

# Software Engineering Lab #2

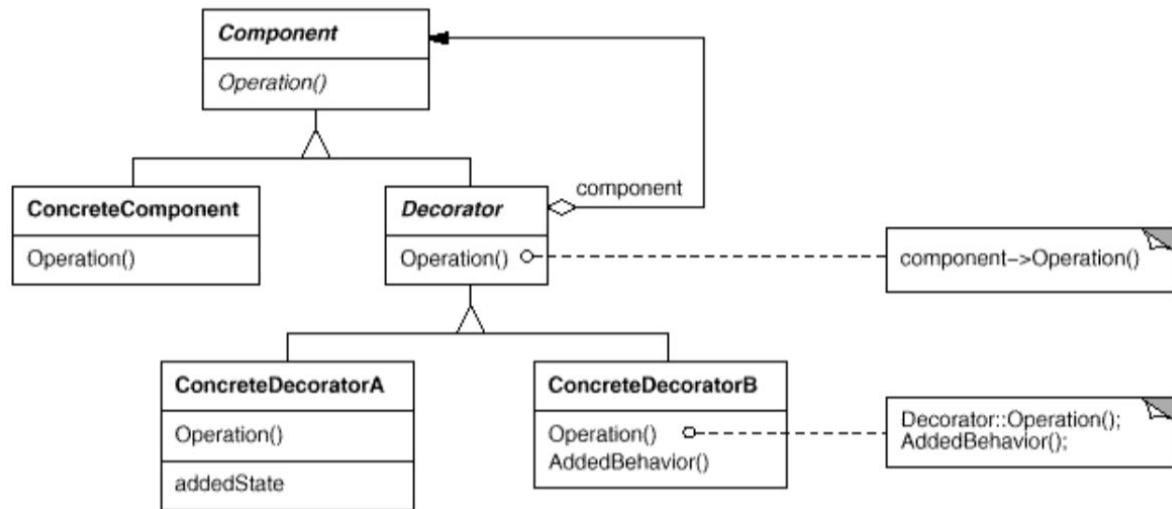
Decorator & Bridge Design Patterns

---

محمدرضا غمخوار ۹۵۱۰۶۴۹۴  
محمدرضا احمدخانی‌ها ۹۵۱۰۵۳۱۳

## الگوی Decorator

این الگو با هدف اضافه کردن وظیفه مندی و مسئولیت به شی هدف در زمان اجرا (به صورت پویا)، مورد استفاده قرار می‌گیرد. در بعضی مواقع این الگو جایگزینی انعطاف پذیر برای ساختن زیر کلاس است و حتی گاهی برای جلوگیری از گسترش بیش از حد کلاس و تبدیل شدنش به God class کاربرد دارد.



نمودار کلاسی الگوی decorator

برای اجرای این مراحل از زبان برنامه‌نویسی python و ابزار pytest استفاده می‌کنیم.

## مرحله اول - باز نویسی تست‌ها

از آنجایی که تست‌های نوشته شده به زبان جاوا هستند و ما برای پیاده‌سازی قصد استفاده از پایتون را داریم پس تست‌ها را باز نویسی می‌کنیم:

```
import pytest
from decorator.coffee_shop import *

class TestBeverage:

    def test_house_blend(self):
        # Pure HouseBlend
        beverage = HouseBlend()
        assert beverage.get_description() == "Delicious HouseBlend"
        assert beverage.cost() == 0.89

    def test_espresso(self):
        # Pure Espresso
        beverage = Espresso()
        assert beverage.get_description() == "Delicious Espresso"
        assert beverage.cost() == 1.99
```

```

def test_house_blend_with_steamed_milk(self):
    # HouseBlend + SteamedMilk
    beverage = SteamedMilk(HouseBlend())
    assert beverage.get_description() == "Delicious HouseBlend with milk"
    assert beverage.cost() == 0.89 + 0.1

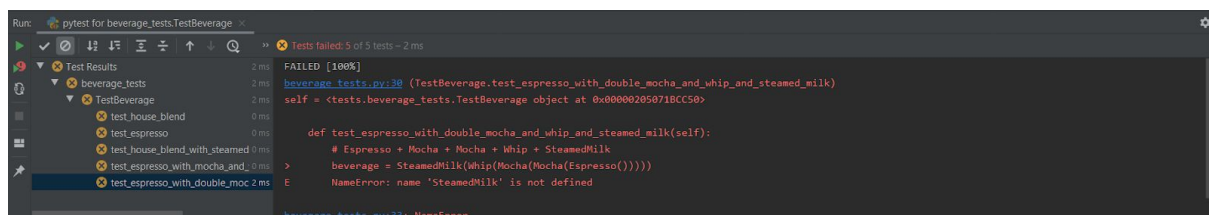
def test_espresso_with_mocha_and_whip(self):
    # Espresso + Mocha + Whip
    beverage = Whip(Mocha(Espresso()))
    assert beverage.get_description() == "Delicious Espresso with mocha with
whip"
    assert beverage.cost() == 1.99 + 0.2 + 0.1

def test_espresso_with_double_mocha_and_whip_and_steamed_milk(self):
    # Espresso + Mocha + Mocha + Whip + SteamedMilk
    beverage = SteamedMilk(Whip(Mocha(Mocha(Espresso()))))
    assert beverage.get_description() == "Delicious Espresso with mocha with
mocha with whip with milk"
    assert beverage.cost() == 1.99 + 0.2 + 0.2 + 0.1 + 0.1

```

## مرحله دو - شناسایی مشکلات کامپایلی تست‌ها

حال آن‌ها را اجرا کرده و شاهد اشتباهات (به واسطه‌ی پیاده‌سازی نشدن کدها نه اشتباه بودن رفتار کد) آن هستیم:



fail شدن همه تست‌ها بواسطه پیاده‌سازی نشدن کد

## مرحله سوم - رفع مشکلات کامپایلی تست‌ها با پیاده‌سازی رابط‌ها

حال به سراغ پیاده‌سازی رابط‌های کد می‌رویم تا تست‌ها صرفاً به واسطه اشتباه بود fail شوند و نه پیاده‌سازی نشدن.

پیاده‌سازی:

```

from abc import abstractmethod, ABC
class Beverage(ABC):
    @abstractmethod
    def get_description(self):
        pass
    @abstractmethod
    def cost(self):
        pass
class CondimentDecorator(Beverage, ABC):
    pass
class SteamedMilk(CondimentDecorator):

```

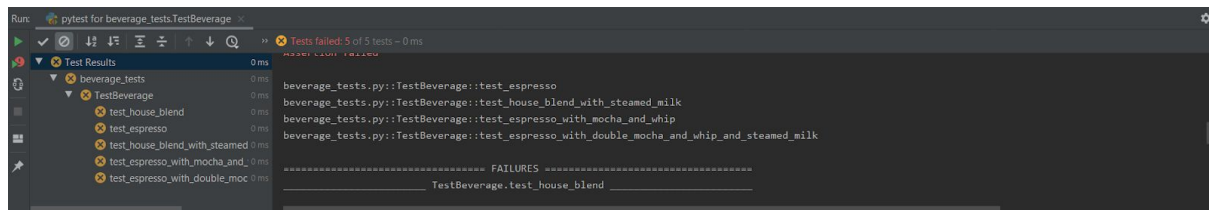
```

def get_description(self):
    pass
def cost(self):
    pass
class Whip(CondimentDecorator):
    pass
class Mocha(CondimentDecorator):
    pass
class HouseBlend(Beverage):
    pass
class Espresso(Beverage):
    pass

```

## مرحله چهارم - اجرای تست‌ها و شناسایی مشکلات زمان اجرا

حال دوباره تست‌ها را اجرا می‌کنیم تا شاهد fail شدنشان باشیم:



مشاهده شد که تست‌ها fail شدند بدون هرگونه اشتباه syntax.

## مرحله پنجم - پیاده‌سازی کد تا زمان موفق شدن تست‌ها

پیاده‌سازی را با توجه به گفته‌های کلاس کامل می‌کنیم تا تست‌ها pass شوند:

```

from abc import abstractmethod, ABC

class Beverage(ABC):
    description = "generic beverage"

    @classmethod
    def get_description(cls):
        return cls.description

    @abstractmethod
    def cost(self):
        pass

class HouseBlend(Beverage):
    description = "Delicious HouseBlend"

    def cost(self):
        return 0.89

```

```

class Espresso(Beverage):
    description = "Delicious Espresso"

    def cost(self):
        return 1.99

class CondimentDecorator(Beverage, ABC):

    def __init__(self, beverage: Beverage):
        self._beverage = beverage

    def get_description(self):
        return self._beverage.get_description() + " " + self.description

class SteamedMilk(CondimentDecorator):
    description = "with milk"

    def cost(self):
        return self._beverage.cost() + self.added_cost()

    @staticmethod
    def added_cost():
        return 0.1

class Whip(CondimentDecorator):
    description = "with whip"

    def cost(self):
        return self._beverage.cost() + self.added_cost()

    @staticmethod
    def added_cost():
        return 0.1

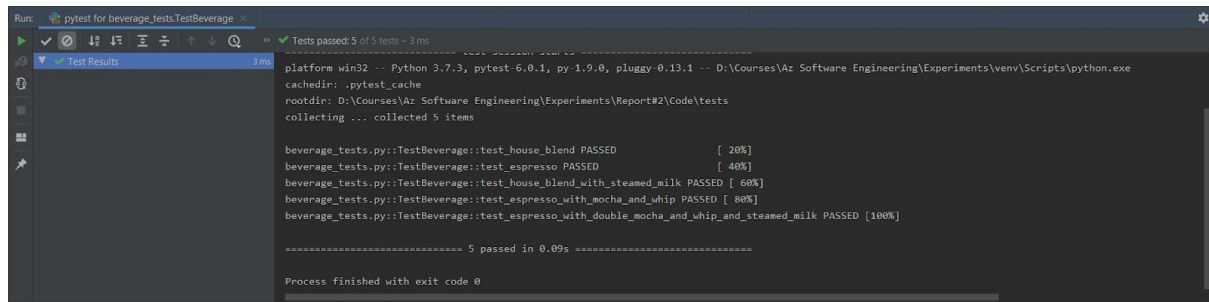
class Mocha(CondimentDecorator):
    description = "with mocha"

    def cost(self):
        return self._beverage.cost() + self.added_cost()

    @staticmethod
    def added_cost():
        return 0.2

```

حال تست‌ها را اجرا می‌کنیم:



```
platform win32 -- Python 3.7.3, pytest-6.0.1, py-1.9.0, pluggy-0.13.1 -- D:\Courses\Az Software Engineering\Experiments\venv\Scripts\python.exe
cachedir: .pytest_cache
rootdir: D:\Courses\Az Software Engineering\Experiments\Report#2\Code\tests
collecting ... collected 5 items

beverage_tests.py::TestBeverage::test_house_blend PASSED [ 20%]
beverage_tests.py::TestBeverage::test_espresso PASSED [ 40%]
beverage_tests.py::TestBeverage::test_house_blend_with_steamed_milk PASSED [ 60%]
beverage_tests.py::TestBeverage::test_espresso_with_mocha_and_whip PASSED [ 80%]
beverage_tests.py::TestBeverage::test_espresso_with_double_mocha_and_whip_and_steamed_milk PASSED [100%]

===== 5 passed in 0.09s =====

Process finished with exit code 0
```

همانطور که مشاهده می‌شود همه تست‌ها با موفقیت pass شدند.

## نکات تکمیلی:

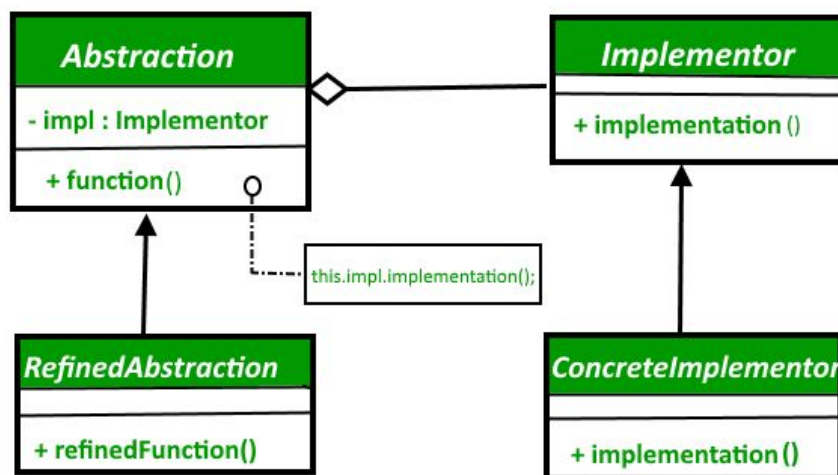
تابع `get_description` به صورت تابع کلاسی (`classmethod`) تعریف شده است زیرا که در هر مرحله کافی است توضیحات (`descriptio`) کلاس را بخواند و نیازی به شی گرفته شده از کلاس ندارد. می‌توانستیم کاری که با `get_description` کردیم را با `cost` هم بکنیم و در نهایت کدی زیباتر داشته باشیم ولی بخاطر تست‌های از قبل نوشته شده و نمودار کلاسی پیشنهاد شده در ویدیو آموزشی کلاس این کار را نکردیم.

## الگوی Bridge

با بکارگیری این الگو لایه `abstraction` از لایه `implementation` جدا می‌شود در نتیجه این دو مستقل از هم می‌توانند تغییر کنند و به علاوه این توانایی را به می‌دهد که شیوه پیاده‌سازی در زمان اجرا به صورت پویا تغییر کند.

برای اضافه کردن رفتار در زیر کلاس‌های `decorator` از تابع `added_cost` استفاده کردیم.

`coverage` تست‌ها در نگاه اول ۹۷٪ است ولی این بخاطر تعریف تابع `abstract` در پایتون است پس در حقیقت `coverage` ما ۱۰۰٪ است.



### نمودار کلاسی الگوی bridge

برای اجرای این مراحل از زبان برنامه‌نویسی python و ابزار pytest استفاده می‌کنیم.

روند کلی این است که برای پیاده‌سازی هر feature جدید در ابتدا تست‌هایی برای آن می‌نویسیم؛ این تست‌ها باید همه شرایط مختلف و مرزی را پوشش دهند و همچنین از طرز کار درست کد با توجه به نیازمندی پروژه مطمئن شوند.

سپس تست‌ها را اجرا می‌کنیم و fail شدنشان را به نظاره می‌نشینیم. سپس با ایجاد interface ها و signature کد خطاهای سینتکسی را از بین می‌بریم.

در گام بعدی حداقل کدی که برای اجرای موفق تست‌ها نیاز است را پیاده‌سازی می‌کنیم تا پس از اجرای تست‌ها همه آن‌ها pass شوند.

در گام آخر نیز کد را refactor می‌کنیم تا در نهایت کدی خوانا و تمیز در اختیار داشته باشیم. همچنین برای آخرین بار تست‌ها را اجرا می‌کنیم تا مطمئن شویم همه چی طبق برنامه پیش رفته است.

### مرحله اول - نوشتن تست‌ها

در این مرحله ۸ تست طراحی کردیم که تمام ۴ حالت ممکن (توان‌های نوع اول و دوم با ضرب‌های نوع اول و دوم) را پوشش دهد و همچنین هر ضرب‌کننده را جداگانه نیز بررسی کند؛ :

```

class TestPowerCalculator:

    def test_normal_multiplication(self):
        multiplier = NormalMultiplication()
        assert multiplier.__class__.__name__ == "NormalMultiplication"
    
```

```

    assert multiplier.calculate(4, 5) == 20

def test_loop_based_multiplication(self):
    multiplier = LoopBasedMultiplication()
    assert multiplier.__class__.__name__ == "LoopBasedMultiplication"
    assert multiplier.calculate(4, 5) == 20

def test_recursive_power_with_normal_multiplication(self):
    multiplier = NormalMultiplication()
    power = RecursivePower(multiplier)
    assert power.multiplication.__class__.__name__ ==
"NormalMultiplication"
    assert power.__class__.__name__ == "RecursivePower"
    assert power.calculate(number=5, power=3) == 125

def test_recursive_power_with_loop_based_multiplication(self):
    multiplier = LoopBasedMultiplication()
    power = RecursivePower(multiplier)
    assert power.multiplication.__class__.__name__ ==
"LoopBasedMultiplication"
    assert power.__class__.__name__ == "RecursivePower"
    assert power.calculate(number=5, power=3) == 125

def test_loop_based_power_with_normal_multiplication(self):
    multiplier = NormalMultiplication()
    power = LoopBasedPower(multiplier)
    assert power.multiplication.__class__.__name__ ==
"NormalMultiplication"
    assert power.__class__.__name__ == "LoopBasedPower"
    assert power.calculate(number=5, power=3) == 125

def test_loop_based_power_with_loop_based_multiplication(self):
    multiplier = LoopBasedMultiplication()
    power = LoopBasedPower(multiplier)
    assert power.multiplication.__class__.__name__ ==
"LoopBasedMultiplication"
    assert power.__class__.__name__ == "LoopBasedPower"
    assert power.calculate(number=5, power=3) == 125

```

```

def test_multiplication_constructor(self):
    try:
        multiplier = Multiplication()
    except TypeError as e:
        assert str(e) == "Can't instantiate abstract class Multiplication with

```



```

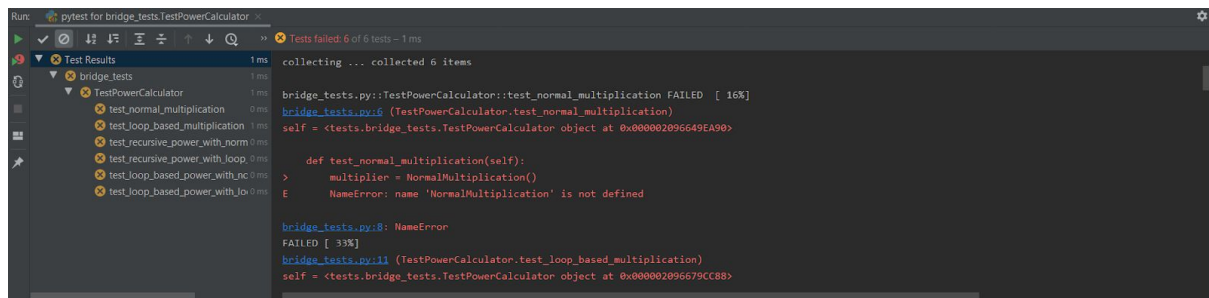
abstract methods calculate"
    else:
        pytest.fail("you should not be able to instantiate abstract class")

def test_power_constructor(self):
    try:
        power = Power(None)
    except TypeError as e:
        assert str(e) == "Can't instantiate abstract class Power with abstract
        methods calculate"
    else:
        pytest.fail("you should not be able to instantiate abstract class")

```

## مرحله دو - شناسایی مشکلات کامپایلی تست‌ها

حال آن‌ها را اجرا کرده و شاهد اشتباهات (به واسطه‌ی پیاده‌سازی نشدن کدها نه اشتباه بودن رفتار کد) آن هستیم:



## مرحله سوم - رفع مشکلات کامپایلی تست‌ها با پیاده‌سازی رابط‌ها

حال به سراغ پیاده‌سازی رابط‌های کد می‌رویم تا تست‌ها صرفاً به واسطه اشتباه بود fail شوند و نه پیاده‌سازی نشدن.

پیاده‌سازی:

```

class Multiplication:

    @abstractmethod
    def calculate(self, first_param, second_param):
        pass

class NormalMultiplication(Multiplication):

    def calculate(self, first_param, second_param):
        pass

class LoopBasedMultiplication(Multiplication):

    def calculate(self, first_param, second_param):

```

```

pass

class Power:
    def __init__(self, multiplication: Multiplication):
        self.multiplication = multiplication

    @abstractmethod
    def calculate(self, number, power):
        pass

class RecursivePower(Power):

    def calculate(self, number, power):
        pass

class LoopBasedPower(Power):

    def calculate(self, number, power):
        pass

```

## مرحله چهارم - اجرای تست‌ها و شناسایی مشکلات زمان اجرا

حال دوباره تست‌ها را اجرا می‌کنیم تا شاهد fail شدنشان باشیم:

```

Run: pytest for bridge_tests.TestPowerCalculator
Tests failed: 6 of 6 tests - 1 ms

Test Results
  bridge_tests
    TestPowerCalculator
      test_normal_multiplication
      test_loop_based_multiplication
      test_recursive_power_with_norm
      test_recursive_power_with_loop
      test_loop_based_power_with_norm
      test_loop_based_power_with_loop

Expected :20
Actual   :None
<Click to see difference>

self = <tests.bridge_tests.TestPowerCalculator object at 0x000025B4A47FA20>

def test_normal_multiplication(self):
    multiplier = NormalMultiplication()
    assert multiplier.__class__.__name__ == "NormalMultiplication"
    > assert multiplier.calculate(4, 5) == 20
E     assert None == 20
bridge_tests.py:18: AssertionError

```

مشاهده شد که تست‌ها fail شدند بدون هرگونه اشتباه syntaxی.

## مرحله پنجم - پیاده‌سازی کد تا زمان موفق شدن تست‌ها

پیاده‌سازی را با توجه به گفته‌های کلاس کامل می‌کنیم تا تست‌ها pass شوند:

```

from abc import ABC, abstractmethod

class Multiplication(ABC):

    @abstractmethod

```

```

    def calculate(self, first_param, second_param):
        pass

class NormalMultiplication(Multiplication):

    def calculate(self, first_param, second_param):
        return first_param * second_param

class LoopBasedMultiplication(Multiplication):

    def calculate(self, first_param, second_param):
        result = 0
        for _ in range(second_param):
            result += first_param
        return result

class Power(ABC):
    def __init__(self, multiplication: Multiplication):
        self.multiplication = multiplication

    @abstractmethod
    def calculate(self, number, power):
        pass

class RecursivePower(Power):

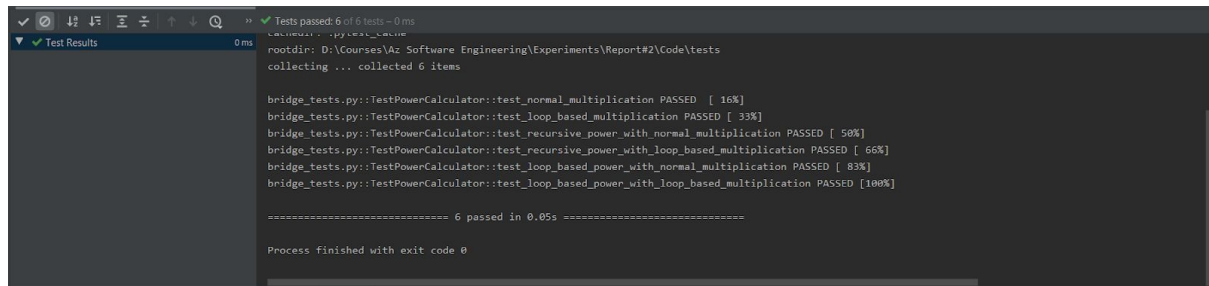
    def calculate(self, number, power):
        if power == 0:
            return 1
        return self.multiplication.calculate(number, self.calculate(number,
power-1))

class LoopBasedPower(Power):

    def calculate(self, number, power):
        result = 1
        for _ in range(power):
            result *= number
        return result

```

حال تست‌ها را اجرا می‌کنیم:



```
✓ Tests passed: 6 of 6 tests - 0 ms
rootdir: D:\Courses\Az Software Engineering\Experiments\Report#2\Code\tests
collecting ... collected 6 items

bridge_tests.py::TestPowerCalculator::test_normal_multiplication PASSED [ 16%]
bridge_tests.py::TestPowerCalculator::test_loop_based_multiplication PASSED [ 33%]
bridge_tests.py::TestPowerCalculator::test_recursive_power_with_normal_multiplication PASSED [ 50%]
bridge_tests.py::TestPowerCalculator::test_recursive_power_with_loop_based_multiplication PASSED [ 66%]
bridge_tests.py::TestPowerCalculator::test_loop_based_power_with_normal_multiplication PASSED [ 83%]
bridge_tests.py::TestPowerCalculator::test_loop_based_power_with_loop_based_multiplication PASSED [100%]

===== 6 passed in 0.05s =====

Process finished with exit code 0
```

همانطور که مشاهده می‌شود همه تست‌ها با موفقیت pass شدند.

## مرحله ششم - Refactor

از آنجایی که از ابتدا اصول برنامه نویسی تمیز رعایت شده بود و همچنین کد نسبتاً ساده بود نیازی به refactor کردن احساس نمی‌شود.

## نکات تکمیلی:

دو نوع توان داریم که یکی به صورت بازگشتی مقدار را محاسبه می‌کند و دیگری با حلقه همین کار را انجام می‌دهد. از طرفی دو نوع ضرب داریم یکی بصورت مستقیم از ضرب پیاده سازی شده در زبان پایتون استفاده می‌کند و دیگری با استفاده از حلقه و جمع کردن پی‌درپی همین کار را انجام می‌دهد. در نگاه اول coverage کد ۹۳٪ است ولی این به دلیل داشتن تابع **abstract** در زبان پایتون است پس در واقع coverage کد ما معادل با ۱۰۰٪ است.