



★ Get unlimited access to the best of Medium for less than \$1/week. [Become a member](#)



# Python Dependency Inversion



Zack Bunch · [Follow](#)

5 min read · Jul 24, 2021



180



3



Encapsulation, Abstraction, Inheritance and Polymorphism. You may have heard these four words tossed around while studying Computer Science or even during interviews. They are the four principles of Object Oriented Programming and the backbone to clean and concise code when working on large scale projects.

To be completely honest, I truly did not understand the benefits of these four concepts while in school for six years. However, once I started working in industry and began reading a code base that has been around for 16+ years, it all started to click.

Today, I want to focus on Abstraction or Dependency Inversion if you're familiar with SOLID principles. The basic idea of abstraction is as simple as looking at something like switches in your home. Your home may contain numerous switches that control lights, fans and even plumbing. As a user, you only care about an object being turned on or off when the switch is used.

You don't care about what is happening behind the scenes. In code, we can model a lightbulb using a class as such:

```
1 class LightBulb:
2     def turn_on(self):
3         print("Lightbulb: turned on")
4
5     def turn_off(self):
6         print("Lightbulb: turned off")
```

We then could model the switch to control the behavior of a lightbulb. If the switch is flipped on or pressed, we expect an action to be sent to the lightbulb.

```
1 class PowerSwitch:
2     def __init__(self, l: LightBulb):
3         self.lightBulb = l
4         self.on = False
5
6     def press(self):
7         if self.on:
8             self.lightBulb.turn_off()
9         else:
10             self.lightBulb.turn_on()
11             self.on = True
```

Now, let's pull all this together and create an instance of Lightbulb named hueBulb. We then create an instance of the PowerSwitch object called switch and pass in the hueBulb instance.

```
1 class LightBulb:
2     def turn_on(self):
3         print("Lightbulb: turned on")
4
5     def turn_off(self):
6         print("Lightbulb: turned off")
7
8 class PowerSwitch:
9     def __init__(self, l: LightBulb):
10        self.lightBulb = l
11        self.on = False
12
13    def press(self):
14        if self.on:
15            self.lightBulb.turn_off()
16        else:
17            self.lightBulb.turn_on()
18            self.on = True
19
20
21 hueBulb = LightBulb()
22 switch = PowerSwitch(hueBulb)
23 switch.press()
24 switch.press()
25
```

The output of this program prints: Lightbulb: turned on and Lightbulb: turned off

In the example above, there is a clear dependency between the lightbulb and the power switch because the power switch object takes in a lightbulb and then directly calls the turn off and turn on method on that instance. While this code may work and get the job done, it does not satisfy SOLID. To adhere to the principle of dependency inversion (the D in SOLID), we need to ensure that high-level modules do not depend on low level modules, but instead depend on abstractions. The abstraction should not depend on details, instead the details should depend on abstractions. In the next section, we will apply dependency inversion to remove the dependency of a power switch on the lightbulb. In order to that, we're going to need an abstract base class. Abstract base classes are not built into Python, but Python does provide a module that supports abstract classes. Let's begin the process of creating an Abstract base class.

We will create a new class named Switchable. It will have the same methods as Lightbulb. We also need to import ABC and abstractmethod to implement the Abstract Base Class. In the code below, notice that ABC is passed in as an argument to Switchable. We then wrap our methods with the @abstractmethod decorator.

```
1 from abc import ABC, abstractmethod
2
3
4 class Switchable(ABC):
5     @abstractmethod
6     def turn_on(self):
7         pass
8
9     @abstractmethod
10    def turn_off(self):
11        pass
```

If you instantiate Switchable and run the program, you will get an error. To use Switchable, we must create sub-classes that inherit from switchable. For our example, our sub-class will be LightBulb. Now, that LightBulb is a subclass of Switchable, it implements the interface that is defined in Switchable.

```
1 from abc import ABC, abstractmethod
2
3
4 class Switchable(ABC):
5     @abstractmethod
6     def turn_on(self):
7         pass
8
9     @abstractmethod
10    def turn_off(self):
11        pass
12
13 class LightBulb(Switchable):
14     def turn_on(self):
15         print("Lightbulb: turned on")
16
17     def turn_off(self):
18         print("Lightbulb: turned off")
```

After you run the program with these modifications, you will see that it runs as expected. So what was the point then? Well, when we passed `Switchable` to our `LightBulb` class, we created a contract between the two. If `LightBulb` did not have the `turn_on` method and you ran the program, you will receive the following error.

```
1 hueBulb = LightBulb()  
2 TypeError: Can't instantiate abstract class LightBulb with abstract method turn_on
```

This happened because LightBulb broke the contract and did not implement everything as defined in Switchable. So to answer our question of why is this useful? We can use this as a way of tracking what still needs to be implemented.

Now let's actually apply the Dependency Inversion Principle to our program. To do so, we will remove the dependency LightBulb from the PowerSwitch class by renaming LightBulb to Switchable.



```
1 from abc import ABC, abstractmethod
2
3
4 class Switchable(ABC):
5     @abstractmethod
6     def turn_on(self):
7         pass
8
9     @abstractmethod
10    def turn_off(self):
11        pass
12
13 class LightBulb(Switchable):
14     def turn_on(self):
15         print("Lightbulb: turned on")
16
17     def turn_off(self):
18         print("Lightbulb: turned off")
19
20 class PowerSwitch:
21     def __init__(self, c: Switchable):
22         self.client = c
23         self.on = False
24
25     def press(self):
26         if self.on:
27             self.client.turn_off()
28         else:
29             self.client.turn_on()
30             self.on = True
31
32
```

```
33 hueBulb = LightBulb()
34 switch = PowerSwitch(hueBulb)
35 switch.press()
36 switch.press()
37
```

Now if you run the program again, the lightbulb is still turned on and off. Nothing about the functionality changed, however we did remove the dependency between lightbulb and PowerSwitch. The PowerSwitch is now dependent on the Switchable class. So... now we can create other classes such as a fan that can provide the same functionality of turning something on and off.

```
1 class Fan(Switchable):
2     def turn_on(self):
3         print("Fan: turned on")
4
5     def turn_off(self):
6         print("Fan: turned off")
```

Instantiate a new Fan Object just like we did with the LightBulb.

```
1 bedroomFan = Fan()  
2 fanSwitch = PowerSwitch(bedroomFan)  
3 fanSwitch.press()  
4 fanSwitch.press()
```

If you run the program now, we see that the Light and Fan were both turned off and on.

```
1 Lightbulb: turned on  
2 Lightbulb: turned off  
3 Fan: turned on  
4 Fan: turned off
```

By implementing dependency inversion, we now have decoupled two classes through an interface, which in our case is Switchable. It is now clear to see

the usefulness that dependency inversion provides by reducing coupling between classes.

If you enjoyed this article or have any input feel free to reach out to me on [LinkedIn](#). Thanks for reading my first article!

Python

Software Engineering

Programming

Linux

**Written by Zack Bunch**

Follow

151 Followers

Software Engineer. Learn programming, software engineering and everything tech from this channel. With a special emphasis on python and C++ my channel

---

**More from Zack Bunch**

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

's transpose, denoted  $A^T$ , is

$$A^T = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{m1} \\ a_{21} & a_{22} & \dots & a_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \dots & a_{mn} \end{bmatrix}$$



 Zack Bunch

## Matrices with Numpy and SciPy

This article assumes a basic familiarity with linear algebra, a branch of mathematics that...

8 min read · Jan 11, 2022



36



...

 Zack Bunch

## When to use `__new__` in Python?

If you're new to Python or a seasoned developer chances are you have seen the...

3 min read · Sep 28, 2021



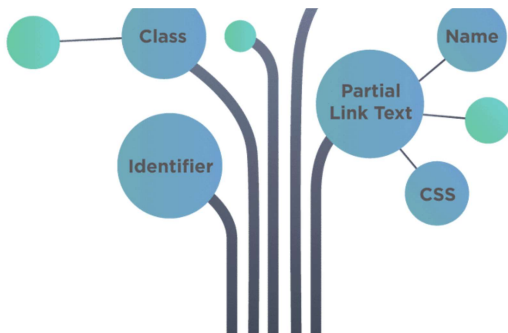
111



2



...



 Zack Bunch

## Page Object Model (POM) in Selenium Python

If you are looking to start automating your frontend testing or planning on using...

4 min read · Jan 10, 2022



16



3



...

 Zack Bunch

## How to create custom argparse types in Python?

As a developer, I find myself writing many scripts that require taking in arguments fro...

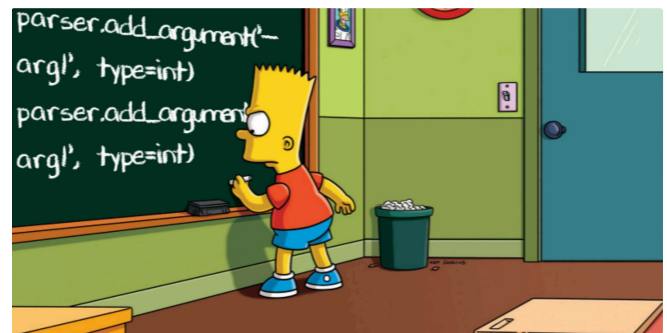
3 min read · Oct 12, 2021



71

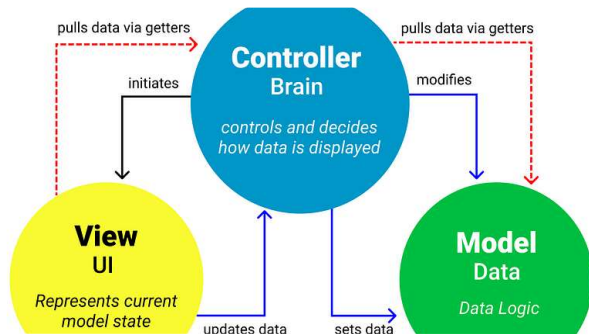


...



See all from Zack Bunch

## Recommended from Medium



Owuordove

### Hands-On Guide to Model-View-Controller (MVC) Architecture in...

Building scalable and maintainable software requires a robust architectural pattern, and...

3 min read · Jan 3, 2024



Sam Jones

### Repository Pattern is INSANE if you know how to use it properly—...

Introduction

3 min read · Nov 6, 2023



76



1



## Lists



### General Coding Knowledge

20 stories · 1123 saves



### Stories to Help You Grow as a Software Developer

19 stories · 984 saves



### Coding & Development


11 stories · 566 saves



### Predictive Modeling w/ Python

20 stories · 1111 saves



 Alicem Koyun in Towards Dev

## SOLID Principles in Python (Hands-on Instruction)

There are standardized design principles known as SOLID software principles, which...

9 min read · Mar 12, 2024



35



 Amir Lavasani

## Design Patterns in Python: Chain of Responsibility

Seamless Request Handling

10 min read · Oct 23, 2023



188



 Daniel Wu

## Exploring Pydantic and Dataclasses in Python: A...

Introduction

4 min read · Nov 12, 2023



138



1



 Francisco Escher in ITNEXT

## DInjections: building a Python dependency injection framework

Inspired by Uber's Golang FX framework

4 min read · Dec 11, 2023



16



See more recommendations