Exercise: SOLID

Problems for exercise and homework for the Python OOP Course @SoftUni.

1. 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
<pre>worker = Worker()</pre>	I'm working!!
<pre>manager = Manager()</pre>	manager fails to support
manager.set_worker(worker)	super_worker
manager.manage()	
<pre>super_worker = SuperWorker()</pre>	
try:	
manager.set_worker(super_worker)	
except AssertionError:	
<pre>print("manager fails to support super_worker")</pre>	
After	Result
worker = Worker()	I'm working!!
manager = Manager()	I work very hard!!!
manager.set_worker(worker)	
manager.manage()	
<pre>super_worker = SuperWorker()</pre>	
try:	
manager.set_worker(super_worker)	
manager.manage()	
except AssertionError:	
<pre>print("manager fails to support super_worker")</pre>	

2. 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()	I'm normal worker. I'm working.
manager.set_worker(Worker())	Lunch break(5 secs)
manager.manage()	I'm super worker. I work very hard!











```
manager.lunch_break()
                                          Lunch break....(3 secs)
                                          I'm a robot. I'm working....
manager.set_worker(SuperWorker())
                                          I don't need to eat....
manager.manage()
manager.lunch_break()
manager.set_worker(Robot())
manager.manage()
manager.lunch_break()
                  After
                                                           Result
                                          I'm normal worker. I'm working.
work_manager = WorkManager()
break manager = BreakManager()
                                          Lunch break....(5 secs)
                                          I'm super worker. I work very hard!
work_manager.set_worker(Worker())
break_manager.set_worker(Worker())
                                          Lunch break....(3 secs)
work_manager.manage()
                                          I'm a robot. I'm working....
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
```

3. 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
<pre>prisoner = Prisoner() print("The prisoner trying to walk to north by 10 and east by -3.")</pre>	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]
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}")
                After
                                                             Result
                                      The prisoner trying to walk to north by 10 and
prisoner = Prisoner()
                                      east by -3.
print("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)
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}")
```

4. 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
<pre>shapes = [Rectangle(2, 3), Rectangle(1, 6)] calculator = AreaCalculator(shapes)</pre>	The total area is: 12
<pre>print("The total area is: ", calculator.total_area)</pre>	
After	Result
<pre>shapes = [Rectangle(1, 6), Triangle(2, 3)] calculator = AreaCalculator(shapes)</pre>	The total area is: 9.0
<pre>print("The total area is: ", calculator.total_area)</pre>	

5. 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
<pre>email = Email('IM', 'MyML')</pre>	Sender: I'm qmal
email.set_sender('qmal')	Receiver: I'm james
email.set_receiver('james')	Content:
email.set_content('Hello, there!')	<myml></myml>
<pre>print(email)</pre>	Hello, there!
After	Result
<pre>email = Email('IM')</pre>	Sender: I'm qmal
email.set_sender('qmal')	Receiver: I'm james
email.set_receiver('james')	Content:
<pre>content = MyContent('Hello, there!')</pre>	<myml>Hello, there!</myml>
email.set_content(content)	
print(email)	













