Member-only story

How to Implement a Linked List in Python

Exploring how to write Linked List and Node objects from scratch using Python

Giorgos Myrianthous · Follow

Published in Towards Data Science

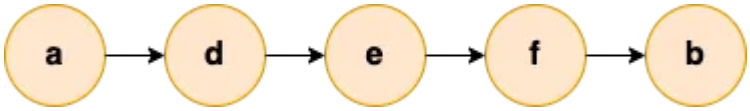
6 min read · Dec 20, 2021

Photo by Mael BALLAND on Unsplash

Introduction

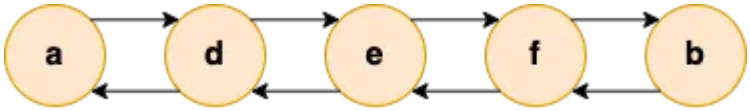
Linked Lists are among the most fundamental data structure that **represents a sequence of nodes**. The first element of the sequence is called the **head** of the Linked List while the last element corresponds to the **tail.**

Every node in the sequence has a pointer to the next element and optionally a pointer to the previous element. In **Singly Linked Lists** each node points to only the next node.



A singly linked list — Source: Author

On the other hand, in **Doubly Linked Lists** each node points to the next as well as to the previous node.



A doubly linked list — Source: Author

Linked Lists are extremely useful in various scenarios. They are typically preferred over standard arrays when

- you need a constant time when adding or removing elements from the sequence
- manage memory more efficiently especially when the number of elements is unknown (if arrays you may have to constantly shrink or grow them. Note though that filled arrays usually take up less memory than Linked Lists.
- you want to insert items in the middle point more efficiently

Unlike other general purpose languages, Python does not have a built-in implementation of Linked Lists in its standard library. In today's article we will explore how to **implement a user-defined Linked List class using Python.**

Implementing User Defined Linked Link Class in Python

First, let's create a user-defined class for individual **nodes** in a Linked List. This class will be suitable for both Singly or Doubly Linked Lists. Therefore, the instances of this class should be capable for storing the value of the node, the next as well as the previous node.

```
class Node:
    def __init__(self, value, next_node=None, prev_node=None):
        self.value = value
        self.next = next_node
        self.prev = prev_node

    def __str__(self):
        return str(self.value)
```

Note that when an instance of a `Node` has `next` set to `None` then it means that it is essentially the tail of the Linked List (either Singly or Doubly). Similarly, in Doubly Linked Lists, when a node has `prev` set to `None` then this indicates that the node is the head of the Linked List.

Now that we have created a class for the Nodes, we can now create the class for the Linked List itself. As mentioned already, a Linked List has a `head`, a `tail` and the nodes pointing to each other.

```
class LinkedList:
    def __init__(self, values=None):
        self.head = None
        self.tail = None

        if values is not None:
            self.add_multiple_nodes(values)
```

Now in order to add the values provided in the constructor as nodes in the Linked List we need to define a couple of additional methods.

The first method called **`add_node`** is used to add a single node to the Linked List.

```
def add_node(self, value):
    if self.head is None:
        self.tail = self.head = Node(value)
    else:
        self.tail.next = Node(value)
        self.tail = self.tail.next
    return self.tail
```

Now let's go through the logic of the method quickly. If the Linked List has no head, then it means it's empty and therefore the node to be added will be both the head and the tail of the Linked List. If the head is not empty, then we add the newly created `Node` as the `next` element of the current `tail` and finally move the tail to point to the newly created `Node`.

The second method is called **`add_multiple_nodes`** which is called in the constructor and in simply calls the `add_node` method we defined earlier in order to add multiple values as nodes in the Linked List instance.

```
def add_multiple_nodes(self, values):
    for value in values:
        self.add_node(value)
```

So far, our Linked List class looks like below:

```
class LinkedList:
    def __init__(self, values=None):
        self.head = None
        self.tail = None

        if values is not None:
            self.add_multiple_nodes(values)

    def add_node(self, value):
        if self.head is None:
            self.tail = self.head = Node(value)
        else:
            self.tail.next = Node(value)
            self.tail = self.tail.next
        return self.tail
```

```
    def add_multiple_nodes(self, values):
        for value in values:
            self.add_node(value)
```

Now let's create an additional method that'd be able to insert a new element but this time in the beginning of the Linked List, i.e. as a head.

```
def add_node_as_head(self, value):
    if self.head is None:
        self.tail = self.head = Node(value)
    else:
        self.head = Node(value, self.head)
    return self.head
```

Now let's override some special methods in our class that could potentially be useful. Firstly, let's implement `__str__` method so that the string representation of a Linked List object is human readable. For example, when printing out a Linked List with nodes `a, b, c, d` the output will be `a -> b -> c -> d`.

```
def __str__(self):
    return ' -> '.join([str(node) for node in self])
```

Secondly, let's also implement `__len__` method that will return the length of our user-defined class, which is essentially the number of nodes included in the sequence. All we need to do is iterate over every node of the sequence until we reach the tail of the Linked List.

```
def __len__(self):
    count = 0
    node = self.head
    while node:
        count += 1
        node = node.next
    return count
```

Finally, let's ensure that `LinkedList` class is iterable by implementing `__iter__` method.

```
def __iter__(self):
    current = self.head
    while current:
        yield current
        current = current.next
```

Additionally, we could also create a property called `values` so that we can access the values of all nodes included in the sequence.

```
@property
def values(self):
    return [node.value for node in self]
```

The final class looks like below:

```
class LinkedList:
    def __init__(self, values=None):
        self.head = None
        self.tail = None

        if values is not None:
            self.add_multiple_nodes(values)

    def __str__(self):
        return ' -> '.join([str(node) for node in self])

    def __len__(self):
        count = 0
        node = self.head
        while node:
            count += 1
            node = node.next
        return count

    def __iter__(self):
        current = self.head
        while current:
            yield current
            current = current.next
```

```
    @property
    def values(self):
        return [node.value for node in self]

    def add_node(self, value):
        if self.head is None:
            self.tail = self.head = Node(value)
        else:
            self.tail.next = Node(value)
            self.tail = self.tail.next
        return self.tail

    def add_multiple_nodes(self, values):
        for value in values:
            self.add_node(value)

    def add_node_as_head(self, value):
        if self.head is None:
            self.tail = self.head = Node(value)
        else:
            self.head = Node(value, self.head)
        return self.head
```

Now even our `Node` class can represent nodes included in either of Singly or Doubly Linked Lists, the `LinkedList` class we defined can only support Singly Linked Lists. This is because when adding nodes, we are not specifying the previous node.

To deal with Doubly Linked Lists, we can simply create an additional class that inherits from `LinkedList` class and overrides the `add_node` and `add_node_as_head` methods:

```
class DoublyLinkedList(LinkedList):
    def add_node(self, value):
        if self.head is None:
            self.tail = self.head = Node(value)
        else:
            self.tail.next = Node(value, None, self.tail)
            self.tail = self.tail.next
        return self

    def add_node_as_head(self, value):
        if self.head is None:
            self.tail = self.head = Node(value)
        else:
            current_head = self.head
            self.head = Node(value, current_head)
```

```
            current_head.prev = self.head
        return self.head
```

Full Code for User-Defined Linked Lists in Python

The full code containing the three classes we created as part of today's tutorial is given below as a GitHub Gist.

The full code containing Node, LinkedList and DoublyLinkedList Python classes — Source: Author

Final Thoughts

In today's guide we discussed about one of the most fundamental data structures, namely Linked Lists. Given that Python's standard library does not contain any implementation of this specific data structure, we explored how one can implement a user-defined Linked List class from scratch.

You may also like

LeetCode Problem 2: Add Two Numbers Solution in Python

Understanding how to efficiently work out Add Two Numbers problem with Linked Lists in Python

towardsdatascience.com

Augmented Assignments in Python

Understanding how augmented assignment expressions work in Python and why you should be careful when using them with...

towardsdatascience.com

Python | Programming | Data Science | Software Development | Algorithms

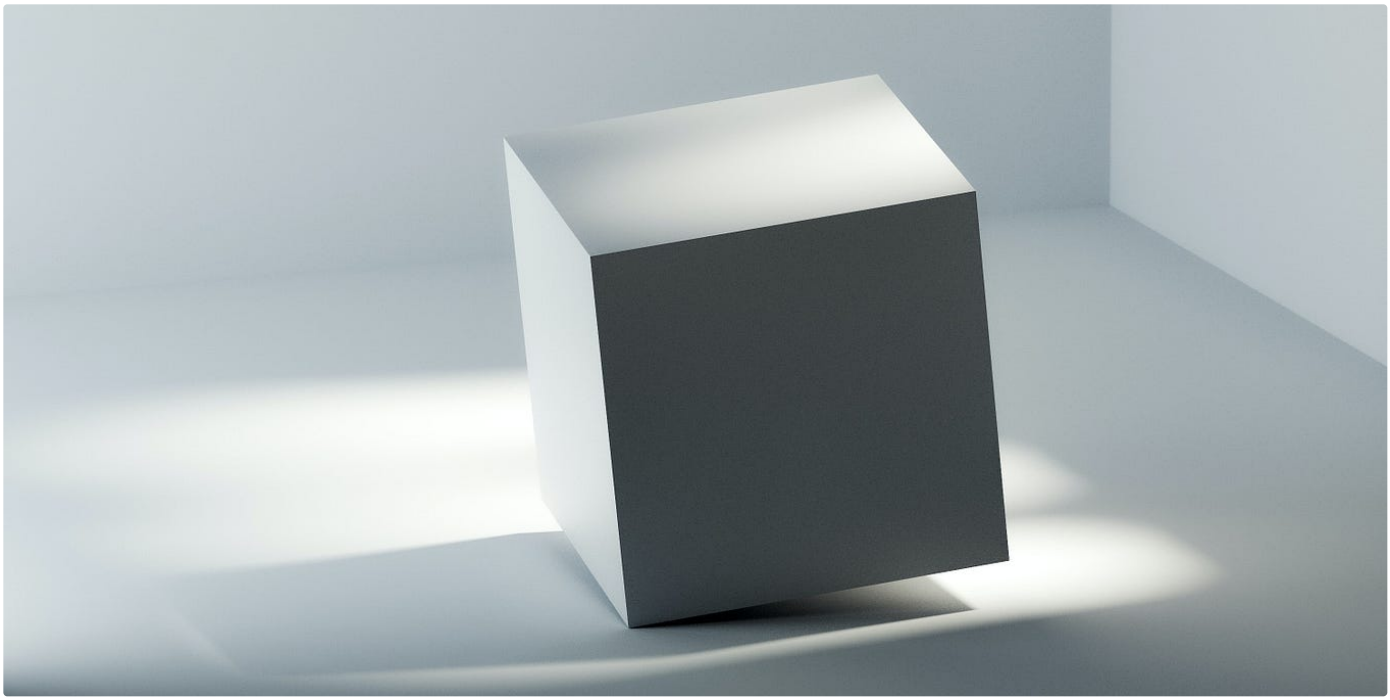
Follow

Written by Giorgos Myrianthous

7.5K Followers · Writer for Towards Data Science

I strive to build data-intensive systems that are not only functional, but also scalable, cost effective and maintainable over the long term.

More from Giorgos Myrianthous and Towards Data Science

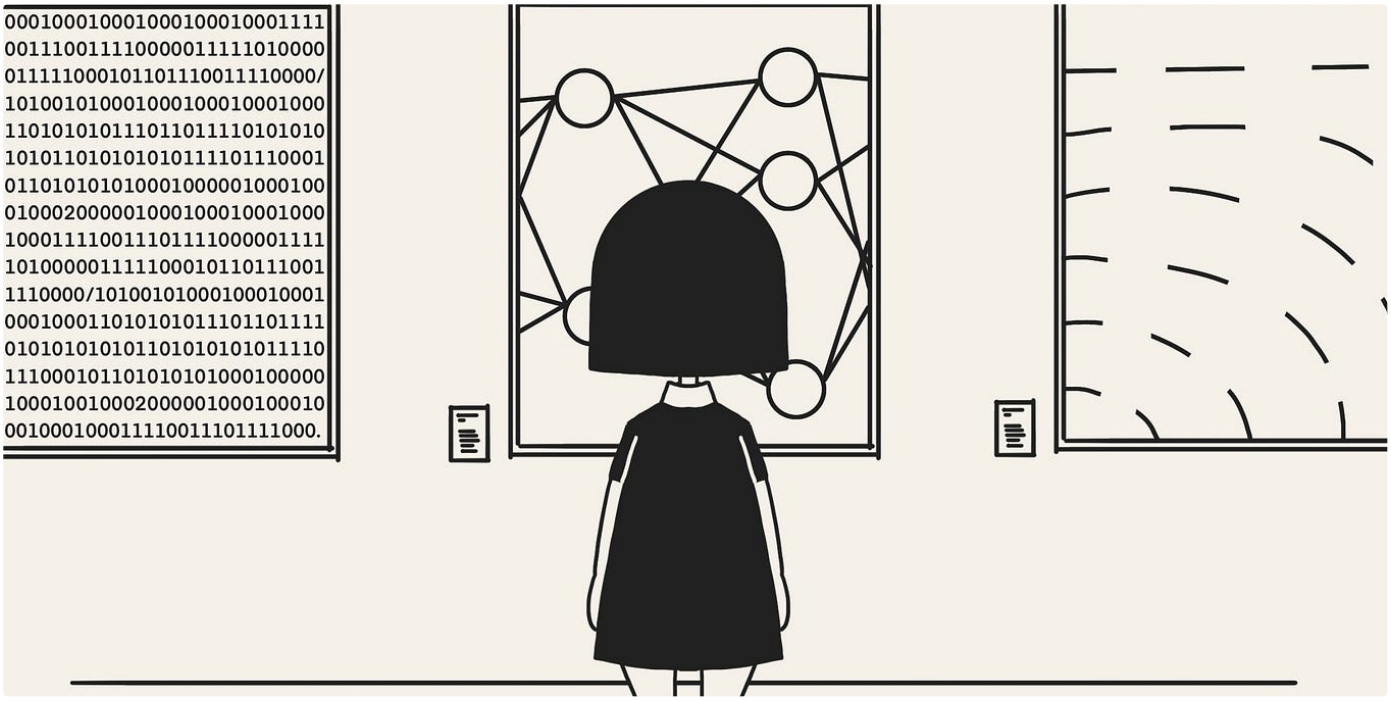


Giorgos Myrianthous in Towards Data Science

How To Fix ModuleNotFoundError: No module named ‘sklearn’

Understanding how to properly install and import scikit-learn in Python

· 6 min read · Mar 23, 2022



Leonie Monigatti in Towards Data Science

10 Exciting Project Ideas Using Large Language Models (LLMs) for Your Portfolio

Learn how to build apps and showcase your skills with large language models (LLMs). Get started today!

· 11 min read · May 15

1.6K 10

Miriam Santos in Towards Data Science

A Data Scientist's Essential Guide to Exploratory Data Analysis

Best Practices, Techniques, and Tools to Fully Understand Your Data

11 min read · May 30

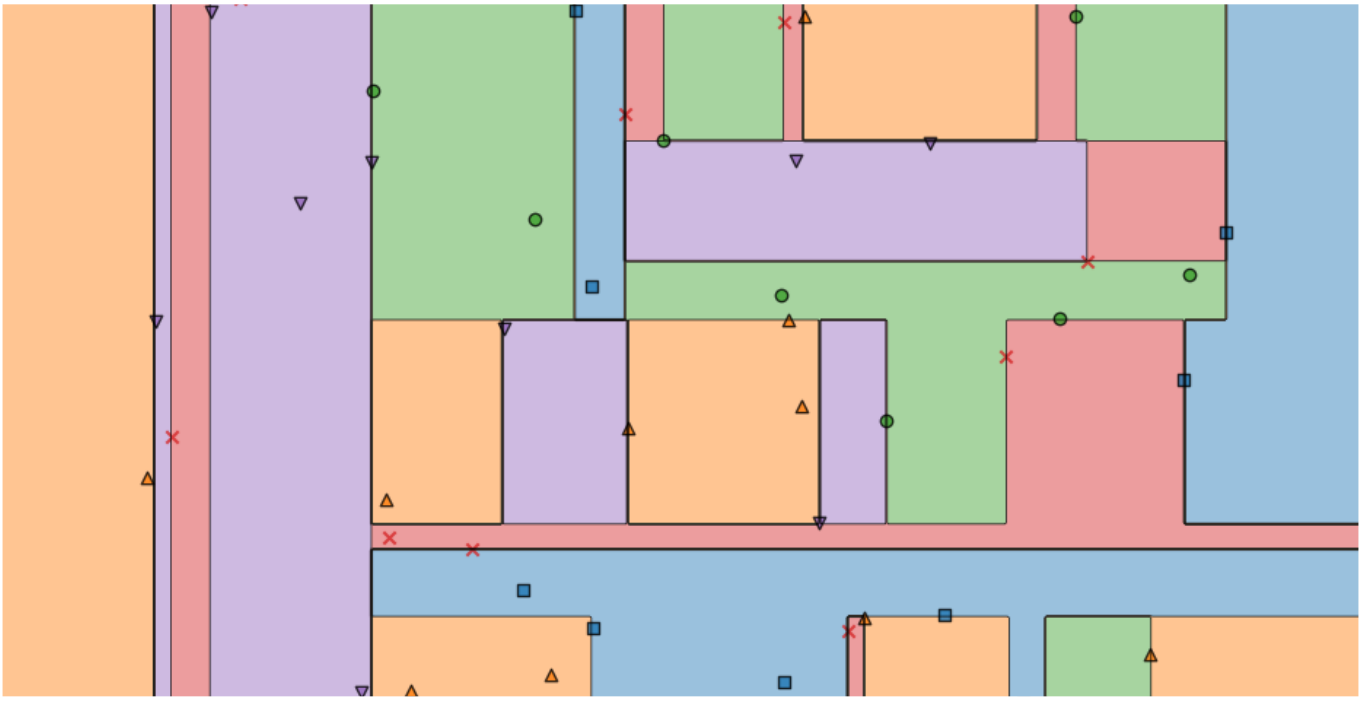
643 7

Giorgos Myrianthous in Level Up Coding

How to fix support for password authentication was removed on GitHub

Understanding how to configure and use Access Tokens in GitHub

· 4 min read · Dec 19, 2021

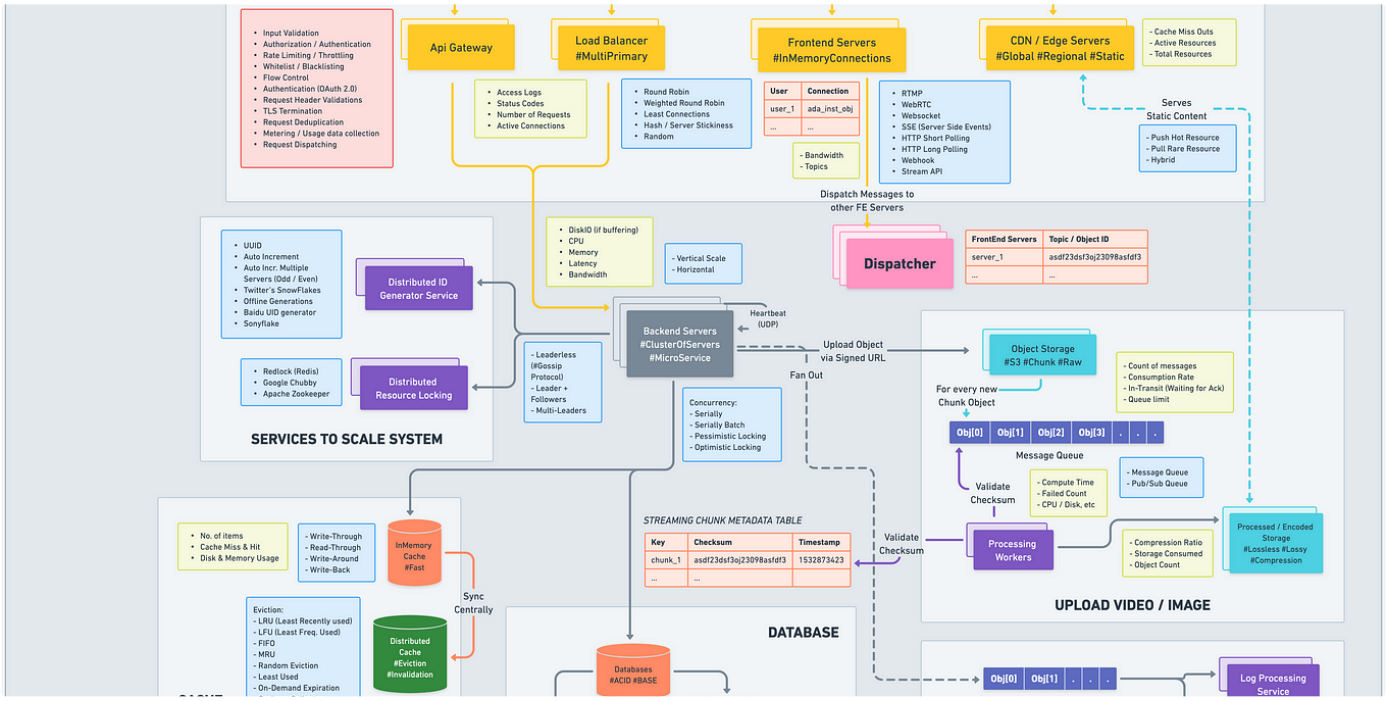


Dr. Robert Kübler in Towards Data Science

Understanding by Implementing: Decision Tree

Learn how a Decision Tree works and implement it in Python

· 11 min read · Nov 25, 2022



Love Sharma in Dev Genius

System Design Blueprint: The Ultimate Guide

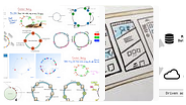
Developing a robust, scalable, and efficient system can be daunting. However, understanding the key concepts and components can make the...

· 9 min read · Apr 20

Lists

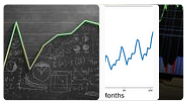
Coding & Development

11 stories · 15 saves



General Coding Knowledge

20 stories · 26 saves

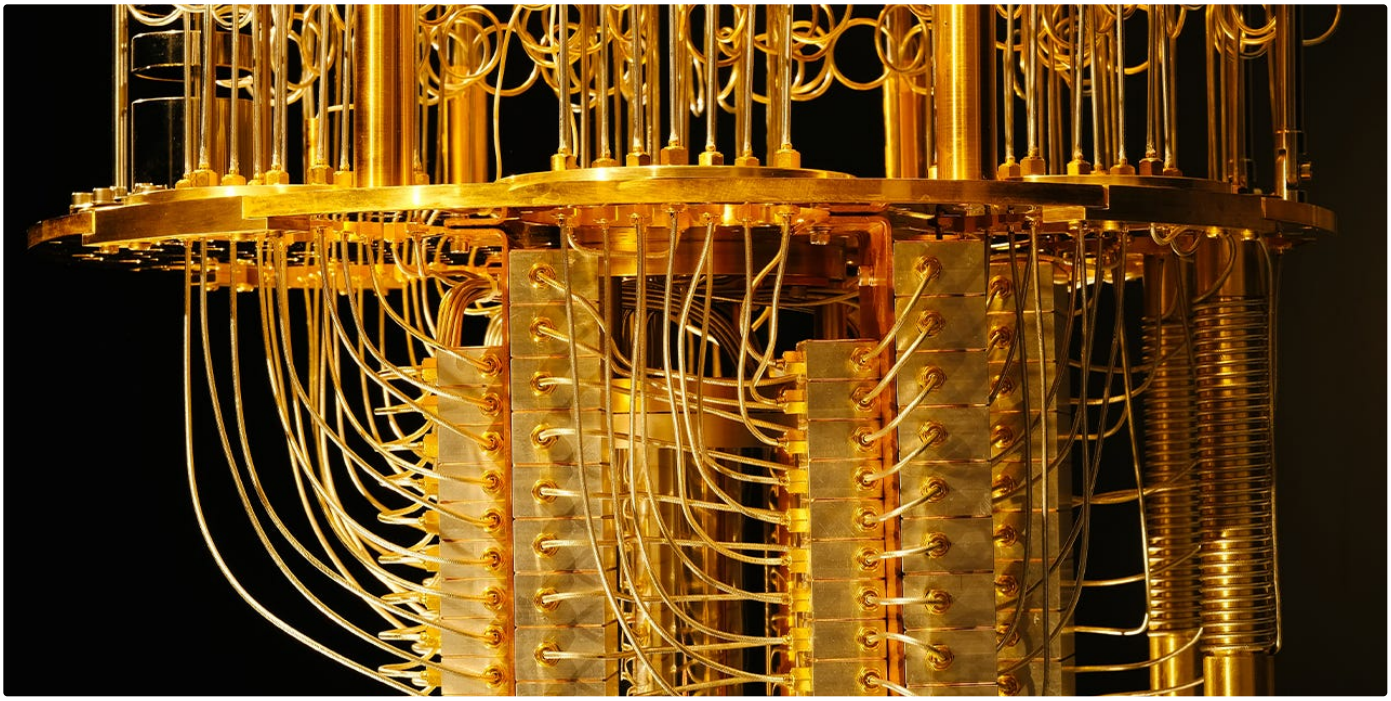


Predictive Modeling w/ Python

18 stories · 35 saves

It's never too late or early to start something

10 stories · 8 saves

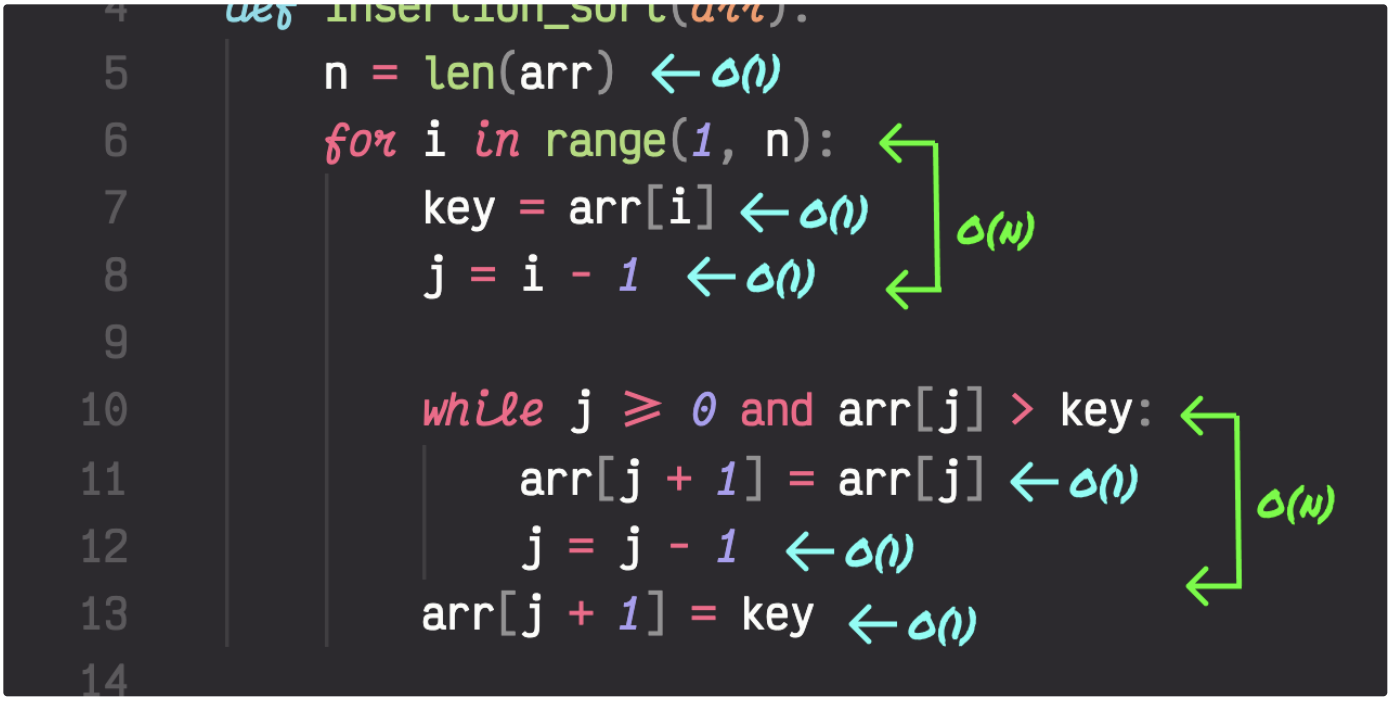


Tara Jones

Open source as a path to learning Quantum Computing

Quantum computing might sound intense, but the open source community is the way into a new career. Here's the projects to watch.

4 min read · Jun 11

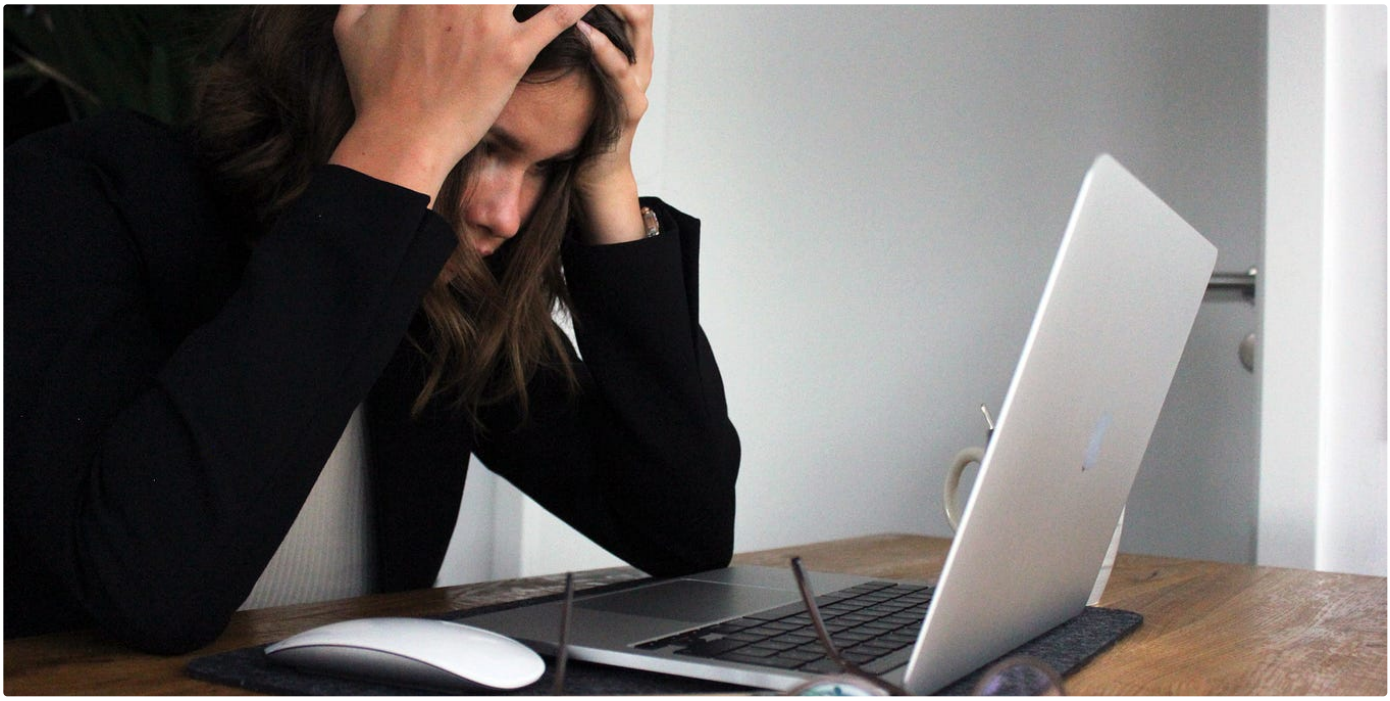


Abdulazeez Sherif in Python in Plain English

Top 8 Algorithms Every Programmer Should Know 💯

In programming, an algorithm is a set of instructions or a procedure for solving a specific problem or achieving a specific task...

7 min read · Jan 24

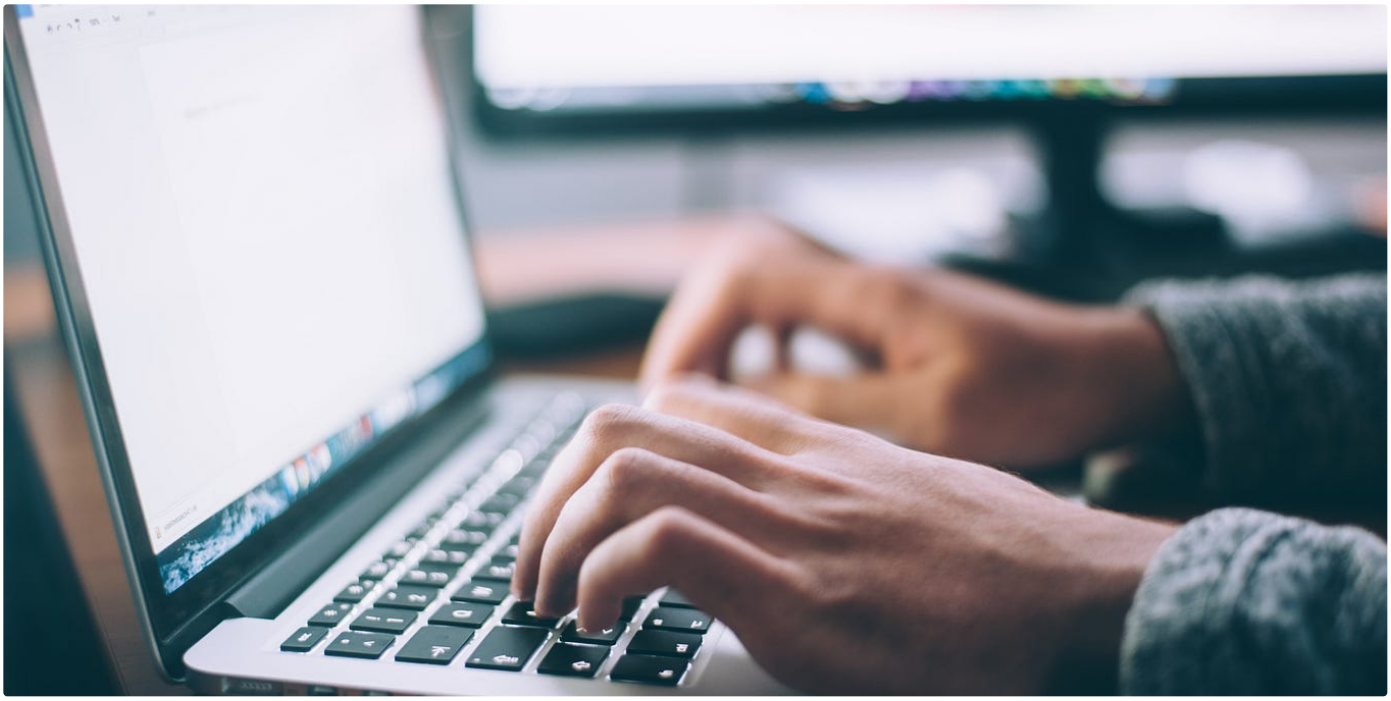


Bobby in Level Up Coding

10 Nooby Python Coding Mistakes You Should NEVER Make

Avoid these 10 nooby Python coding mistakes at all costs!

· 7 min read · Feb 21



Ekin Karabulut in Better Programming

How To Not Use GPUs

A steroid-filled intro to GPUs—for data scientists

10 min read · Jun 13