

# **📝 Data Structures and Algorithms Notes with Python**

By Eliana Lopez

This repository contains a deep dive on Data Structures and Algorithms theory and concepts along with Python based examples of many popular algorithms and data structures.

## **Table of Contents**

click to expand

## **Introduction and the Basics**

For this introduction I am going to define key terms so we can do into more depth about the applications and uses of data structures and algorithms.

These will be the key terms defined in this section:

* Data Types
* Abstract Data Types
* Algorithm
* Data Structure

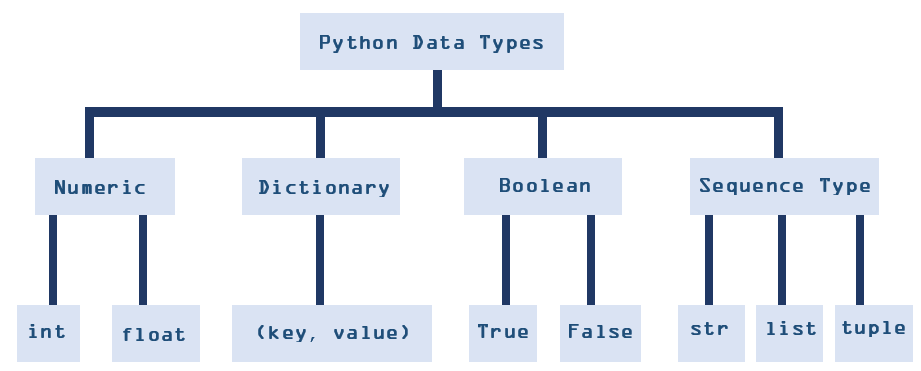
### **Data Types**

Data types are important because they illustrate the kind of value in the variable and tells us what operations can be performed on any particular data. Down below is an intuitive definition of what a data type is:

* An attribute of data that informs the interpreter on how to classify the variable.

In Python there are four categories of data types: *numeric, sequential, Booleans, and dictionary*

I have created a diagram of these categories of data types along with their respective data types down below.



### **What is an Abstract Data Types (ADT)?**

For me, I like to think of an abstract data type as a thought process of the rules and operations of data, there is no concrete implementation, but just thoughts and ideas on how this process might work.

* A logical process on how we view data and allowed operations without the regard on how it will be implemented.

Note: An ADT is the *what it does.*

### **What is an Algorithm?**

Overall this is the most intuitive definition of an algorithm:

* A set of instructions used to solved a problem

Outside of computing, one example of an algorithm can be a recipe from a cookbook to craft a meal. Where we follow a certain set of instruction from the cook book to create our meal, which would thus solve the problem of our hunger.

### **What is a Data Structure?**

The *implementation of an abstract data type* can be referred to a data structure but intuitively is a way of organizing data so it can be used effectively.

* A data structure is a format for accessing, storing, organizing, or structuring data.

Overall it is a technique of how data can inter-relate to each other logically or mathematically, or in layman's terms *how it does it.*

The Connection between Data Structures and Abstract Data Types:

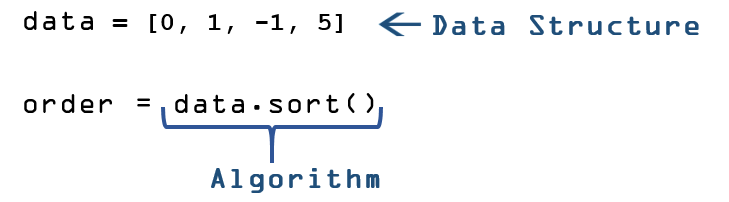
ADT gives us the blue print while a data structure tells us how to implement it

### **The Interconnection between Data Structures and Algorithms**

To put it simply, data structures need algorithms and algorithms need data structures.

The connection between algorithms and data structures is that an algorithm processes data and that data is then stored into a data structure.

This is an illustration of the interconnection of a data structure and an algorithm:



## **Time & Space Complexity**

Often, there is more than one way to solve the same problem with different programs. So how would you be able to compare the performance of different algorithms, *is one program better than the other?*

Well the answer to that question is quite simple, you have to see *which program is more efficient*.

There are two ways to determine which algorithm is more efficient:

* The amount of space or memory an algorithm requires
* The amount of time an algorithm requires to execute

Overall time and space complexity can be impacted from several factors such as hardware, operating system.

### **Time Complexity**

The time complexity is the amount of time an algorithm takes to complete its process, and this time is usually measured by the input n, in order to compare its efficiency with other algorithms.

When determining time complexity, you have to ask, *how does the runtime of our algorithm change as input becomes larger?* An alternative question you can ask is, *How much time, in a worse case scenario, is utilized?*

### **Three types of asymptotic notations**

Asymptotic is the nature of a graph as it reaches an untouchable bound, where we can understand how a function behaves in the asymptotic end.

There are three types of asymptotic notations utilized to illustrate the run time complexity:

Big-O - O(n)

* Worst Case
* Upperbound
* Slowest runtime

Big Theta - Θ(n)

* Average Case
* Average runtime
* Tight Bound of function complexity

Big Omega - Ω(n)

* Best Case
* Fastest runtime
* Lowerbound

### **Big-O**

Big-O notation is utilized to describe the performance or complexity of an algorithm, it does with by its tail behavior (see the graph below). Big-O is typically described as the worst case scenario because the more complex your algorithm, along with large amounts of data, the longer it will take to operate with time.

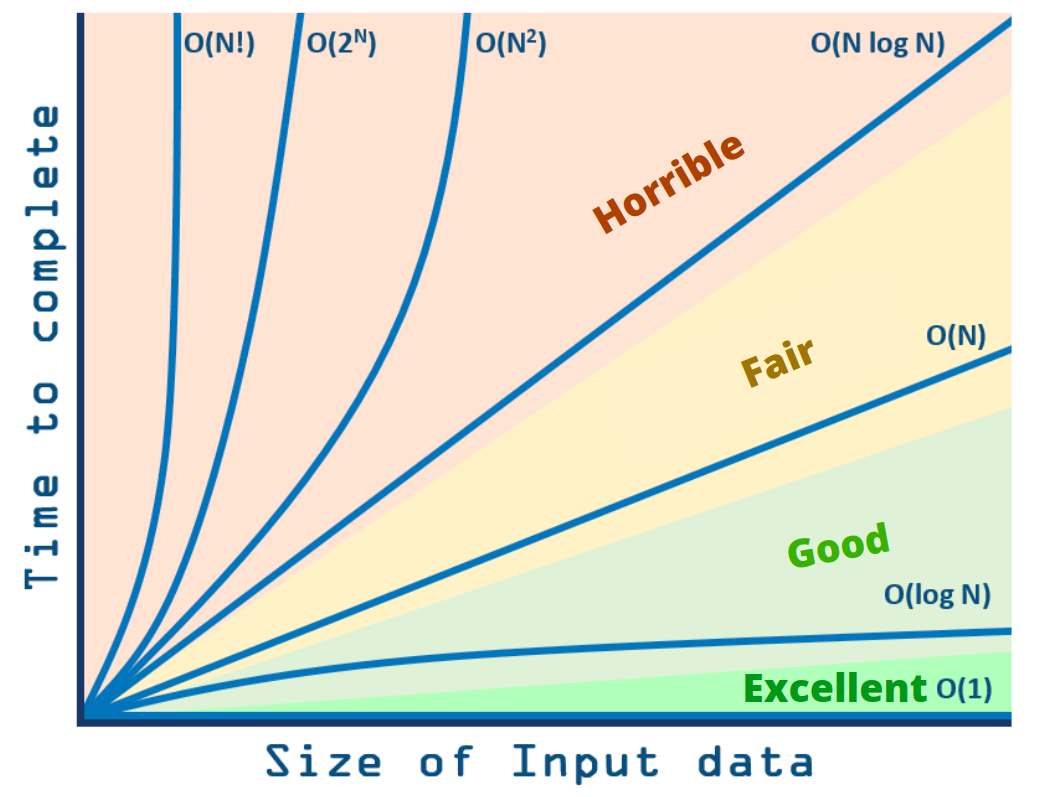
When thinking about Big-O, ask yourself this question, *how does an algorithm speed scale when the input scale become very large?*

What is the n of O(n)?

* The parameter 𝑛 is often referred to as *the size of the problem*

#### **Common Functions for Big-O**

* O(1) - *Constant Time*As an algorithm gets *very very* large, the run time of the algorithm would remain the same. If you notice from the graph below, the O(1) function remains constant as input gets bigger and bigger down the x-axis.
* O(log(n)) - *Logarithmic Time*The run time shapes into any typical log function, so as an algorithm gets *very very* large, this will lead to a small increase in log(n).
* O(n) - *Linear Time*The algorithm will take on the order of n operations to insert an item, so it is in proportion to the input size.
* O(nlog(n)) - *Logarithmic Linear Time*The algorithm is *doing log(n) work n times* which is where multiplication comes in place.
* O(n²) - *Quadratic Time*The algorithm is proportionally the squared number of inputs.
* O(2^n) - *Exponential Time*The algorithm is growths a constant amount of time, (in this case doubles) within each addition of input n.
* O(n!) - *Factorial Time*Any algorithm that calculates all permutation of a given array is O(N!).



Note for Big-O As seen in the graph above you can notice that Big-O gives us a tight-upperbound to functions.

Note for O(log(n)): Normally logarithmic are typically base 2. One thing to think about while looking at a log is, *what must I power my base (of 2) by to get n.* Example on how to solve this problem is down below.



### **Space Complexity**

When determining space complexity, you have to ask, *how does the space usage of our algorithm change as input becomes larger?* An alternative question you can ask is, *How much memory, in a worse case scenario, is needed?*

* Space complexity includes both *auxiliary space* (temporary space used during execution) and used space

Space Complexity = Auxiliary Space + Input space

Similar to time complexity, space complexity can also be expressed using Big-O notation, such as O(n), O(nlog(n)), O(n²).

*What space does your program create?*

* *does it create a single array of n-elements?*
* *does it create log(n) elements?*
* *ect...*

*Note: Runtime stack is a part of space complexity*

### **Optimizing Time or Optimizing Space?**

If we had to choose between either optimizing time or optimizing space.

Overall, it depends on your needs, but in a production setting, optimizing time is the main priority because we can buy memory, but we can't buy time!

This can lead to the best trade off of both increasing the space and lowering the time!

## **Basic Data Structures**

A data structure is linear if data items are arranged in a sequential order.

A linear data structure can be envision as having two ends, either a front/rear end type, a left/right end type, or a top/bottom end type.

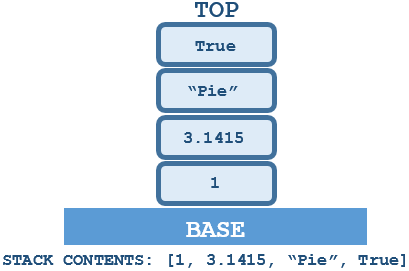
The following are examples of linear structures:

* Stack
* Queue
* Deque
* Linked-List

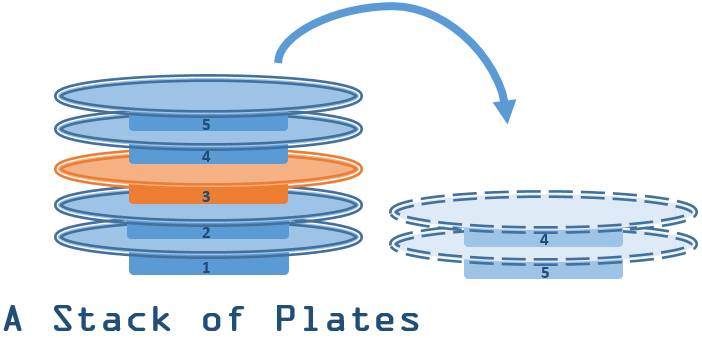
### **Stack**

A stack is a linear data structure with a Last In First Out (LIFO) removal procedure, so the addition and removal of items takes place within the same end.

The stack can be envision as having a top/bottom end type, where the first item that is inserted would be in the bottom, and the last item inserted will be at the top.



Think of a stack as collection of plates stacked upon each other. The only way to access the plate of your choice (i.e. The Orange One) is to remove the plates that are sitting on top of it. The diagram below illustrates the LIFO process that is utilized in order to access the Orange Plate.



#### **The Reversal Property of a Stack**

One fundamentally and useful application of the stack is that the orders of the items inserted in a stack are removed in the reverse order of the insertion.

Our plate diagram briefly illustrates this concept by removing the last two plates(plate 5, and plate 4) first.

#### **Stack ADT**

Stack as an ADT is the picture of how the stack works with data along with the operations the stack utilizes.

As stated earlier, *an ADT is the "what it does"* another way to view an ADT is a mathematical blue print.

These are he following operations of a stack:

* Stack() - creates a new empty stack.
* push(*item*) - adds a new item to the top of the stack.
* pop() - removes item from the top of the stack.
* peek() - returns the item from the top of the stack but does not remove it, thus it peaks at the stack.
* is\_empty() - returns a Boolean value (True or False), if whether the stack is empty or not.
* size() - returns the number of items in the stack.

To see Stack ADT implementation in Python via a custom Stack Class [click here for the source code](https://github.com/elianalopez/Data-Structures-and-Algorithms-Notes-with-Python/tree/main/BasicDataStructures/Stack).

#### **Stack Big-O Analysis - Time Complexity**

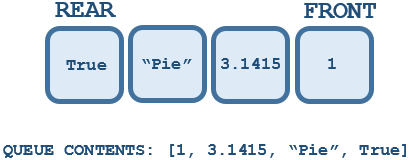
* push(*item*) - O(1)
* pop() - O(1)
* peek() - O(1)
* is\_empty() - O(1)
* size() - O(1)

NOTE: The space usage of the array based stack is O(n) where n is the amount of elements in the stack.

### **Queue**

Another type of linear data structure is called a Queue and it has a First In First Out (FIFO) removal procedure, so the items are remove chronologically, where the first items are removed.

The first image down below is a standard structure of a queue. From this image you can see that a queue has a front/rear end type, where the first item that is inserted will be located at the front, and the last item inserted will be at the rear.



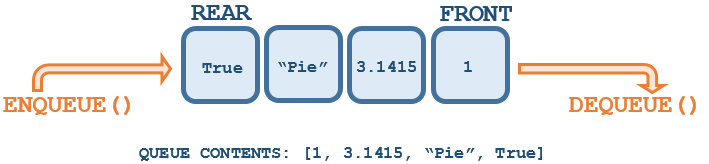
One way to think of a queue is like being in line in a supermarket, whoever is first in line would get check out first, and this process would continue in a sequential manner until after the last person on line checks out.

#### **Queue ADT**

These are the following operations of a queue:

* Queue() - creates a new empty queue.
* enqueue(*item*) - adds a new item to the rear of the queue.
* dequeue() - removes item from the front of the queue.
* is\_empty() - returns a Boolean value (True or False), if whether the queue is empty or not.
* size() - returns the number of items in the queue.

Down below is an illustration of how the enqueue and dequeue operation would function within a queue.



#### **Queue Big-O Analysis - Time Complexity**

* enqueue(*item*) - O(n)
* dequeue() - O(1)
* is\_empty() - O(1)
* size() - O(1)

NOTE: The space usage of the array based queue is O(n) where n is the amount of elements in the queue.

### **Deque**

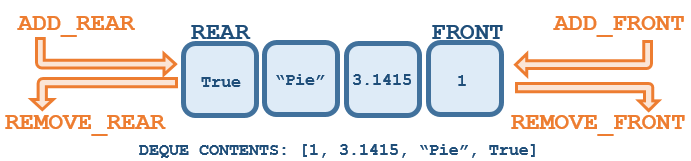
A deque also known as a double-ended queue is another type of linear collection of items. Similar to a queue, it also has a front/rear end type however what makes a deque different is how items can be added or removed from the front or the rear.

Removal is not LIFO or FIFO based but up to the programmer themselves to make consistent use of the addition and removal operations. Thus new items can be added or removed at either the front or the rear.

#### **Deque ADT**

These are the following operations of a deque:

* Deque() - creates a new empty deque.
* add\_front(*item*) - adds a new item to the front of the deque.
* remove\_front() - removes item from the front of the deque.
* add\_rear(*item*) - adds a new item to the rear of the deque.
* remove\_rear() - removes item from the front of the deque.
* is\_empty() - returns a Boolean value (True or False), if whether the deque is empty or not.
* size() - returns the number of items in the deque.



#### **Deque Big-O Analysis - Time Complexity**

* add\_front(*item*) - O(1)
* remove\_front() - O(1)
* add\_rear(*item*) - O(n)
* remove\_rear() - O(n)
* is\_empty() - O(1)
* size() - O(1)

### **Lists**

A list is a collection of items where each item holds a relative position with respect to others.

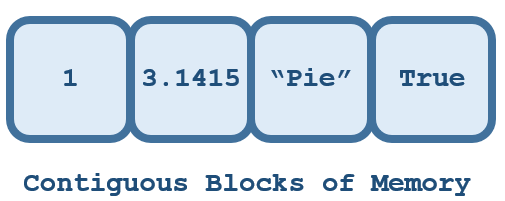
#### **Array-Based List**

Arrays are contiguous data structures, which means that they are neighboring blocks of memory.

In Python there is the built-in array based list which is written as comma-delimited values surrounded by square brackets, which be be something like this:

[1, 3.1415, "Pie", True]

Since arrays are contiguous blocks of memory, this is how the space of the array-based list from above would look like:

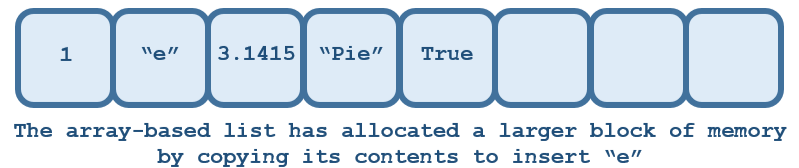


The array-based list in Python is dynamic, which means that the size can be modified by the programmer after its orignal declaration. Also in Python the array-based list is hetrogeneous, so items in the list do not need to be from the class.

Let's say we want to insert another element to this array, so we inserted *"e"* to *index 1* of the array, then our array-based list would look like this:

[1, "e", 3.1415, "Pie", True]

However our size would look like this:



This is because the elements of this dynamic array allocates a larger chunk of memory by copying the contents from its original array, as a result there is unused space which means there is more allocated memory than necessary.

An alternative data structure to utilize are linked-lists because of its two main benefits over the array-based list.

* There would be no pre-allocation of space
* Insertion is easier

### **Singly Linked-List**

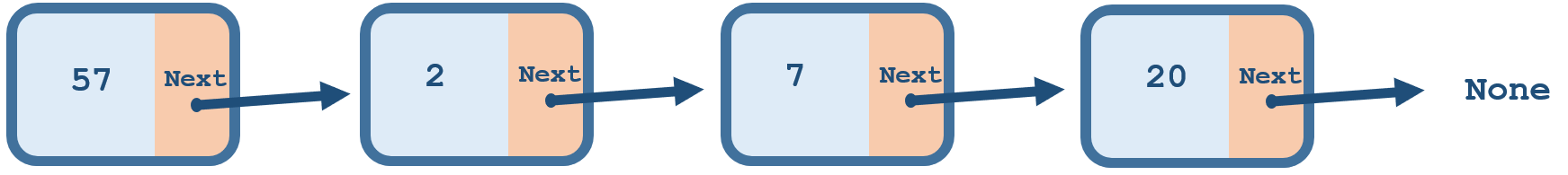
There are two types of linked-list we would go over:

* Unordered Linked-List
* Ordered Linked-List

### **Nodes**

#### **Unordered List**

The unordered list is a collection of items where new items are inserted at the end of the list regardless of order or value. The illustration below demonstrates what an unordered linked-list looks like:



#### **Unordered List ADT**

These are the following operations of the unordered linked list:

* List() - creates a new empty list.
* add(*item*) - adds a new item to the list.
* remove(*item*) - removes item from the list.
* search(*item*) - searchesthe item in the list.
* append(*item*) - adds a new item to the end of the list.
* index(*item*) - returns the postion of the item in the list.
* insert(*pos*, *item*) - adds a new item at the given index postion.
* pop() - removes item from the end of the list.
* pop(*pos*) - removes item from the given index postion.
* is\_empty() - returns a Boolean value (True or False), if whether the list is empty or not.
* size() - returns the number of items in the list.

#### **Ordered List**

The ordered list is somewhat similar to the unordered list, but what differentiates the two is the that the order list will always keep the items of the list in some sort of ordered, whether it be alphabetical or numerical.

For example if we wanted to create an ordered list of U.S. cities by ascending order and added them in this following order:

'Austin' << 'Chicago' << 'New York' << 'Boston'

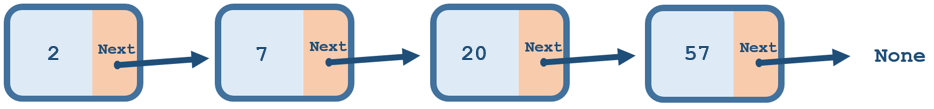
This would be the output of our list:

['Austin', 'Boston', 'Chicago', 'New York']

The same can go numericaly with the numbers 1 << 3 << 2.

If we wanted our list in ascending order we will get [1, 2 , 3] or if we wanted our list in descending order then we will get [3, 2, 1].

Down below is an illustration of our linked-list in ascending order:



#### **Ordered List ADT**

These are the following operations of the ordered linked list:

* OrderedList() - creates a new empty list.
* add(*item*) - adds a new item to the list by following the order.
* remove(*item*) - removes item from the list.
* search(*item*) - searchesthe item in the list.
* index(*item*) - returns the postion of the item in the list.
* insert(*pos*, *item*) - adds a new item at the given index postion.
* pop() - removes item from the end of the list.
* pop(*pos*) - removes item from the given index postion.
* is\_empty() - returns a Boolean value (True or False), if whether the list is empty or not.
* size() - returns the number of items in the list.

## **Recursion**

A recursive function is a function that calls itself.

There are three principals of a recursive function:

* Know when to stop
* Decide how to take step 1
* Break down the task into smaller tasks

## **Searching and Sorting**

## **Favorite Resources**

For Overall Data Structures Information with Python  
Miller, B. N., & Ranum, D. L. (2014). *Problem solving with algorithms and data structures using Python*. Decorah, IA.  
 [Click this link to access the online book](http://www.openbookproject.net/books/pythonds/index.html)

For Python Operation Complexity  
Pattis, R. E. (n.d.). *Complexity of Python Operations*. Retrieved November 30, 2020, from  
 <https://www.ics.uci.edu/~pattis/ICS-33/lectures/complexitypython.txt>