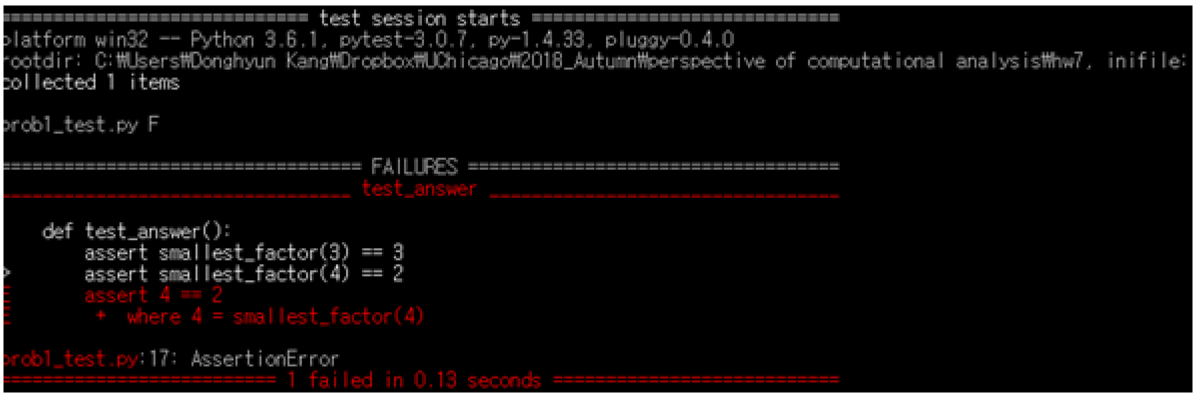
Assignment 7 - Donghyun Kang

1. Unit testing in Python

P1

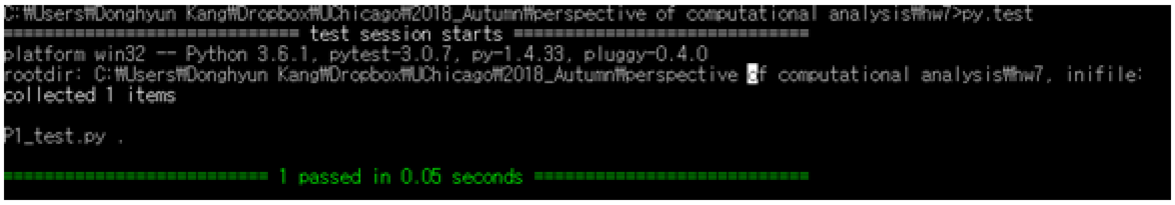


```
============================== test session starts ==============================
latform win32 -- Python 3.6.1, pytest-3.0.7, py-1.4.33, pluggy-0.4.0
ootdir: C:₩Users₩Donghyun Kang₩Dropbox₩UChicago₩2018_Autumn₩perspective of computational analysis₩hw7, inifile:
ollected 1 items

rob1_test.py F

=================================== FAILURES ===================================
_________________________________ test_answer __________________________________

    def test_answer():
        assert smallest_factor(3) == 3
>       assert smallest_factor(4) == 2
E       assert 4 == 2
E        +  where 4 = smallest_factor(4)

rob1_test.py:17: AssertionError
========================== 1 failed in 0.13 seconds ============================
```

- I found that numbers which can be factored into square of a prime number such as 4 and 9 caused the problem. This was solved by adding 1 to the latter when the range was defined. After fixing this part, my code passed the unit test as following. See my code in the repository P1_test.py.

- The following displays the modified function and the test code, with no errors.

```
    def smallest_factor(n)
    """Return the smallest prime factor of the positive integer n."""
        if n == 1: return 1
        for i in range(2, int(n**.5) + 1):
            if n % i == 0: return i
        return n

    def test_smallest_factor():
        assert smallest_factor(3) == 3
        assert smallest_factor(4) == 2
        assert smallest_factor(5) == 5
        assert smallest_factor(6) == 2
        assert smallest_factor(7) == 7
        assert smallest_factor(8) == 2
        assert smallest_factor(9) == 3
        assert smallest_factor(10) == 2
```

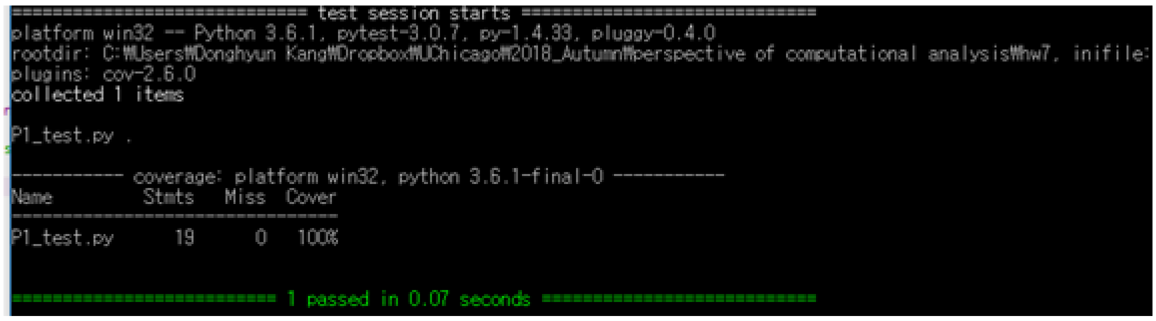


```
C:₩Users₩Donghyun Kang₩Dropbox₩UChicago₩2018_Autumn₩perspective of computational analysis₩hw7>py.test
============================== test session starts ==============================
platform win32 -- Python 3.6.1, pytest-3.0.7, py-1.4.33, pluggy-0.4.0
rootdir: C:₩Users₩Donghyun Kang₩Dropbox₩UChicago₩2018_Autumn₩perspective of computational analysis₩hw7, inifile:
collected 1 items

P1_test.py .

========================== 1 passed in 0.05 seconds ===========================
```

P2

- The following captures of the code chunk and the terminal output show that the function get the complete coverage with additional test cases., with the 100% coverage rate.

```python
def smallest_factor(n)
"""Return the smallest prime factor of the positive integer n."""
    if n == 1: return 1
    for i in range(2, int(n**.5) + 1):
        if n % i == 0: return i
    return n

def test_smallest_factor():
    assert smallest_factor(3) == 3
    assert smallest_factor(4) == 2
    assert smallest_factor(5) == 5
    assert smallest_factor(6) == 2
    assert smallest_factor(7) == 7
    assert smallest_factor(8) == 2
    assert smallest_factor(9) == 3
    assert smallest_factor(10) == 2
    assert smallest_factor(11) == 11
    assert smallest_factor(12) == 2
    assert smallest_factor(13) == 13
    assert smallest_factor(100) == 2
```

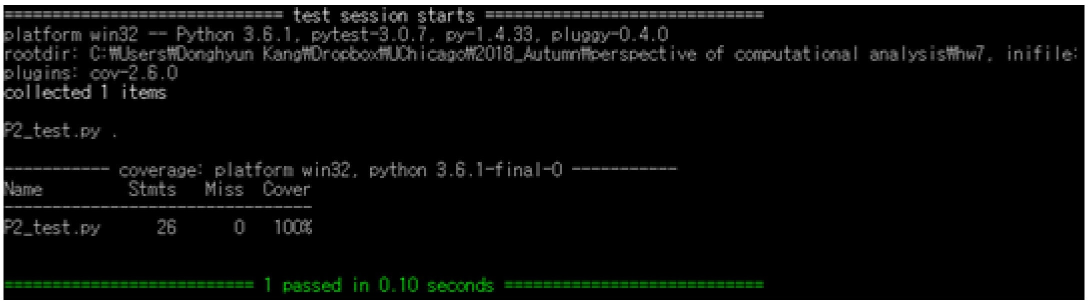


```
============================== test session starts ==============================
platform win32 -- Python 3.6.1, pytest-3.0.7, py-1.4.33, pluggy-0.4.0
rootdir: C:₩Users₩Donghyun Kang₩Dropbox₩UChicago₩2018_Autumn₩perspective of computational analysis₩hw7, inifile:
plugins: cov-2.6.0
collected 1 items

P1_test.py .

----------- coverage: platform win32, python 3.6.1-final-0 -----------
Name            Stmts   Miss  Cover
-----------------------------------
P1_test.py         19      0   100%

========================== 1 passed in 0.07 seconds ===========================
```

- Then, the below code chunk shows a comprehensive unit test for the function “month_length.”

```python
def month_length(month, leap_year=False):
    """Return the number of days in the given month."""
    if month in {"September", "April", "June", "November"}:
        return 30
    elif month in {"January", "March", "May", "July",
                   "August", "October", "December"}:
        return 31
    if month == "February":
        if not leap_year:
            return 28
        else:
            return 29
    else:
        return None

def test_month_length():
    assert month_length("January") == 31
    assert month_length("February", leap_year = False) == 28
    assert month_length("February", leap_year = True) == 29
    assert month_length("March") == 31
    assert month_length("April") == 30
    assert month_length("May") == 31
    assert month_length("June") == 30
    assert month_length("July") == 31
    assert month_length("August") == 31
    assert month_length("September") == 30
    assert month_length("October") == 31
    assert month_length("November") == 30
    assert month_length("December") == 31
    assert month_length("whatever") == None
```

- The testing result is as below, which shows the 100% coverage rate.



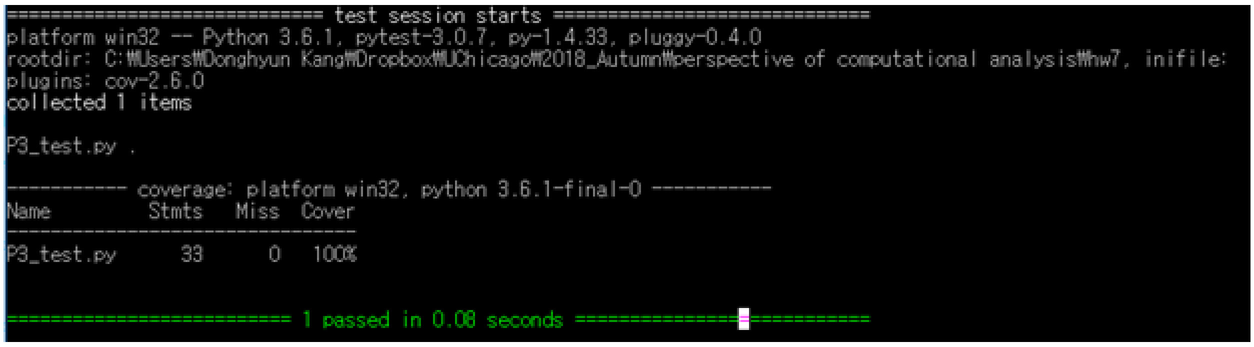
```
============================== test session starts ==============================
platform win32 -- Python 3.6.1, pytest-3.0.7, py-1.4.33, pluggy-0.4.0
rootdir: C:₩Users₩Donghyun Kang₩Dropbox₩UChicago₩2018_Autumn₩perspective of computational analysis₩hw7, inifile:
plugins: cov-2.6.0
collected 1 items

P2_test.py .

----------- coverage: platform win32, python 3.6.1-final-0 -----------
Name         Stmts   Miss  Cover
--------------------------------
P2_test.py      26      0   100%


========================== 1 passed in 0.10 seconds ===========================
```

P3

- My code is as below.

```
    import pytest

    def operate(a, b, oper):
        """Apply an arithmetic operation to a and b."""
        if type(oper) is not str:
            raise TypeError("oper must be a string")
        elif oper == '+':
            return a + b
        elif oper == '-':
            return a - b
        elif oper == '*':
            return a * b
        elif oper == '/':
            if b == 0:
                raise ZeroDivisionError("division by zero is undefined")
            return a / b
        raise ValueError("oper must be one of '+', '/', '-', or '*'")
    def test_operate():
        with pytest.raises(TypeError) as typeinfo:
            operate(1, 1, 2)
        assert typeinfo.value.args[0] == "oper must be a string"

        with pytest.raises(TypeError) as typeinfo:
            operate(1, 1, [1,2])
        assert typeinfo.value.args[0] == "oper must be a string"

        assert operate(4, 3, "+") == 7
        assert operate(4, 3, "-") == 1
        assert operate(4, 3, "*") == 12
        assert operate(12, 3, "/") == 4

        with pytest.raises(ZeroDivisionError) as zero:
            operate(12, 0, "/") == 4
        assert zero.value.args[0] == "division by zero is undefined"

        with pytest.raises(ValueError) as v_err:
            operate(1, 2, "a")
        assert v_err.value.args[0] == "oper must be one of '+', '/', '-', or '*'"
```

- The test result is as below, with the 100% coverage rate.



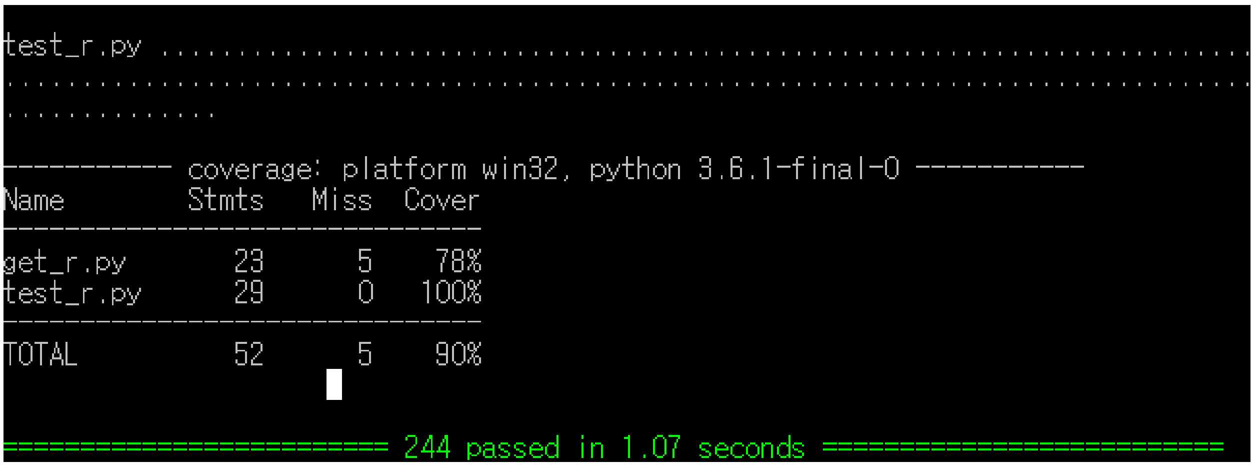
```
============================== test session starts ==============================
platform win32 -- Python 3.6.1, pytest-3.0.7, py-1.4.33, pluggy-0.4.0
rootdir: C:\Users\Donghyun Kang\Dropbox\UChicago\2018_Autumn\perspective of computational analysis\hw7, inifile:
plugins: cov-2.6.0
collected 1 items

P3_test.py .

----------- coverage: platform win32, python 3.6.1-final-0 -----------
Name           Stmts   Miss  Cover
----------------------------------
P3_test.py        33      0   100%


========================== 1 passed in 0.08 seconds ===========================
```

2. Test driven Development

- The below is my get_r code.

```
import numpy as np
def get_r(K, L, alpha, Z, delta):
    '''
    This function generates the interest rate or vector of interest rates
    '''
    #raise value error for alpha, Z, delta
    if alpha <= 0 or alpha >= 1:
        raise ValueError("alpha has to be within (0,1)")
    if Z <= 0:
        raise ValueError("Z has to be greater than 0")
    if delta < 0:
        raise ValueError("delta has to be within (0,1)")
    if np.isscalar(K) and np.isscalar(L):
        if K <= 0 or L <= 0:
            raise ValueError("K and L have to be greater than 0")
        else :
            r = alpha * Z * ((L/K))**(1-alpha) - delta
            return r
    if not np.isscalar(K) and not np.isscalar(L):
        K = np.array(K)
        L = np.array(L)
        if len(K) == len(L):
            assert("the length of two input should be the same")
        #check negative values
        if len(K[K <= 0]) >= 1 or len(L[L <= 0]) >= 1:
            raise ValueError("K and L have to be greater than 0")
        else:
            r = alpha * Z * ((L/K))**(1-alpha) - delta
            return r
```

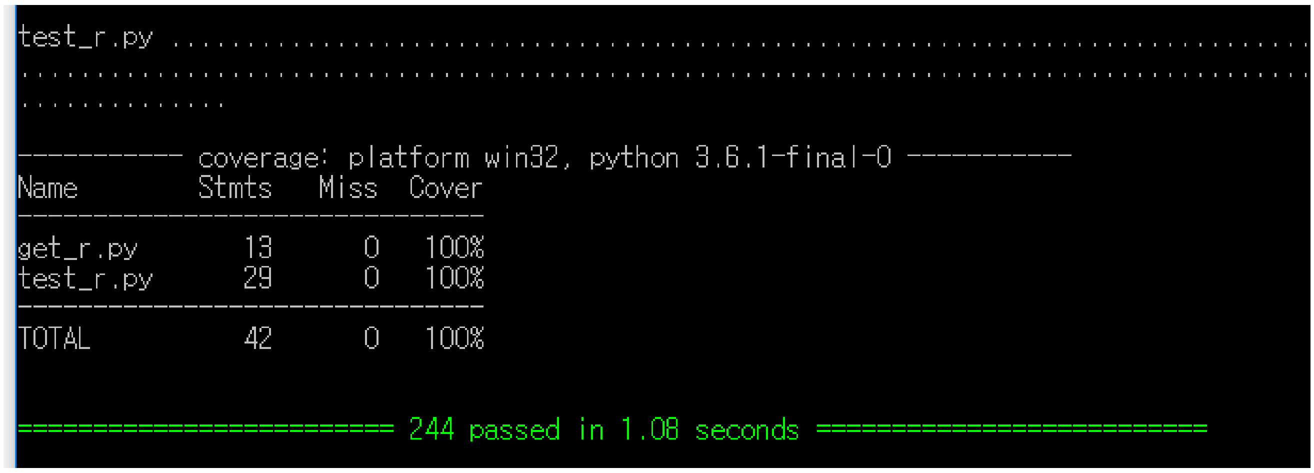
- The result of test code is attached as following.



```
test_r.py ..........................................................................
..........................................................................................
..............

----------- coverage: platform win32, python 3.6.1-final-0 -----------
Name          Stmts   Miss  Cover
---------------------------------
get_r.py         23      5    78%
test_r.py        29      0   100%
---------------------------------
TOTAL            52      5    90%


======================= 244 passed in 1.07 seconds ========================
```

  - The five missing in *get_r.py* occurred because the test code does not have illegitimate cases for alpha, Z, delta, K, and L.
- If I created my *get_r.py* as below, I could get 100% coverage for *get_r.py* as well, but I think the previous code is better in a sense that it can catch illegitimate inputs.

```
import numpy as np
def get_r(K, L, alpha, Z, delta):
    '''
    This function generates the interest rate or vector of interest rates
    '''
    #raise value error for alpha, Z, delta
    if np.isscalar(K) and np.isscalar(L):
        r = alpha * Z * ((L/K))**(1-alpha) - delta
        return r
    if not np.isscalar(K) and not np.isscalar(L):
        K = np.array(K)
        L = np.array(L)
        if len(K) == len(L):
            assert("the length of two input should be the same")
        r = alpha * Z * ((L/K))**(1-alpha) - delta
        return r
```



```
test_r.py ...............................................................................
.................................................................................
.............

----------- coverage: platform win32, python 3.6.1-final-0 -----------
Name          Stmts   Miss  Cover
---------------------------------
get_r.py         13      0   100%
test_r.py        29      0   100%
---------------------------------
TOTAL            42      0   100%


========================= 244 passed in 1.08 seconds ==========================
```

3. Watts (2014)

(a) When initially introduced in the 1960s, rational choice theory imposed a framework of theoretical assumptions that fit with or” rationalized" observed behavior. What were some of the criticisms of this approach?

- The Rational Choice Theory (RCT, hereafter) was originated from economics with the assumption of individuals as a utility maximizer and it was widely adopted (and adapted) by other social science disciplines including sociology and political science. This was partly because RCT seemed to provide a framework for scientific endeavor in a sense that it helped researchers to formulate testable statements guided by the principle of parsimony. (Watts 2014, pp. 319-320) However, the hard-core RCTs were first criticized because of their lacks of empirical supports and thereby the somewhat unrealistic assumptions of rational actors were also attacked, especially by sociologists and psychologists. (ibid., p. 320) Watts argued that this led RCT scholars not only to expand the scope of rational actions but also to evaluate their theories in terms of understandability, which had not been the criteria for scientific explanations. Watts labeled this framework as “rationalizable action”, defining that the theories drawing upon this basically posit that “individual or collective action can be explained in terms of the intentions, beliefs, circumstances, and opportunities of the actors involved.” (ibid., p. 316) Watts criticized that this framework relying on empathetic reasoning cannot be distinguished from the folk-commonsense theory based on our daily experiences and thus does not guarantee scientific explanations of social worlds.

(b) What is the main pitfall that Watts sees in using commonsense theories of action? [Hint: A good explanation of the answer is in the last half of the section entitled, “Theorizing by Mental Simulation".] The answer to this question precedes its decomposition into three parts in the Section, “Three Problems with Rationalizable Action as Causal Explanation."

- Since human beings are given the ability to do mental simulations to infer the reasons behind actions conducted by other people, social scientists also employ mental simulations based on everyday life experiences when they develop social theories. (ibid., pp. 325-326) But the major problem of this is that it blurs the distinction between the “modes of prediction and sense making.” (ibid., p. 327) Watts argued that this conflation might not be seriously problematic in everyday life in a sense that the immediate feedback could correct some errors or the conflation does not cause serious problems but the theories based on commonsense are not sufficient to build valid scientific explanations of social worlds that can be universally applied. (ibid., pp. 327-328)

(c) What is Watts' proposed solution to the issues with rational choice modeling and causal explanation?

- Watts basically proposed to broaden/adjust/re-conceptualize the definition of “prediction.” He first discussed that the valid scientific causal explanations qualify “Woodward’s manipulationist criterion that explanations must answer a what-if-it-had-been-different question.” (ibid, p. 335) To meet this, Watts suggested that sociologists (or social scientists in general) in theory should employ experiments and counterfactual causal inference techniques when it comes to research. However, admitting that these strategies might not be suitable to all social science enterprises, Watts argued that we can engage in building rigorous social scientific theories by focusing on the predictive power of theories. (ibid., p. 337) And he further laid out that prediction refers to “in a broad sense of out-of-sample testing, allowing both for probabilistic prediction and for predictions about stylized facts or patterns of outcomes.” (ibid., p. 340) And this should be distinguished from mere “causal stories” which often provide us satisfactory understandability.

(d) Although this paper does a good job or relating causality to prediction, I don't like its disdain for theory that specifically outlines the assumptions and mechanisms of process being modeled. Write a short addendum to the paper about how theoretical models with their necessary simplifications and their specific assumptions about mechanisms - could benefit causal inference and prediction.

- One possible way to advocate the importance of theory in making a better predictive model is to acknowledge that theories can guide researchers or modelers to select the right sets of variables when they attempt to develop models. This might be justified with the principle of parsimony which is often invoked when it comes to evaluating a model. Furthermore, I think there is also a good chance that focusing only on the predictive power might generate models that do not have anything to do with scientific inquiries, even though this might be what Watts intended when he attempted the definition of prediction. This is because that researchers might be able to (in theory) build a model well predict outcomes in what Watt’s proposed, lacking further values in terms of increasing our actual scientific understanding of social worlds.