Object Oriented Programming in Python

By Usman Malik (https://twitter.com/usman_malikk) • October 24, 2018 • 2 Comments (/object-oriented-programming-in-python/#disqus_thread)

Introduction

Object-Oriented Programming (OOP) is a programming paradigm where different components of a computer program are modeled after real-world objects. An object is anything that has some characteristics and can perform a function.

Consider a scenario where you have to develop a Formula 1 car racing game using the object-oriented programming approach. The first thing you need to do is to identify real-world objects in the actual Formula 1 race. What are the entities in a Formula 1 race that have some characteristics and can perform any function? One

of the obvious answers to this question is the car. A car can have characteristics like engine capacity, make, model, manufacturer, and so on. Similarly, a car can be started, stopped, accelerated and so on. A driver can be another object in a Formula 1 race. A driver has a nationality, age, gender, and so on, and he can perform functionalities like driving the car, moving the steering or changing the transmission.

Just like in this example, in object-oriented programming we will create objects for the corresponding real-world entity.

It is important to mention here that object-oriented programming is not a language-dependent concept. It is a general programming concept and most of the modern languages, such as Java, C#, C++, and Python, support object-oriented programming. In this article, we will see a detailed introduction to Object-Oriented Programming in Python, but before that, we will see some of the advantages and disadvantages of object-oriented programming.

Pros and Cons of OOP

Following are some of the advantages of object-oriented programming:

- Object-oriented programming fosters reusability. A computer program is written in the form of objects and classes, which can be reused in other projects as well.
- The modular approach used in object-oriented programming results in highly maintainable code.
- In object-oriented programming, every class has a specific task. If an error occurs in one part of the code, you can rectify it locally without having to affect other parts of the code.
- Data encapsulation (which we will study later in the article) adds an extra layer of security to the program developed using the object-oriented approach.

Though object-oriented programming has several advantages as discussed, it has some downsides as well, some of which have been enlisted below:

- Detailed domain knowledge of the software being developed is needed in order to create objects. Not every entity in software is a candidate for being implemented as an object. It can be hard for newbies to identify this fine line.
- As you add more and more classes to the code, the size and complexity of the program grows exponentially.

In the next section, we will see some of the most important concepts of object-oriented programming.

As the name suggests, object-oriented programming is all about objects. However, before an object can be created we need to define the class for the object.

Class

A class in object-oriented programming serves as a blueprint for the object. A class can be considered as a map for the house. You can get an idea of what the house looks like by simply seeing the map. However, a class itself is nothing. For instance, a map is not a house, it only explains how the actual house will look.

The relationship between a class and object can be understood by looking at the relationship between a car and an Audi. An Audi is actually a car. However, there is no such thing as a car only. A car is an abstract concept, it is actually implemented in the form of Toyota, Ferrari, Honda, etc.

The keyword `class` is used in order to create a class in Python. The name of the class follows the `class` keyword, followed by the colon character. The body of the class starts on a new line, indented one tab from the left.

Let's see how we can create a very basic class in Python. Take a look at the following code:

```
# Creates class Car
class Car:

    # create class attributes
    name = "c200"
    make = "mercedez"
    model = 2008

    # create class methods
    def start(self):
        print ("Engine started")

    def stop(self):
        print ("Engine switched off")
```

In the example above, we create a class named `Car` with three attributes: `name`, `make`, and `model`. The `car` class also contains two methods: `start()` and `stop()`.

Objects

Earlier, we said that a class provides a blueprint. However, to actually use the objects and methods of a class, you need to create an *object* out of that class. There are few class methods and attributes that can be used without an object, which we will see in the later section. For now, just keep in mind that by default, we need to create an object of a class before we can use its methods and attributes.

An object is also called an instance; therefore, the process of creating an object of a class is called *instantiation*. In Python, to create an object of a class we simply need to write the class name followed by opening and closing parenthesis.

Let's create an object of the `Car` class that we created in the last section.

```
# Creates car_a object of Car class
car_a = Car()

# Creates car_b object of car class
car_b = Car()
```

In the script above, we created two objects of the car class: `car_a` and `car_b`. To check the type of the objects we created, we can use the `type` method and pass it the name of our object. Execute the following script:

```
print(type(car_b))
```

In the output, you will see:

```
<class '__main__.Car'>
```

Which says that the type of `car_b` object is a class `Car`.

At this point we've created our class and the corresponding objects. Now is the time to access class attributes and call class method using the class object. To do so, you simply have to write the object name, followed by dot operator and the name of the attribute or the method that you want to access or call, respectively. Take a look at the following example:

```
car_b.start()
```

In the script above, we call the `start()` method via the `car_b` object. The output will be as follows:

```
Engine started
```

Similarly, you can access an attribute using the following syntax:

```
print(car_b.model)
```

In the output, you will see the value of the `model` attribute, as shown below:

```
2008
```

Attributes

In the previous section, we saw how we can create objects of a class and can use those objects to access the attributes of a class.

In Python, every object has some default attributes and methods in addition to user-defined attributes. To see all the attributes and methods of an object, the built-in `dir()` function can be used. Let's try to see all the attributes of the `car_b` object that we created in the last section. Execute the following script:

```
dir(car_b)
```

In the output, you will see the following attributes:

```
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 'make',
 'model',
 'name',
 'start',
 'stop']
```

This built-in function is useful for inspecting all of the attributes and functions of an object, especially when used via Python's REPL.

Class vs Instance Attributes

Attributes can be broadly categorized into two types: Class attributes and Instance attributes. Class attributes are shared by all the objects of a class while instance attributes are the exclusive property of the instance.

Remember, an instance is just another name for the object. Instance attributes are declared inside any method while class attributes are declared outside of any method. The following example clarifies the difference:

```
class Car:

    # create class attributes
    car_count = 0

    # create class methods
    def start(self, name, make, model):
        print ("Engine started")
        self.name = name
        self.make = make
        self.model = model
        Car.car_count += 1
```

In the script above, we create a class `Car` with one class attribute `car_count` and three instance attributes `name`, `make` and `mode`. The class contains one method `start()` which contains the three instance attributes. The values for the instance attributes are passed as arguments to the `start()` method. Inside the `start` method, the `car_count` attribute is incremented by one.

It is important to mention that inside the method, the instance attributes are referred using the `self` keyword, while class attributes are referred by the class name.

Let's create an object of the `Car` class and call the `start()` method.

```
car_a = Car()
car_a.start("Corrola", "Toyota", 2015)
print(car_a.name)
print(car_a.car_count)
```

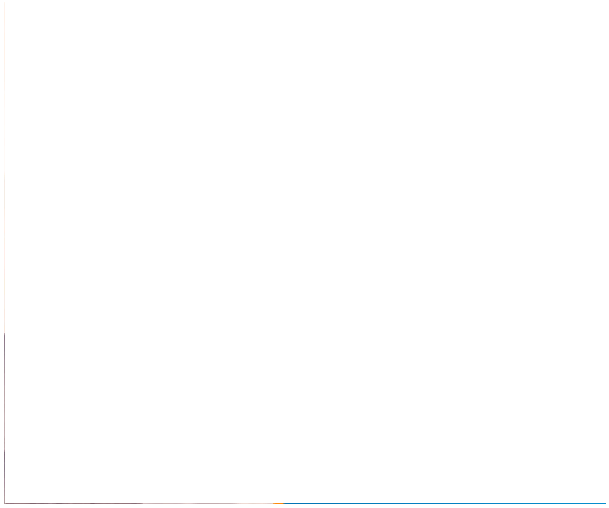
In the above script we print the instance attribute `name` and class attribute `car_count` . You will see in the output that the `car_count` attribute will have a value of 1, as shown below:

```
Engine started
Corrola
1
```

Now, let's create another object of the `car` class and call the `start()` method.

```
car_b = Car()
car_b.start("City", "Honda", 2013)
print(car_b.name)
print(car_b.car_count)
```

Now if you print the value of the `car_count` attribute, you will see 2 in the output. This is because the `car_count` attribute is a class attribute and hence it is shared between the instances. The `car_a` object incremented its value to 1, while `car_b` object incremented it again, hence the final value became 2. The output looks like this:



```
Engine started
City
2
```

Methods

As we described earlier, in object-oriented programming, the methods are used to implement the functionalities of an object. In the previous section, we created `start()` and `stop()` methods for the `Car` class. Till now, we have been using the objects of a class in order to call the methods. However, there is a type of method that can be called directly using the class name. Such a method is called a *static* method.

Static Methods

To declare a static method, you have to specify the `@staticmethod` descriptor before the name of the method as shown below:

```
class Car:

    @staticmethod
    def get_class_details():
        print ("This is a car class")

Car.get_class_details()
```

In the above script, we create a class `Car` with one static method `get_class_details()`. Let's call this method using the class name.

```
Car.get_class_details()
```

You can see that we did not need to create an instance of the `Car` class in order to call the `get_class_details()` method, rather we simply used the class name. It is important to mention that static methods can only access *class* attributes in Python.

Returning Multiple Values from a Method

One of the best features of the Python language is the ability of class methods to return multiple values. Take a look at the following example:

```
class Square:

    @staticmethod
    def get_squares(a, b):
        return a*a, b*b

print(Square.get_squares(3, 5))
```

In the above script, we created a class named `Square` with one static method `get_squares()`. The method takes two parameters; multiply each parameter with itself and returns both the results using `return` statement. In the output of the script above, you will see the squares of 3 and 5.

The str Method

Till now we have been printing attributes using the `print()` method. Let's see what happens if we print the object of a class.

To do so we'll create a simple `Car` class with one method and try to print the object of the class to the console. Execute the following script:

```
class Car:

    # create class methods
    def start(self):
        print ("Engine started")

car_a = Car()
print(car_a)
```

In the script above we create `car_a` object of the `Car` class and print its value on the screen. Basically here we are treating `car_a` object as a string. The output looks likes this:

```
<__main__.Car object at 0x000001CCCF4335C0>
```

The output shows the memory location where our object is stored. Every Python object has a `__str__` method by default. When you use the object as a string, the `__str__` method is called, which by default prints the memory location of the

object. However, you can provide your own definition for the `__str__` method as well. For instance, look at the following example:

```
# Creates class Car
class Car:

    # create class methods

    def __str__(self):
        return "Car class Object"

    def start(self):
        print ("Engine started")

car_a = Car()
print(car_a)
```

In the script above, we override the `__str__` method by providing our own custom definition for the method. Now, if you print the `car_a` object, you will see the message "Car class Object" on the console. This is the message that we printed inside our custom the `__str__` method.

Using this method you can create custom and more meaningful descriptions for when an object is printed. You could even display some of the data within the class, like the `name` of a `Person` class.

Constructors

A constructor is a special method that is called by default whenever you create an object of a class.

To create a constructor, you have to create a method with keyword `__init__`. Take a look at the following example:

```
class Car:

    # create class attributes
    car_count = 0

    # create class methods
    def __init__(self):
        Car.car_count +=1
        print(Car.car_count)
```

In the script above, we create a `Car` class with one class attribute `car_count`. The class contains a constructor which increments the value of `car_count` and prints the resultant value on screen.

Now, whenever an object of the `Car` class will be created the constructor will be called, the value of the `car_count` will be incremented and displayed on the screen. Let's create a simple object and see what happens:

```
car_a = Car()
car_b = Car()
car_c = Car()
```


In the output, you will see a value of 1, 2, and 3 printed since with every object the value of `car_count` variable is incremented and displayed on the screen.

Except for the name, the constructor can be used as an ordinary method. You can pass and receive values from a constructor. It is usually used in this way when you want to initialize attribute values upon instantiating the class.

Local vs Global Variables

We know that there are two types of Python attributes, instance attributes, and class attributes. The attributes of a class are also referred to as variables. Depending on the scope, variables can also be categorized into two types: Local variables and Global variables.

Local Variables

A local variable in a class is a variable that can only be accessed inside the code block where it is defined. For instance, if you define a variable inside a method, it cannot be accessed anywhere outside that method. Look at the following script:

```
# Creates class Car
class Car:
    def start(self):
        message = "Engine started"
        return message
```

In the script above we create a local variable `message` inside the `start()` method of the `Car` class. Now let's create an object of the `Car` class and try to access the local variable `message` as shown below:

```
car_a = Car()
print(car_a.message)
```

The above script will return the following error:

```
AttributeError: 'Car' object has no attribute 'message'
```

This is because we cannot access the local variable outside the block in which the local variable is defined.

Global Variable

A global variable is defined outside of any code block e.g method, if-statements, etc. A global variable can be accessed anywhere in the class. Take a look at the following example.

```
# Creates class Car
class Car:
    message1 = "Engine started"

    def start(self):
        message2 = "Car started"
        return message2

car_a = Car()
print(car_a.message1)
```

In the script above, we created a global variable `message1` and printed its value on the screen. In the output, you will see the value of the `message1` variable, printed without an error.

It is important to mention that there is a difference between class and instance attributes, and local vs global variables. The class and instance attributes differ in the way they are accessed i.e. using class name and using instance name. On the other hand, local vs global variables differ in their scope, or in other words the place where they can be accessed. A local variable can only be accessed inside the method. Though in this article, both the local variable and instance attributes are defined inside the method, local attribute is defined with the self-keyword.

Access Modifiers

The access modifiers in Python are used to modify the default scope of variables. There are three types of access modifiers in Python: public, private, and protected.

Variables with the public access modifiers can be accessed anywhere inside or outside the class, the private variables can only be accessed inside the class, while protected variables can be accessed within the same package.

To create a private variable, you need to prefix double underscores with the name of the variable. To create a protected variable, you need to prefix a single underscore with the variable name. For public variables, you do not have to add any prefixes at all.

Let's see public, private, and protected variables in action. Execute the following script:

```
class Car:
    def __init__(self):
        print ("Engine started")
        self.name = "corolla"
        self.__make = "toyota"
        self._model = 1999
```

In the script above, we create a simple `Car` class with a constructor and three variables `name`, `make`, and `model`. The `name` variable is public while the `make` and `model` variables have been declared private and protected, respectively.

Let's create an object of the `Car` class and try to access the `name` variable. Execute the following script:

```
car_a = Car()
print(car_a.name)
```

Since `name` is a public variable, therefore we can access it outside the class. In the output, you will see the value for the `name` printed on the console.

Now let's try to print the value of the `make` variable. Execute the following script:

```
print(car_a.make)
```

In the output, you will see the following error message:

```
AttributeError: 'Car' object has no attribute 'make'
```

We have covered most of the basic object-oriented programming concepts in the last few sections. Now, let's talk about the pillars of the object-oriented programming: Polymorphism, Inheritance, and Encapsulation, collectively referred to as PIE.

Inheritance

Inheritance in object-oriented programming is pretty similar to real-world inheritance where a child inherits some of the characteristics from his parents, in addition to his/her own unique characteristics.

In object-oriented programming, inheritance signifies an IS-A relation. For instance, a car *is* a vehicle. Inheritance is one of the most amazing concepts of object-oriented programming as it fosters code re-usability.

The basic idea of inheritance in object-oriented programming is that a class can inherit the characteristics of another class. The class which inherits another class is called the *child class* or derived class, and the class which is inherited by another class is called *parent* or base class.

Let's take a look at a very simple example of inheritance. Execute the following script:

```
# Create Class Vehicle
class Vehicle:
    def vehicle_method(self):
        print("This is parent Vehicle class method")

# Create Class Car that inherits Vehicle
class Car(Vehicle):
    def car_method(self):
        print("This is child Car class method")
```

In the script above, we create two classes `Vehicle` class, and the `Car` class which inherits the `Vehicle` class. To inherit a class, you simply have to write the parent class name inside the parenthesis that follows the child class name. The `Vehicle` class contains a method `vehicle_method()` and the child class contains a method `car_method()`. However, since the `Car` class inherits the `Vehicle` class, it will *also* inherit the `vehicle_method()`.

Let's see this in action. Execute the following script:

```
car_a = Car()
car_a.vehicle_method() # Calling parent class method
```

In the script above, we create an object of the `Car` class and call the `vehicle_method()` using that `Car` class object. You can see that the `Car` class doesn't have any `vehicle_method()` but since it has inherited the `Vehicle` class that contains the `vehicle_method()`, the car class can also use it. The output looks likes this:

```
This is parent Vehicle class method
```

In Python, a parent class can have multiple children and similarly, a child class can have multiple parent classes. Let's take a look at the first scenario. Execute the following script:

```
# Create Class Vehicle
class Vehicle:
    def vehicle_method(self):
        print("This is parent Vehicle class method")

# Create Class Car that inherits Vehicle
class Car(Vehicle):
    def car_method(self):
        print("This is child Car class method")

# Create Class Cycle that inherits Vehicle
class Cycle(Vehicle):
    def cycleMethod(self):
        print("This is child Cycle class method")
```

In the script above the parent `Vehicle` class is inherited by two child classes `Car` and `Cycle`. Both the child classes will have access to the `vehicle_method()` of the parent class. Execute the following script to see that:

```
car_a = Car()
car_a.vehicle_method() # Calling parent class method
car_b = Cycle()
car_b.vehicle_method() # Calling parent class method
```

In the output, you will see the output of the `vehicle_method()` method twice as shown below:

```
This is parent Vehicle class method
This is parent Vehicle class method
```

You can see how a parent class can be inherited by two child classes. In the same way, a child can have multiple parents. Let's take a look at the example:

```
class Camera:
    def camera_method(self):
        print("This is parent Camera class method")

class Radio:
    def radio_method(self):
        print("This is parent Radio class method")

class CellPhone(Camera, Radio):
     def cell_phone_method(self):
        print("This is child CellPhone class method")
```

In the script above, we create three classes: `Camera`, `Radio`, and `CellPhone`. The `Camera` class and the `Radio` classes are inherited by the `CellPhone` class which means that the `CellPhone` class will have access to the methods of both `Camera` and `Radio` classes. The following script verifies this:

```
cell_phone_a = CellPhone()
cell_phone_a.camera_method()
cell_phone_a.radio_method()
```

The output looks likes this:

```
This is parent Camera class method
This is parent Radio class method
```

Polymorphism

The term polymorphism literally means having multiple forms. In the context of object-oriented programming, polymorphism refers to the ability of an object to behave in multiple ways.

Polymorphism in programming is implemented via method-overloading and method overriding.

Method Overloading

Method overloading refers to the property of a method to behave in different ways depending upon the number or types of the parameters. Take a look at a very simple example of method overloading. Execute the following script:

```
# Creates class Car
class Car:
   def start(self, a, b=None):
        if b is not None:
            print (a + b)
        else:
            print (a)
```

In the script above, if the `start()` method is called by passing a single argument, the parameter will be printed on the screen. However, if we pass 2 arguments to the `start()` method, it will add both the arguments and will print the result of the sum.

Let's try with single argument first:

```
car_a = Car()
car_a.start(10)
```

In the output, you will see 10. Now let's try to pass 2 arguments:

```
car_a.start(10,20)
```

In the output, you will see 30.

Method Overriding

Method overriding refers to having a method with the same name in the child class as in the parent class. The definition of the method differs in parent and child classes but the name remains the same. Let's take a simple example method overriding in Python.

```
# Create Class Vehicle
class Vehicle:
    def print_details(self):
        print("This is parent Vehicle class method")

# Create Class Car that inherits Vehicle
class Car(Vehicle):
    def print_details(self):
        print("This is child Car class method")

# Create Class Cycle that inherits Vehicle
class Cycle(Vehicle):
    def print_details(self):
        print("This is child Cycle class method")
```

In the script above the `Car` and `Cycle` classes inherit the `Vehicle` class. The vehicle class has `print_details()` method, which is overridden by the child classes. Now if you call the `print_details()` method, the output will depend upon the object through which the method is being called. Execute the following script to see this concept in action:

```
car_a = Vehicle()
car_a. print_details()

car_b = Car()
car_b.print_details()

car_c = Cycle()
car_c.print_details()
```

The output will look like this:

```
This is parent Vehicle class method
This is child Car class method
This is child Cycle class method
```

You can see that the output is different, although the `print_details()` method is being called through derived classes of the same base class. However, since the child classes have overridden the parent class method, the methods behave differently.

Encapsulation

Encapsulation is the third pillar of object-oriented programming. Encapsulation simply refers to data hiding. As a general principle, in object-oriented programming, one class should not have direct access to the data of the other class. Rather, the access should be controlled via class methods.

To provide controlled access to class data in Python, the access modifiers and properties are used. We have already seen access modifiers, in this section, we will see properties in action.

Suppose we want to ensure that the car model should always be between 2000 and 2018. If a user tries to enter a value less than 2000 for the car model, the value is automatically set to 2000 and if the entered value is greater than 2018, it should be set to 2018. If the value is between 2000 and 2018, it should not be changed. We can create a property for the model attribute which implements this logic as follows:

```
# Creates class Car
class Car:

    # Creates Car class constructor
    def __init__(self, model):
        # initialize instance variables
        self.model = model

    # Creates model property
    @property
    def model(self):
        return self.__model

    # Create property setter
    @model.setter
    def model(self, model):
        if model < 2000:
            self.__model = 2000
        elif model > 2018:
            self.__model = 2018
        else:
            self.__model = model

    def getCarModel(self):
        return "The car model is " + str(self.model)

carA = Car(2088)
print(carA.getCarModel())
```

A property has three parts. You have to define the attribute, which is `model` in the above script. Next, you have to define the property for the attribute using the @property decorator (/the-python-property-decorator/). Finally, you have to create property setter which is `@model.setter` descriptor in the above script.

Now, if you try to enter a value greater than 2018 for the model attribute, you will see that the value is set to 2018. Let's test this. Execute the following script:

```
car_a = Car(2088)
print(car_a.get_car_model())
```

Here we are passing 2088 as the value for `model`, however if you print the value for the `model` attribute via `get_car_model()` function, you will see 2018 in the output.

Conclusion

In this article, we studied some of the most important object-oriented programming concepts. Object-oriented programming is one of the most famous and commonly used programming paradigms. The importance of object-oriented programming is reflected by the fact that most of the modern programming languages are either fully object-oriented or support object-oriented programming.

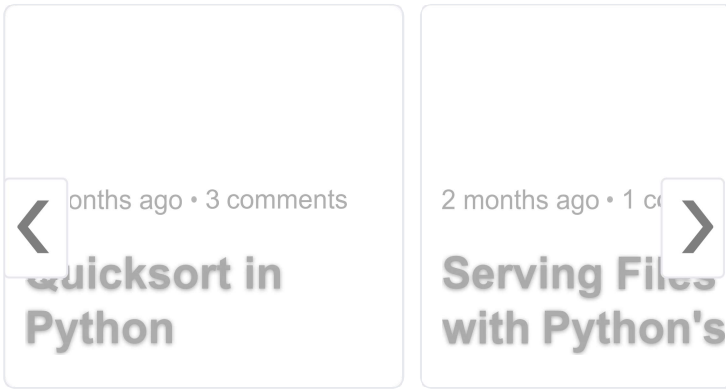
---

python (/tag/python/)

witter.com/share?
20Oriented%20Programming%20in%20Python&url=https://stackabuse.com/object-
ramming-in-python/)
vww.facebook.com/sharer/sharer.php?u=https://stackabuse.com/object-oriented-
-in-python/)
vww.linkedin.com/shareArticle?mini=true%26url=https://stackabuse.com/object-
ramming-in-python/%26source=https://stackabuse.com)

Subscribe

ALSO ON STACKABUSE



Comments Community 1 Login

Recommend 1 Tweet Share

Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Monil Bid • 5 months ago • edited

Hey Usman. This is a great article. Thanks for sharing it.
I wanted to point out a bug in the code under the Encapsulation section. The `__init__` should declare `self.__model = model`. The current code works because it creates `self.__model` in the `model.setter`, but really it should be created in the `__init`

‹ Previous Post (/java-rest-api-documentation-with-swagger2/)

Next Post › (/python-gui-development-with-tkinter-part-2/)

Ad

Follow Us

(https://twitter.com/StackAbuse) (https://www.facebook.com/stackabuse) (https://stackabuse.com/rss/)

Newsletter

Subscribe to our newsletter! Get occassional tutorials, guides, and reviews in your inbox.

Enter your email...

Subscribe

No spam ever. Unsubscribe at any time.

Ad

Interviewing for a job?

(http://stackabu.se/daily-coding-problem)

- Improve your skills by solving one coding problem every day
- Get the solutions the next morning via email
- Practice on **actual problems** asked by top companies, like:

</> Daily Coding Problem (http://stackabu.se/daily-coding-problem)

Want a remote job?

Sr. Security Engineer
HyperScience *1 day ago*
(https://hireremote.io/remote-job/1113-sr.-security-engineer-at-hyperscience)
java (https://hireremote.io/remote-java-jobs)
python (https://hireremote.io/remote-python-jobs) ruby
(https://hireremote.io/remote-ruby-jobs)
docker (https://hireremote.io/remote-docker-jobs)

Infrastructure Engineer
Toptal *2 days ago* (https://hireremote.io/remote-job/1112-infrastructure-engineer-at-toptal)
ansible (https://hireremote.io/remote-ansible-jobs) terraform
(https://hireremote.io/remote-terraform-jobs)
version (https://hireremote.io/remote-version-jobs) control
(https://hireremote.io/remote-control-jobs)

Engineer, Platform
Auth0 *3 days ago* (https://hireremote.io/remote-job/1110-engineer,-platform-at-auth0)

postgresql (https://hireremote.io/remote-postgresql-jobs) database (https://hireremote.io/remote-database-jobs) mongodb (https://hireremote.io/remote-mongodb-jobs) go (https://hireremote.io/remote-go-jobs)

More jobs (https://hireremote.io)

Jobs via HireRemote.io (https://hireremote.io)

Recent Posts

Using Spies for Testing in JavaScript with Sinon.js (/using-spies-for-testing-in-javascript-with-sinon/)

A Beginner-Level Introduction to MongoDB with Node.js (/a-beginner-level-introduction-to-mongodb-with-node-js/)

Reactive Programming with Spring 5 WebFlux (/reactive-programming-with-spring-5-webflux/)

Tags

algorithms (/tag/algorithms/) amqp (/tag/amqp/) angular (/tag/angular/) announcements (/tag/announcements/) apache (/tag/apache/) api (/tag/api/) arduino (/tag/arduino/) artificial intelligence (/tag/artificial-intelligence/) asynchronous (/tag/asynchronous/) aws (/tag/aws/)

Follow Us

(https://twitter.com/StackAbuse) (https://www.facebook.com/stackabuse) (https://stackabuse.com/rss/)