mutable and immutable data types in python
you have an immutable object if you can't change the object's state after you've created it. In contrast, a mutable object allows you to modify its internal state after creation. you're able to change an object's state or contained data is what defines if that object is mutable or immutable.

Immutable objects are common in functional programming, while mutable objects are widely used in object-oriented programming. Because Python is a multiparadigm programming language, it provides mutable and immutable objects for you to choose from when solving a problem.

Mutable Built-in Data Types in Python

Mutable data types are another face of the built-in types in Python. The language provides a few useful mutable collection types that you can use in many situations. These types allow you to change the value of specific items without affecting the identity of the container object.

Lists

Python lists are a classic example of a mutable data type.

Mutation	Description	Syntax
Item assignment	Replaces the data item stored at a given `index` with a new data item, `new_value`	`a_list[index] = new_value`
Slice assignment	Replaces the data items within a given `slice` of the list	`a_list[start:stop:step] = new_values`
Item deletion	Deletes the data item at a given `index`	`del a_list[index]`
Slice deletion	Deletes the data items within a `slice` of the list	`del a_list[start:stop:step]`

Method	Description
`a_list.append(item)`	Appends `item` to the end of `a_list`.
`a_list.clear()`	Removes all items from `a_list`.
`a_list.extend(iterable)`	Extends `a_list` with the contents of `iterable`.
`a_list.insert(index, item)`	Inserts `item` into `a_list` at `index`.
`a_list.pop(index)`	Returns and removes the item at `index`. With no argument, it returns the last item.
`a_list.remove(item)`	Removes the first occurrence of `item` from `a_list`.
`a_list.reverse()`	Reverses the items of `a_list` in place.
`a_list.sort(key=None, reverse=False)`	Sorts the items of `a_list` in place.

https://realpython.com/python-mutable-vs-immutable-types/

Bubble Sort

Bubble Sort is a simple sorting algorithm. This sorting algorithm repeatedly compares two adjacent elements and swaps them if they are in the wrong order. It is also known as the sinking sort. It has a time complexity of $O(n^2)$ in the average and worst cases scenarios and O(n) in the best-case scenario.

bubble sort

```
def bubbleSort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
arr = [ 2, 1, 10, 23 ]
bubbleSort(arr)
print("Sorted array is:")
for i in range(len(arr)):
    print("%d" % arr[i])
```

Insertion Sort

This sorting algorithm maintains a sub-array that is always sorted. Values from the unsorted part of the array are placed at the correct position in the sorted part. It is more efficient in practice than other algorithms such as selection sort or bubble sort. Insertion Sort has a Time-Complexity of $O(n^2)$ in the average and worst case, and O(n) in the best case

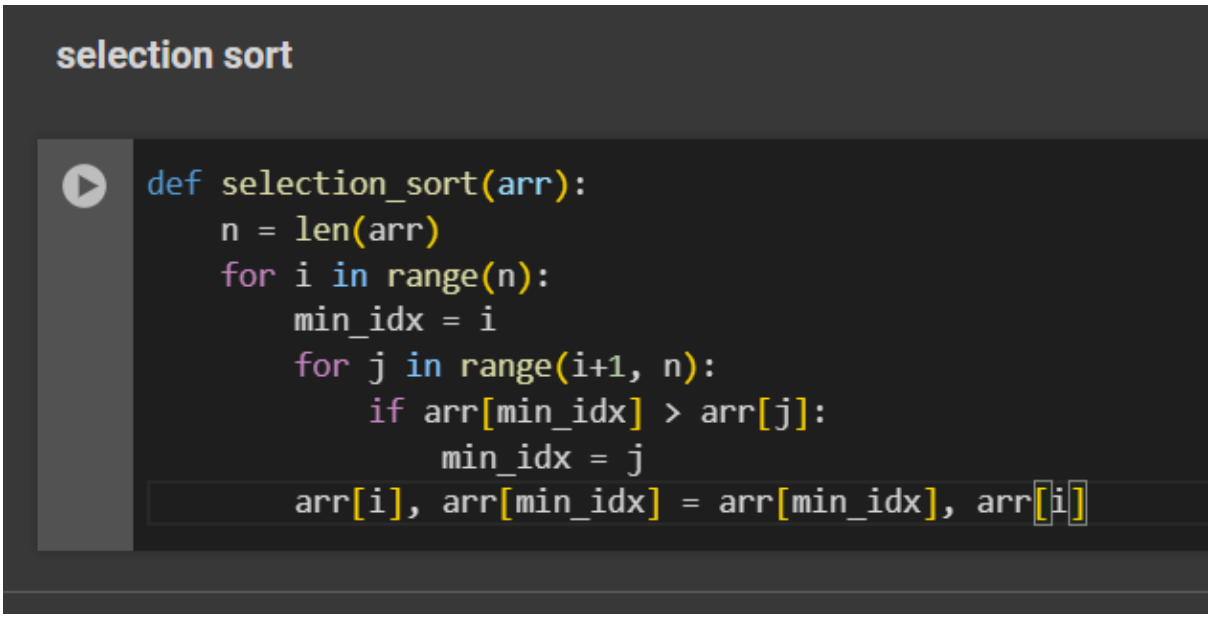
insertion sort

```
def insertion_sort(list1 ):
        for i in range(1, len(list1)):
            a = list1[i]
            j = i - 1
            while j >= 0 and a < list1[j]:
                list1[j + 1] = list1[j]
                j -= 1
            list1[j + 1] = a
        return list1
list1 = [ 7, 2, 1, 6 ]
print("The unsorted list is:", list1)
print("The sorted new list is:", insertion_sort(list1))
```

https://www.geeksforgeeks.org/sorting-algorithms-in-python/

Selection Sort

Selection Sort works by repeatedly selecting the smallest element from the unsorted part of the array and putting it at the beginning of the array.



Merge Sort

Merge Sort is a divide-and-conquer algorithm that works by dividing the array into two halves, sorting each half, and then merging the two halves.

merge sort

```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr)//2
        L = arr[:mid]
        R = arr[mid:]

        merge_sort(L)
        merge_sort(R)

        i = j = k = 0

        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
            arr[k] = R[j]
           j += 1
            k += 1
```

https://medium.com/@antrixsh/sorting-algorithms-in-python-implementation-and-analysis-d3caaf3d0283