# Exercise: SOLID

Problems for exercise and homework for the [Python OOP Course @SoftUni](https://softuni.bg/courses/python-oop).

## Workers

You are provided with a code on which you have to apply the **DIP** (Dependency Inversion Principle) so that when adding new **worker classes**, the Manager class will work properly.

### Examples

|  |  |
| --- | --- |
| **Before** | **Result** |
| worker = Worker()  manager = Manager()  manager.set\_worker(worker)  manager.manage()  super\_worker = SuperWorker()  try:  manager.set\_worker(super\_worker)  except AssertionError:  print("manager fails to support super\_worker....") | I'm working!!  manager fails to support super\_worker.... |
| **After** | **Result** |
| worker = Worker()  manager = Manager()  manager.set\_worker(worker)  manager.manage()  super\_worker = SuperWorker()  try:  manager.set\_worker(super\_worker)  manager.manage()  except AssertionError:  print("manager fails to support super\_worker....") | I'm working!!  I work very hard!!! |

## Workers - Updated

You are provided with a code on which you have to apply the **ISP** (Interface Segregation Principle) by **splitting** the **Worker** class into two classes (**Workable** and **Eatable**), so the **Robot** class no longer needs to implement the **eat** method

### Examples

|  |  |
| --- | --- |
| **Before** | **Result** |
| manager = Manager()  manager.set\_worker(Worker())  manager.manage()  manager.lunch\_break()  manager.set\_worker(SuperWorker())  manager.manage()  manager.lunch\_break()  manager.set\_worker(Robot())  manager.manage()  manager.lunch\_break() | I'm normal worker. I'm working.  Lunch break....(5 secs)  I'm super worker. I work very hard!  Lunch break....(3 secs)  I'm a robot. I'm working....  I don't need to eat.... |
| **After** | **Result** |
| work\_manager = WorkManager()  break\_manager = BreakManager()  work\_manager.set\_worker(Worker())  break\_manager.set\_worker(Worker())  work\_manager.manage()  break\_manager.lunch\_break()  work\_manager.set\_worker(SuperWorker())  break\_manager.set\_worker(SuperWorker())  work\_manager.manage()  break\_manager.lunch\_break()  work\_manager.set\_worker(Robot())  work\_manager.manage()  try:  break\_manager.set\_worker(Robot())  break\_manager.lunch\_break()  except:  pass | I'm normal worker. I'm working.  Lunch break....(5 secs)  I'm super worker. I work very hard!  Lunch break....(3 secs)  I'm a robot. I'm working.... |

## Prisoner

You are provided with a code containing a class **Prisoner** and a **class Person**. A **prisoner** is obviously a **person**, but since a **prisoner** is **not free** to move an arbitrary distance, the **Person** class can be named **FreePerson**, then the idea that a **Prisoner inherits FreePerson** is **wrong**. Rewrite the code and apply the **LSP** (Liskov Substitution Principle).

### Examples

|  |  |
| --- | --- |
| **Before** | **Result** |
| prisoner = Prisoner()  print("The prisoner trying to walk to north by 10 and east by -3.")  try:  prisoner.walk\_north(10)  prisoner.walk\_east(-3)  except:  pass  print(f"The location of the prison: {prisoner.PRISON\_LOCATION}")  print(f"The current position of the prisoner: {prisoner.position}") | The prisoner trying to walk to north by 10 and east by -3.  The location of the prison: [3, 3]  The current position of the prisoner: [0, 13] |
| **After** | **Result** |
| prisoner = Prisoner()  print("The prisoner trying to walk to north by 10 and east by -3.")  try:  prisoner.walk\_north(10)  prisoner.walk\_east(-3)  except:  pass  print(f"The location of the prison: {prisoner.PRISON\_LOCATION}")  print(f"The current position of the prisoner: {prisoner.position}") | The prisoner trying to walk to north by 10 and east by -3.  The location of the prison: (3, 3)  The current position of the prisoner: (3, 3) |

## Shapes

You are provided with code containing **class Rectangle** and **class AreaCalculator**. Refactor the code using the **Open/Closed Principle** so that the code is open for extension (adding **more shapes**) but closed for modification.

### Examples

|  |  |
| --- | --- |
| **Before** | **Result** |
| shapes = [Rectangle(2, 3), Rectangle(1, 6)]  calculator = AreaCalculator(shapes)  print("The total area is: ", calculator.total\_area) | The total area is: 12 |
| **After** | **Result** |
| shapes = [Rectangle(1, 6), Triangle(2, 3)]  calculator = AreaCalculator(shapes)  print("The total area is: ", calculator.total\_area) | The total area is: 9.0 |

## Emails

You are provided with code containing **class IEmail** and **class Email**. The code does not follow the principle of **single responsibility** (the Email class has **2 responsibilities**). Create a new **class - IContent**, and a class that inherits it called **MyContent** to split the responsibilities.

### Examples

|  |  |
| --- | --- |
| **Before** | **Result** |
| email = Email('IM', 'MyML')  email.set\_sender('qmal')  email.set\_receiver('james')  email.set\_content('Hello, there!')  print(email) | Sender: I'm qmal  Receiver: I'm james  Content:  <myML>  Hello, there!  </myML> |
| **After** | **Result** |
| email = Email('IM')  email.set\_sender('qmal')  email.set\_receiver('james')  content = MyContent('Hello, there!')  email.set\_content(content)  print(email) | Sender: I'm qmal  Receiver: I'm james  Content:  <MyML>Hello, there!</MyML> |