Assignment 7
Di Tong

1. Unit Testing in Python (All codes stored in file no1)

Problem 1 (All codes stored in file no1/p1):

Here is the function that contains a subtle but important error (file no1/p1/smallestfactor.py). The problem lies at the upper boundary of range: it does not include the square root of n in the iteration.

```
#smallestfactor.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
```

Here is a unit test I wrote for this function (file no1/p1/test_smallestfactor.py). I choose a number whose smallest prime factor is its square root to detect the error of the function.

```
#test_smallestfactor.py

import smallestfactor

def test_smallestfactor():
    assert smallestfactor.smallest_factor(121) == 11, "incorrect result"
```

Here is the test result of my unit test:

```
student@cs-vm:~/capp30121-aut-18-ditong1996/lab1$ py.test
============================= test session starts ==============================
platform linux -- Python 3.5.2, pytest-3.3.1, py-1.5.2, pluggy-0.6.0
metadata: {'Plugins': {'html': '1.16.0', 'metadata': '1.5.1', 'json': '0.4.0'},
'Platform': 'Linux-4.4.0-135-generic-x86_64-with-Ubuntu-16.04-xenial', 'Packages
': {'py': '1.5.2', 'pytest': '3.3.1', 'pluggy': '0.6.0'}, 'Python': '3.5.2'}
rootdir: /home/student/capp30121-aut-18-ditong1996/lab1, inifile:
plugins: metadata-1.5.1, json-0.4.0, html-1.16.0
collected 1 item

test_smallestfactor.py F                                                  [100%]

=================================== FAILURES ===================================
_____________________________ test_smallestfactor ______________________________

    def test_smallestfactor():
>       assert smallestfactor.smallest_factor(121) == 11, "incorrect result"
E       AssertionError: incorrect result
E       assert 121 == 11
E        +  where 121 = <function smallest_factor at 0x7fbbae287378>(121)
E        +    where <function smallest_factor at 0x7fbbae287378> = smallestfacto
r.smallest_factor

test_smallestfactor.py:6: AssertionError
=========================== 1 failed in 2.95 seconds ===========================
```

Here is the corrected version of the initial function (file no1/p1/correct.py). By adding one to the right boundary of range, I include the square root of n in the iteration.

Assignment 7
Di Tong

```
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
```

Here is the test result for the correct.py file:

```
student@cs-vm:~/capp30121-aut-18-ditong1996/lab1$ py.test
============================= test session starts ==============================
platform linux -- Python 3.5.2, pytest-3.3.1, py-1.5.2, pluggy-0.6.0
metadata: {'Packages': {'pytest': '3.3.1', 'py': '1.5.2', 'pluggy': '0.6.0'}, 'P
lugins': {'metadata': '1.5.1', 'html': '1.16.0', 'json': '0.4.0'}, 'Platform': '
Linux-4.4.0-135-generic-x86_64-with-Ubuntu-16.04-xenial', 'Python': '3.5.2'}
rootdir: /home/student/capp30121-aut-18-ditong1996/lab1, inifile:
plugins: metadata-1.5.1, json-0.4.0, html-1.16.0
collected 1 item

test_smallestfactor.py .                                                 [100%]

=========================== 1 passed in 0.53 seconds ===========================
```

Problem 2 (All codes stored in file no1/p2):

I added two test cases for the codes (if n == 1: return 1) and (return n) to the test file for the smallest_factor function to get complete coverage (file no1/p2/test_smallestfactor.py):

```
#test_smallestfactor.py

import smallestfactor

def test_smallestfactor():
    assert smallestfactor.smallest_factor(121) == 11, "incorrect result"
    assert smallestfactor.smallest_factor(1) == 1, "incorrect result"
    assert smallestfactor.smallest_factor(7) == 7, "incorrect result"
```

Here are the test results:

```
============================= test session starts ==============================
platform darwin -- Python 3.6.6, pytest-4.0.0, py-1.7.0, pluggy-0.8.0
rootdir: /Users/ellenhsieh/Desktop/no1/p1, inifile:
plugins: remotedata-0.3.1, openfiles-0.3.0, cov-2.6.0
collected 1 item

test_smallestfactor.py .                                                 [100%]

=========================== 1 passed in 0.06 seconds ===========================
```

```
============================= test session starts ==============================
platform darwin -- Python 3.6.6, pytest-4.0.0, py-1.7.0, pluggy-0.8.0
rootdir: /Users/ellenhsieh/Desktop/no1/p1, inifile:
plugins: remotedata-0.3.1, openfiles-0.3.0, cov-2.6.0
collected 1 item

test_smallestfactor.py .                                                 [100%]

---------- coverage: platform darwin, python 3.6.6-final-0 -----------
Name                     Stmts   Miss  Cover
--------------------------------------------
smallestfactor.py            5      0   100%
test_smallestfactor.py       5      0   100%
--------------------------------------------
TOTAL                       10      0   100%


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

Assignment 7
Di Tong

Here is the month_length function in problem 2 (file no1/p2/monthlength.py):

```python
#monthlength.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
```

Here is a comprehensive unit test I wrote for the month_length function (file no1/p2/test_monthlength.py). I test each line of codes in the function in their original order.

```python
#test_monthlength.py

import monthlength

def test_monthlength():
    assert monthlength.month_length("September", leap_year=False) == 30, "incorrect result"
    assert monthlength.month_length("March", leap_year=False) == 31, "incorrect result"
    assert monthlength.month_length("February", leap_year=False) == 28, "incorrect result"
    assert monthlength.month_length("February", leap_year=True) == 29, "incorrect result"
    assert monthlength.month_length("Else", leap_year=False) == None, "incorrect result"
```

Here are the test results:

```
============================= test session starts ==============================
platform darwin -- Python 3.6.6, pytest-4.0.0, py-1.7.0, pluggy-0.8.0
rootdir: /Users/ellenhsieh/Desktop/no1/p2, inifile:
plugins: remotedata-0.3.1, openfiles-0.3.0, cov-2.6.0
collected 1 item

test_monthlength.py .                                                    [100%]

=========================== 1 passed in 0.02 seconds ===========================
============================= test session starts ==============================
platform darwin -- Python 3.6.6, pytest-4.0.0, py-1.7.0, pluggy-0.8.0
rootdir: /Users/ellenhsieh/Desktop/no1/p2, inifile:
plugins: remotedata-0.3.1, openfiles-0.3.0, cov-2.6.0
collected 1 item

test_monthlength.py .                                                    [100%]

---------- coverage: platform darwin, python 3.6.6-final-0 -----------
Name                  Stmts   Miss  Cover
-----------------------------------------
monthlength.py           10      0   100%
test_monthlength.py       7      0   100%
-----------------------------------------
TOTAL                    17      0   100%


=========================== 1 passed in 0.04 seconds ===========================
```

Assignment 7
Di Tong

Problem 3 (All codes stored in file no1/p3):

Here is the operate function in problem 3 (file no1/p3/operate.py):

```python
#operate.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 '*'")
```

Here is a comprehensive unit test I wrote for the operate function (file no1/p3/test_operate.py). I first test the four kinds of operations (+, -, *, /) one by one and then test for the zero division error, type error and value error, respectively.

```python
#test_operate.py
import pytest
import operate

def test_operate():
    assert operate.operate(1, 2, '+') == 3, "incorrect result"
    assert operate.operate(3, 2, '-') == 1, "incorrect result"
    assert operate.operate(3, 2, '*') == 6, "incorrect result"
    assert operate.operate(4, 2, '/') == 2, "incorrect result"
    with pytest.raises(ZeroDivisionError) as excinfo:
        operate.operate(4, 0, '/')
    assert excinfo.value.args[0] == "division by zero is undefined"
    with pytest.raises(TypeError) as excinfo:
        operate.operate(4, 0, 3)
    assert excinfo.value.args[0] == "oper must be a string"
    with pytest.raises(ValueError) as excinfo:
        operate.operate(4, 0, '%')
    assert excinfo.value.args[0] == "oper must be one of '+', '/', '-', or '*'"
```

Here are the test results:

```
============================== test session starts ===============================
platform darwin -- Python 3.6.6, pytest-4.0.0, py-1.7.0, pluggy-0.8.0
rootdir: /Users/ellenhsieh/Desktop/no1/p3, inifile:
plugins: remotedata-0.3.1, openfiles-0.3.0, cov-2.6.0
collected 1 item

test_operate.py .                                                          [100%]

---------- coverage: platform darwin, python 3.6.6-final-0 -----------
Name                Stmts   Miss  Cover
---------------------------------------
operate.py             14      0   100%
test_operate.py        16      0   100%
---------------------------------------
TOTAL                  30      0   100%


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

2. Test driven development (All codes stored in file no2)

Here is the content of my python module get r.py (file no2/get r.py) :

```python
def get_r(K, L, alpha, Z, delta):
    '''
    This function generates the interest rate or vector of interest rates
    '''
    r = alpha * Z * ((L / K)**(1 - alpha)) - delta
    return r
```

Here are the pytest test results:

```
============================= test session starts ==============================
platform darwin -- Python 3.6.6, pytest-4.0.0, py-1.7.0, pluggy-0.8.0
rootdir: /Users/ellenhsieh/Desktop/no2, inifile:
plugins: remotedata-0.3.1, openfiles-0.3.0, cov-2.6.0
collected 244 items

test_r.py ............................................................. [ 25%]
....................................................................... [ 54%]
....................................................................... [ 84%]
......................................                                  [100%]

========================= 244 passed in 1.86 seconds ==========================
```

3. Watts (2014)

When rational choice theory was initially introduced in the 1960s, some criticisms of this approach invalidated it through pointing out its "implausible or empirically invalid assumptions about the preferences, knowledge, and computational capabilities of the actors in question." (Watts 2014, p.320) Some other criticisms questioned this approach by presenting empirical evidence that contradicted predictions based on this approach.

According to Watts (2014), the main pitfall in using commonsense theories of action is that it conflates understandability and causality and hence, lacks the scientific validity that enables theories to be applied universally. Understandability and causality are not effectively interchangeable. In other words, the explanations built on commonsense do not necessarily have the ability to causally account for the observations they derived from, let alone acting as more generalizable causal mechanisms.

Assignment 7
Di Tong

Watts (2014) proposed three partial solutions to the issues with rational choice modeling and causal explanation, all of which are based on Woodward's manipulationist criterion: "explanations must answer a what-if-it-had-been-different question—and then proceed to lay out different but related standards of evidence for such claims to be taken seriously." (p. 335) The first and most straightforward solution is to increase the application of experimental methods, including field experiments, natural experiments, quasi-experiments and laboratory experiments. The second solution is the implementation of counterfactual causal inference model on nonexperimental data. It is "an approach that most naturally applies to "large N" observational studies." (Watts 2014, p.336) The third solution is out-of-sample testing, which evaluates the validity of certain explanations according to their ability to predict.

Addendum

Watts (2014)'s criticism on sociologists' reliance on commonsense definitely offers credible and sharp advice for sociological studies. His emphasis on causal inference, prediction and scientific validity in general also points to the right direction for future research. Nevertheless, theoretical models do not receive their deserved credits in Watts (2014)'s paper. Despite the limitations of and the potential problems brought by theories with questionable or rigid assumptions, theories could benefit causal inference and prediction. Theories can provide initial clues and assumptions to be examined by scientific causal studies. They can also trigger interesting and meaningful question that set the research direction for causal inference and prediction. Besides, they can provide insights that support the formulation of valid causal mechanisms and sensible predictions based on data analysis.