Q1: Unit Testing in Python

Problem1:

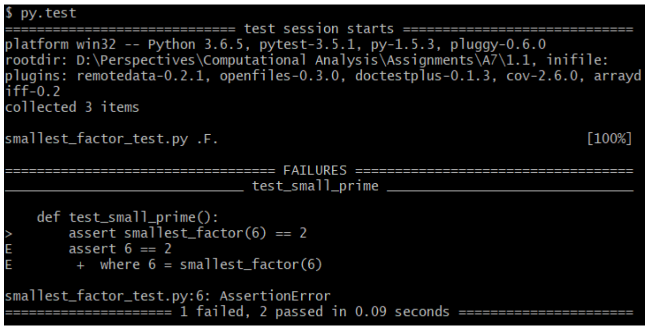
Define the smallest_factor function

```python
# smallest_factor.py

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)):
        if n % i == 0: return i
    return n
```

Define the unittest functions

```python
# smallest_factor_test

from smallest_factor import smallest_factor

def test_one():
    assert smallest_factor(1) == 1, "failed on 1"
def test_small_prime():
    assert smallest_factor(6) == 2, "failed on 6"
def test_big_prime():
    assert smallest_factor(11) == 11, "failed on 11"
```

py.test



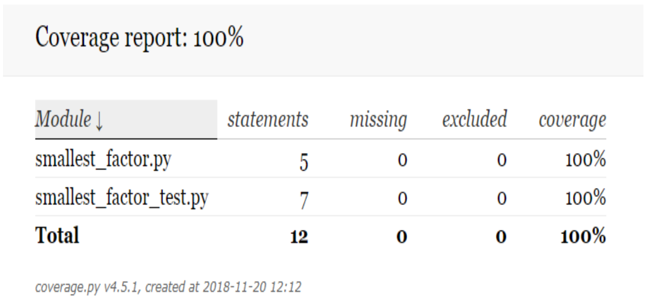
```
$ py.test
============================ test session starts ============================
platform win32 -- Python 3.6.5, pytest-3.5.1, py-1.5.3, pluggy-0.6.0
rootdir: D:\Perspectives\Computational Analysis\Assignments\A7\1.1, inifile:
plugins: remotedata-0.2.1, openfiles-0.3.0, doctestplus-0.1.3, cov-2.6.0, arrayd
iff-0.2
collected 3 items

smallest_factor_test.py .F.                                            [100%]

================================= FAILURES ==================================
______________________________ test_small_prime _____________________________

    def test_small_prime():
>       assert smallest_factor(6) == 2
E       assert 6 == 2
E        +  where 6 = smallest_factor(6)

smallest_factor_test.py:6: AssertionError
==================== 1 failed, 2 passed in 0.09 seconds =====================
```

Corrected function

```python
def smallest_factor_corrected(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
```

Problem2:

check the coverage for problem 1



Coverage report: 100%

Module ↓	statements	missing	excluded	coverage
smallest_factor.py	5	0	0	100%
smallest_factor_test.py	7	0	0	100%
Total	**12**	**0**	**0**	**100%**

coverage.py v4.5.1, created at 2018-11-20 12:12

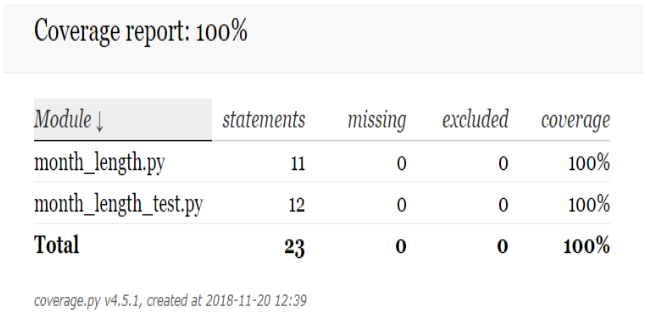
As reported, my unittests for problem1 are already comprehensive.

```python
# month_length.py

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
```

Unittests

```python
# month_length_test.py

from month_length import month_length

def test_30days():
    assert month_length('June')==30, "failed on June"

def test_31days():
    assert month_length('October')==31, "failed on October"

def test_leapFeb():
    assert month_length('February', leap_year=True)==29, \
    "failed on February in leap years"

def test_nonleapFeb():
    assert month_length('February', leap_year=False)==28, \
    "failed on February in non-leap years"

def test_valueError():
    assert month_length('Homework')==None, "failed on wrong inputs"
```

As reported, these tests have a comprehensive coverage


Coverage report: 100%

Module ↓	statements	missing	excluded	coverage
month_length.py	11	0	0	100%
month_length_test.py	12	0	0	100%
Total	**23**	**0**	**0**	**100%**

coverage.py v4.5.1, created at 2018-11-20 12:39

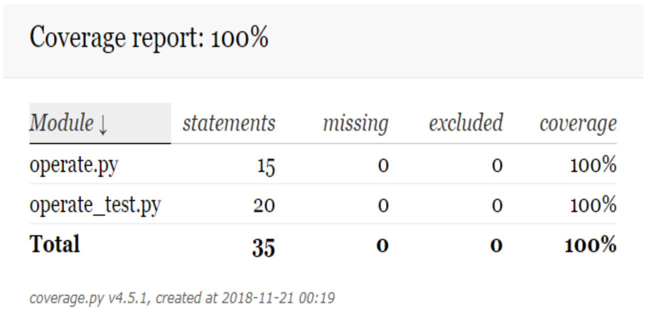
Problem3

```python
# opertate.py

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 '*'")
```

unittests

```python
# operate_test.py

from operate import operate
import pytest

def test_str():
    with pytest.raises(TypeError) as e:
        operate(1, 2, 3)
    assert e.value.args[0] == "oper must be a string"

def test_add_minus_times():
    assert operate(3, 4, "+")==7
    assert operate(11, 7, "-")==4
    assert operate(4, 5, "*")==20

def test_minus():
    assert operate(36, 4, "/")==9
    with pytest.raises(ZeroDivisionError) as e:
        operate(36, 0, "/")
    assert e.value.args[0]=="division by zero is undefined"

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

As reported, all lines have been tested, including the exceptions.


Coverage report: 100%

Module ↓	statements	missing	excluded	coverage
operate.py	15	0	0	100%
operate_test.py	20	0	0	100%
Total	**35**	**0**	**0**	**100%**

coverage.py v4.5.1, created at 2018-11-21 00:19

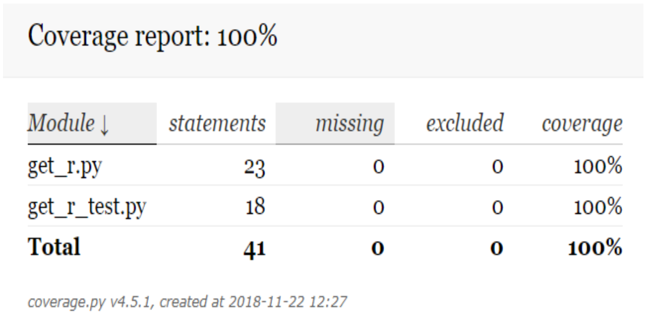
Q2: Test Driven Development

```python
# get_r.py

import numpy as np
from numbers import Number

def get_r(K, L, alpha, Z, delta):
    '''
    This function generates the interest rate or vector of interest rates
    '''
    if not (isinstance(alpha, Number) and isinstance(delta, Number) \
    and isinstance(Z, Number)):
        raise TypeError("alpha, delta and Z must be Numbers")
    if not ((0<alpha<1) and (0<delta<1) and (Z>0)):
        raise ValueError("alpha and delta must be in the range of (0,1); \
Z must be a positive Number")
    if np.isscalar(K): K = [K]
    if np.isscalar(L): L = [L]
    if len(K)!=len(L):
        raise ValueError("K and L must be of the same length")
    try:
        K_array = np.array(K, dtype=float)
        L_array = np.array(L, dtype=float)
    except ValueError:
        raise TypeError("K and L should contain only Number objects")
    if not (all(K_array>0) and all(L_array>0)):
        raise ValueError("K, L and Z must be in be positive")

    r = (alpha * Z * ((L_array/K_array)**(1-alpha)) - delta).tolist()
    if len(r)==1:
        return r[0]
    else:
        return r
```

Here I especially set testing conditions for the following errors: (1) TypeError, where K and L might not be numeric vectors or scalars, alpha/Z/delta might not be numeric. (2) TypeError, where numeric values do not fall in their theoretical range and violates economic rules. (3) K and L might be of different length.

```python
# get_r_test.py

from get_r import get_r
import pytest

def test_type():
    pytest.raises(TypeError, get_r, K=1, L=1, alpha='A', Z=2, delta=0.1)
    pytest.raises(TypeError, get_r, K=['a',2], L=[2,3], alpha=0.4, Z=2, delta=0.1)

def test_value():
    pytest.raises(ValueError, get_r, K=1, L=1, alpha=-0.1, Z=2, delta=1.2)
    pytest.raises(ValueError, get_r, K=[-1,2], L=[2,3], alpha=0.4, Z=2, delta=0.1)

def test_length():
    pytest.raises(ValueError, get_r, K=[1,2], L=[2,3,4], alpha=0.4, Z=2, delta=0.1)

def test_normal():
    def round_to_n(ndigits):
        def inner(num):
            return round(num, ndigits)
        return inner

    assert list(map(round_to_n(ndigits=4), get_r([1,2,3], [3,7,11], 0.4, 2, 0.1))) \
    ==[1.4465, 1.5964, 1.6444]
    assert round(get_r(3, 2, 0.4, 2, 0.1),3)==0.527
```

As reported, these tests have a full coverage over the get_r function.



Coverage report: 100%

Module ↓	statements	missing	excluded	coverage
get_r.py	23	0	0	100%
get_r_test.py	18	0	0	100%
Total	**41**	**0**	**0**	**100%**

coverage.py v4.5.1, created at 2018-11-22 12:27

Q3: Watts, 2014

As stated in the paper, the Rational Choice Theory is a striking case where sociologists fail to realize the basis on common sense. Its validity was attacked for several reasons: the assumptions of "actors' preference, knowledge and computational capability" are implausible, or in other words, not predictive of people's real-life actions. (Watts, 2014, p.325)

One common unrecognized mistake that sociologists make is that they are implicitly conflating commonsense theory into their sociological arguments without checking for external validity. What is perceived true on everyday basis is not entirely valid when we extend to a more universal context. Though not explicitly articulated, theories of action of the kind equalized "understandability with causality" to some extent, and what makes sense of "observed outcome" are often unsuccessful in revealing general "causal mechanisms". Therefore, sociologists, now and then, make unreliable predictions based on ex ante evidence, and induce false causality based on ex post observations. (Watts, 2014, p.325)

The authors argue that we should depart from making causal inferences from "understanding in an empathetic sense", and instead draw upon more scientifically rigorous methodologies to achieve valid answers. One most welcome approach is field experiments, which generate real-life randomness and help to identify causation. Other "weaker but useful" experimental methodology include: natural experiment and quasi experiment, and lab experiments could also be a powerful tool under certain circumstances; besides, thanks to advanced statistical methods, social scientists can model on large-scale unexperimental datasets to make solid causal inference under the counterfactual framework; finally, they can also simply assess explanations on how well they make predictions. (Watts, 2014, pp.335-337)

This paper drastically criticizes the fact that sociologists have long been implicitly relying their explanations on commonsense, and so naive it is to conclude that understandability will ensure validity. I totally agree with the statement, but we shall not reject formulating theories to help answer questions. (1) "All models are wrong, but some are useful." Theoretical models may not be completely correct in regards to its prediction capability, but they can provide guidelines on how to analyze a complicated question. They can lend insights that facilitates further large data analysis. (2) Internally valid theoretical models are consistent with their assumptions, we can improve models by relaxing or adjusting these assumptions whenever we find countering statistical evidence. (3) Understandability is not a falsity, but rather a weapon that empowers theoretical models to evolve over time. It excites communication among peer faculties and enriches the set of thoughts and methodologies associated with social sciences. What we need to do is be cautious with simplified theories, rather than fleed away from them.

Reference

Watts, Duncan J., Common Sense and Sociological Explanations," *American Journal of Sociology*, September 2014, 120 (2), 313{351.