable. Time complexity is O(n*log(len(iterable)))since it maintains a heap of n elements and do heappush and heappop for all the rest elements.
```

Dictionary

A dictionary is a collection which is ordered, changeable and does not allow duplicates. Note dictionaries are *ordered* in Python 3.7 while unordered in Python 3.6 and earlier.

```
dic = dict() # initialize a dictionary
dic = {} # initialize a dictionary
len(dic)  # length of a dictionary
dic.update(ele) # add a new item to an existing dictionary
dic3 = dic1 | dic2 # merge dic1 and dic2 and return a new dic; dic1 and dic2 not changed after the merger
dic.pop(key) # remove an element from a python dictionary, return its value
dic.popitem() # removes the last inserted item from a python dictionary, return a (key, value) pair
del dic[key] # remove an element from a dictionary, no return
dic.clear() # empty the dictionary
```

```
import copy
copy_dic = dic.copy() # make a shallow copy of a dictionary
copy_dic = copy.deepcopy(dic) # make a deep copy of a dictionary
sorted(dic) # return a sorted list of keys
dic.get(key, value) # return the value of the item with the specified key,
if the key doesn't exist, return value (optional, default None)
for x in dic: # or for x in dic.keys()
    print(x) # print out all key names in the dictionary
for x in dic.values():
    print(x) # print out all values in the dictionary
for x, y in dic.items():
    print(x, y) # print key, values in the dictionary
dic.update(iterable or dict) # add element(s) in the iterable or another
dictionary to an existing dictionary if the key is not in the latter. If
the key is in the dictionary, it updates the key with the new value.

# Dictionary Comprehension
squares = {x: x*x for x in range(6)}
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Collections

In Python, the Collections module provides different types of containers that are used to store different objects like list, tuple, set and dictionary, and provide a way to access contained objects and iterate over them. Popular containers stored in the Collections module are

- Counter: dictionary subclass that keeps the counts of elements in an iterable in a dictionary where the key represents the element and value represents the count of the element in the iterable
- DefaultDict: dictionary subclass that provide some default values for the key that does not exist and never raises a KeyError
- OrderedDict: similar to DefaultDict but remembers the order in which keys were inserted
- Deque: list-like container with fast appends and pops on either end

```
import collections
collections.Counter(iterable).items()
d = collections.defaultdict(int) # initiate a dictionary of unknown length,
value type is int, default value is 0
d = collections.defaultdict(list) # initiate a dictionary of unknown
length, value type is list, default value is an empty list
d = collections.defaultdict(set) # initiate a dictionary of unknown length,
value type is set, default value is an empty set
d = collections.OrderedDict() # initiate a dictionary of unknown length and
```

```
keep its entries sorted as they are initially inserted

d = collections.deque() # initiate a deque
```

Deque

Deque is a double-ended queue which can add or remove elements from either end. It is part of the Collections module.

```
d = collections.deque() # initiate a deque

d.append(x) # insert element to the right end of the deque
d.appendleft(x) # insert element to the left end of the deque
d.extend(iterable) # add multiple values at the right end of the deque
d.extendleft(iterable) # add multiple values at the left end of the deque
d.insert(i, x) # insert x into the deque at position i

d.pop() # delete an element from the right end of the deque
d.popleft() # delete an element from the left end of the deque
d.remove(x) # remove the first occurrence of x from the deque, searching
from left to right
d.clear() # remove all elements in the deque

d.count(x)  # return the number of occurrences of x in the deque
d.reverse() # reverse elements in place, no return
d.index(x) # return the first index of x in the deque, searching from left
to right
d.copy() # return a shallow copy of the deque
```

String

Strings in python are surrounded by either single quotation marks, or double quotation marks.

```
s = ‘this is a great book’ # initiate a string s
B = sorted(s) # B is a list now, to return to str, use C =
''.join(sorted(s)); can't use s.sort() since string cannot be sorted
directly

s.isalpha() # return True if all characters in the string are alphabets and
there is at least one character, False otherwise
```