Assignment 7

1. Unit Testing in Python

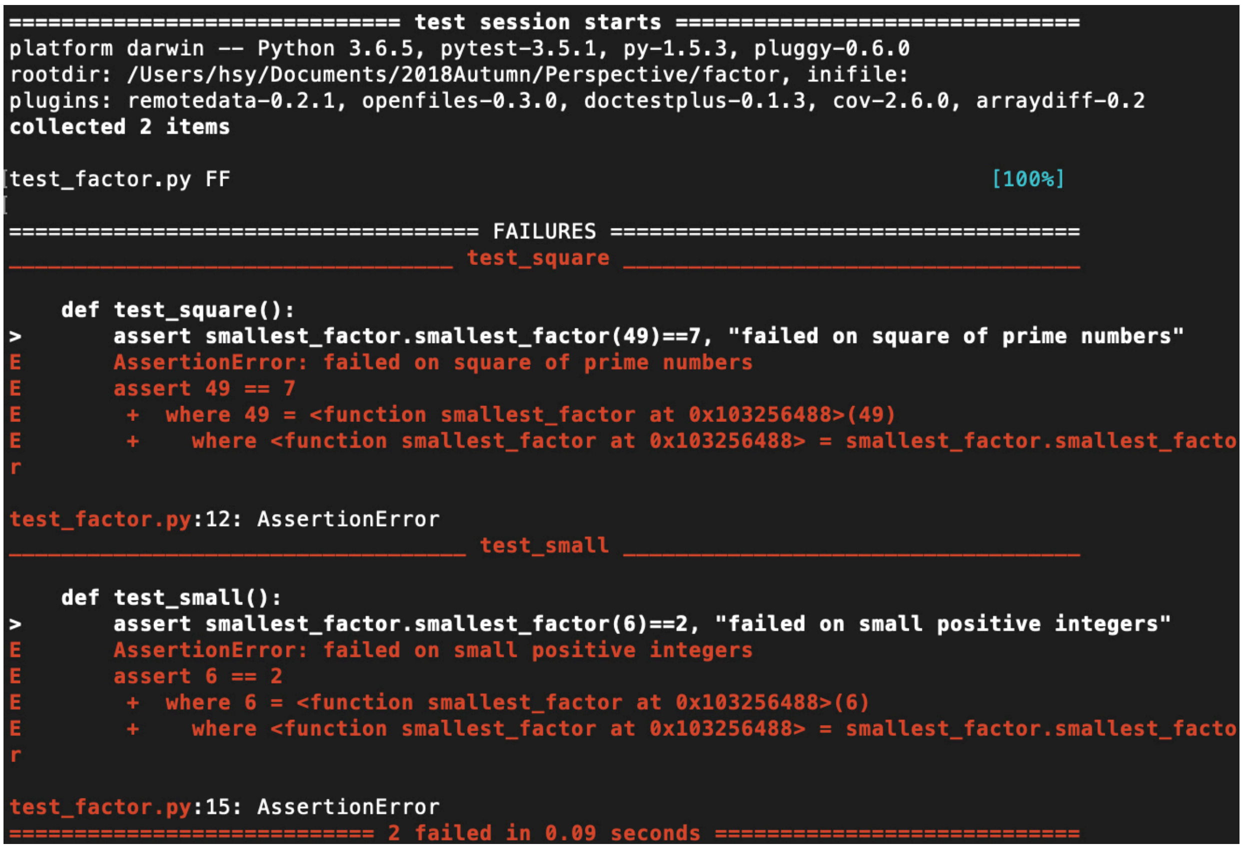
Problem 1

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

def test_square():
    assert smallest_factor(49)==7, "failed on square of prime numbers"

def test_small():
    assert smallest_factor(6)==2, "failed on small positive integers"
```

The testing result was:



```
============================== test session starts ===============================
platform darwin -- Python 3.6.5, pytest-3.5.1, py-1.5.3, pluggy-0.6.0
rootdir: /Users/hsy/Documents/2018Autumn/Perspective/factor, inifile:
plugins: remotedata-0.2.1, openfiles-0.3.0, doctestplus-0.1.3, cov-2.6.0, arraydiff-0.2
collected 2 items

[test_factor.py FF                                                        [100%]

==================================== FAILURES ====================================
__________________________________ test_square ___________________________________

    def test_square():
>       assert smallest_factor.smallest_factor(49)==7, "failed on square of prime numbers"
E       AssertionError: failed on square of prime numbers
E       assert 49 == 7
E        +  where 49 = <function smallest_factor at 0x103256488>(49)
E        +    where <function smallest_factor at 0x103256488> = smallest_factor.smallest_facto
r

test_factor.py:12: AssertionError
___________________________________ test_small ___________________________________

    def test_small():
>       assert smallest_factor.smallest_factor(6)==2, "failed on small positive integers"
E       AssertionError: failed on small positive integers
E       assert 6 == 2
E        +  where 6 = <function smallest_factor at 0x103256488>(6)
E        +    where <function smallest_factor at 0x103256488> = smallest_factor.smallest_facto
r

test_factor.py:15: AssertionError
============================ 2 failed in 0.09 seconds ============================
```

I wrote the first test case because the for-loop range in smallest_factor "range(2, int(n**.5))" doesn't include int(n**.5). When a number n is the square of a prime number m, m won't be checked in this function, so it will return n instead of m. I wrote the second test case because I noticed that when n was small, the range in smallest_factor "range(2, int(n**.5))" would be empty, which might lead to errors. This test case failed because 2 was the smallest prime factor of 6, but 2 wasn't even checked in the loop.

The correct version of the function:

```
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
```

Problem 2

In this problem, I separated the function smallest_factor and the test code into two files:

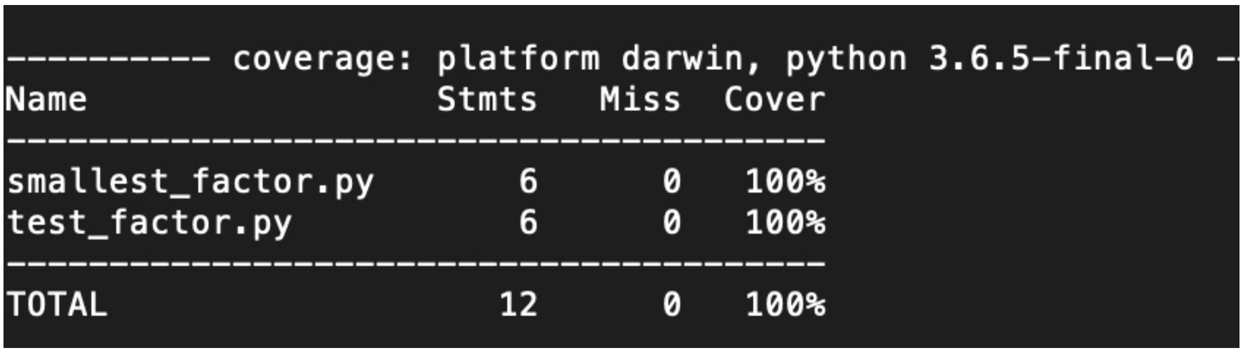
```
#smallest_factor.py
def smallest_factor(n):
    """Return the smallest prime factor of the positive integer n."""
    if n == 1: return 1
    for i in range(2, int(n**.5)):
        if n % i == 0: return i
    return n
```

```
#test_factor.py
import smallest_factor

def test_square():
    assert smallest_factor.smallest_factor(49)==7, "failed on square of
prime numbers"

def test_small():
    assert smallest_factor.smallest_factor(6)==2, "failed on small
positive integers"
```

Then I ran "py.test --cov" to check the coverage:



```
---------- coverage: platform darwin, python 3.6.5-final-0 -
Name                     Stmts   Miss  Cover
--------------------------------------------
smallest_factor.py           6      0   100%
test_factor.py               6      0   100%
--------------------------------------------
TOTAL                       12      0   100%
```

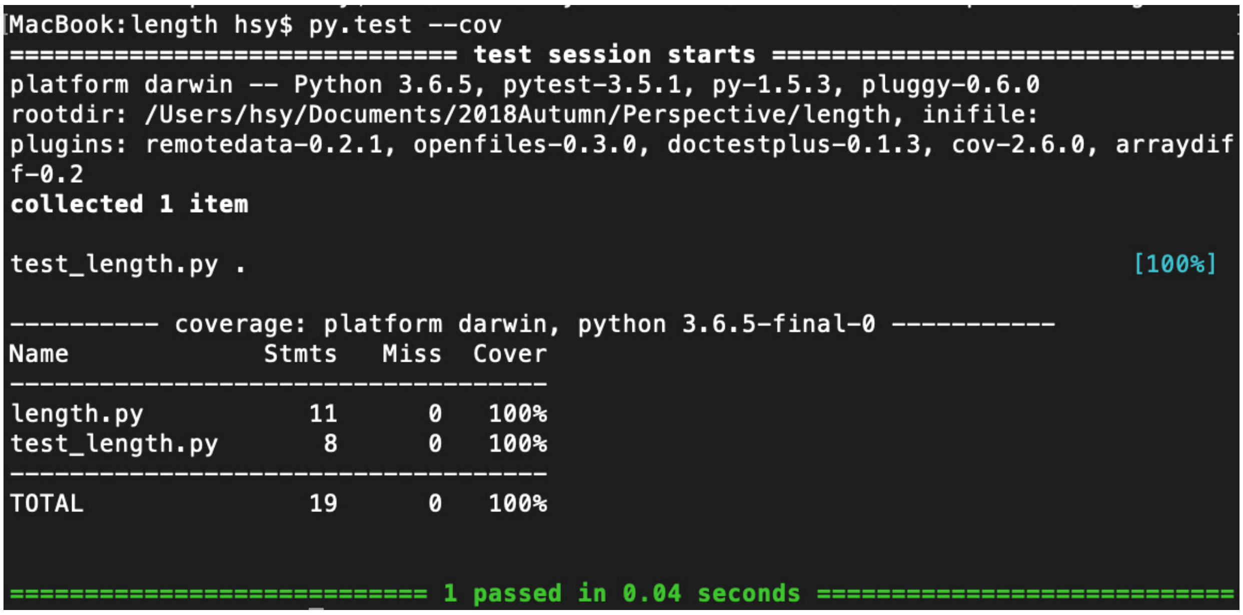
The coverage of both files are 100%. Notice that although I didn't write a test case for n=1, the line "if n == 1: return 1" was covered because the "if" condition must be executed.

Following is the unit test code for the function "month_length":

```
#test_length.py
import length

def test_length():
    assert length.month_length('September')==30
    assert length.month_length('January')==31
    assert length.month_length('February')==28
    assert length.month_length('February', leap_year=True)==29
    assert length.month_length('Nonsense')==None
```

The coverage is 100%.



```
MacBook:length hsy$ py.test --cov
============================= test session starts ==============================
platform darwin -- Python 3.6.5, pytest-3.5.1, py-1.5.3, pluggy-0.6.0
rootdir: /Users/hsy/Documents/2018Autumn/Perspective/length, inifile:
plugins: remotedata-0.2.1, openfiles-0.3.0, doctestplus-0.1.3, cov-2.6.0, arraydif
f-0.2
collected 1 item

test_length.py .                                                         [100%]

---------- coverage: platform darwin, python 3.6.5-final-0 -----------
Name             Stmts   Miss  Cover
------------------------------------
length.py           11      0   100%
test_length.py       8      0   100%
------------------------------------
TOTAL               19      0   100%


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

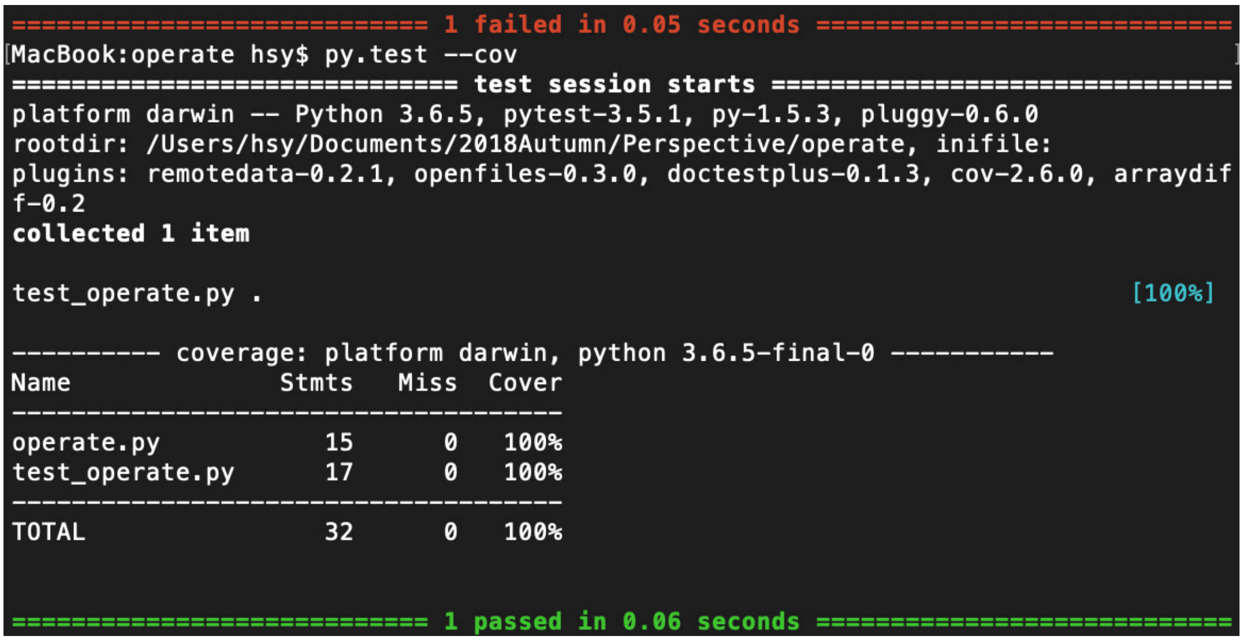
Problem 3

There are 3 types of exception (TypeError, ZeroDivisionError, ValueError) in this function. So in the test code, we need to make sure these exceptions are raised property with correct error messages.

```
import operate
import pytest
```

```
def test_operate():
    assert operate.operate(4, 1, '+') == 5, "add"
    assert operate.operate(4, 1, '-') == 3, "minus"
    assert operate.operate(4, 1, '*') == 4, "multiply"
    assert operate.operate(4, 2, '/') == 2, "normal divide"
    with pytest.raises(TypeError) as excinfo1:
        operate.operate(4, 1, 1)
    assert excinfo1.value.args[0] == "oper must be a string"
    with pytest.raises(ZeroDivisionError) as excinfo2:
        operate.operate(4, 0, '/')
    assert excinfo2.value.args[0] == "division by zero is undefined"
    with pytest.raises(ValueError) as excinfo3:
        operate.operate(4, 0, 'haha')
    assert excinfo3.value.args[0] == "oper must be one of '+', '/', '-',
or '*'"
```

Then I ran "py.test --cov" to check the coverage:



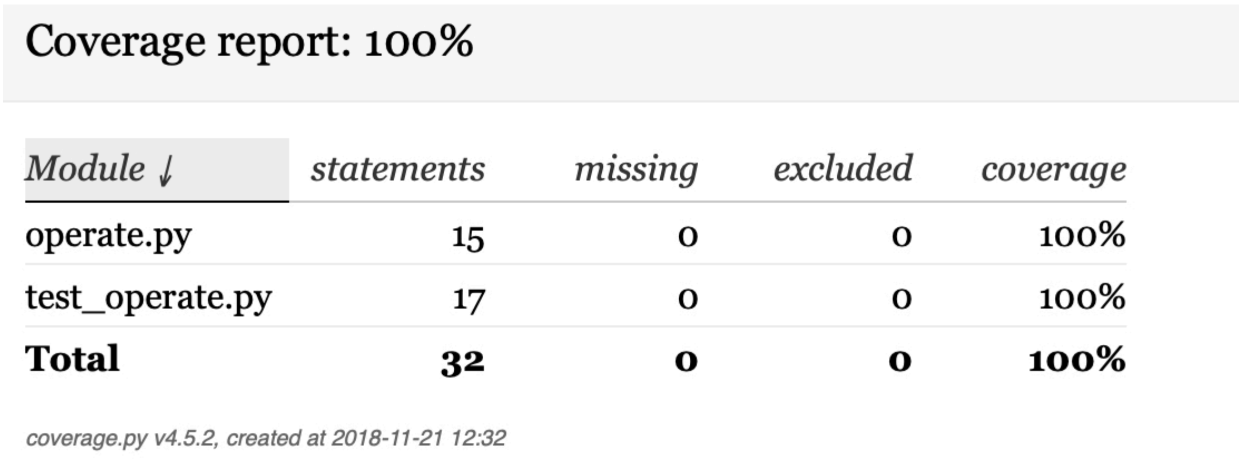
```
=========================== 1 failed in 0.05 seconds ===========================
MacBook:operate hsy$ py.test --cov
============================= test session starts ==============================
platform darwin -- Python 3.6.5, pytest-3.5.1, py-1.5.3, pluggy-0.6.0
rootdir: /Users/hsy/Documents/2018Autumn/Perspective/operate, inifile:
plugins: remotedata-0.2.1, openfiles-0.3.0, doctestplus-0.1.3, cov-2.6.0, arraydif
f-0.2
collected 1 item

test_operate.py .                                                        [100%]

---------- coverage: platform darwin, python 3.6.5-final-0 -----------
Name               Stmts   Miss  Cover
--------------------------------------
operate.py            15      0   100%
test_operate.py       17      0   100%
--------------------------------------
TOTAL                 32      0   100%


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

The cov-report tool can generate a html report for the test, showing that the coverage is 100%.



Coverage report: 100%

Module ↓	statements	missing	excluded	coverage
operate.py	15	0	0	100%
test_operate.py	17	0	0	100%
Total	**32**	**0**	**0**	**100%**

coverage.py v4.5.2, created at 2018-11-21 12:32

2. Test driven development

(b)

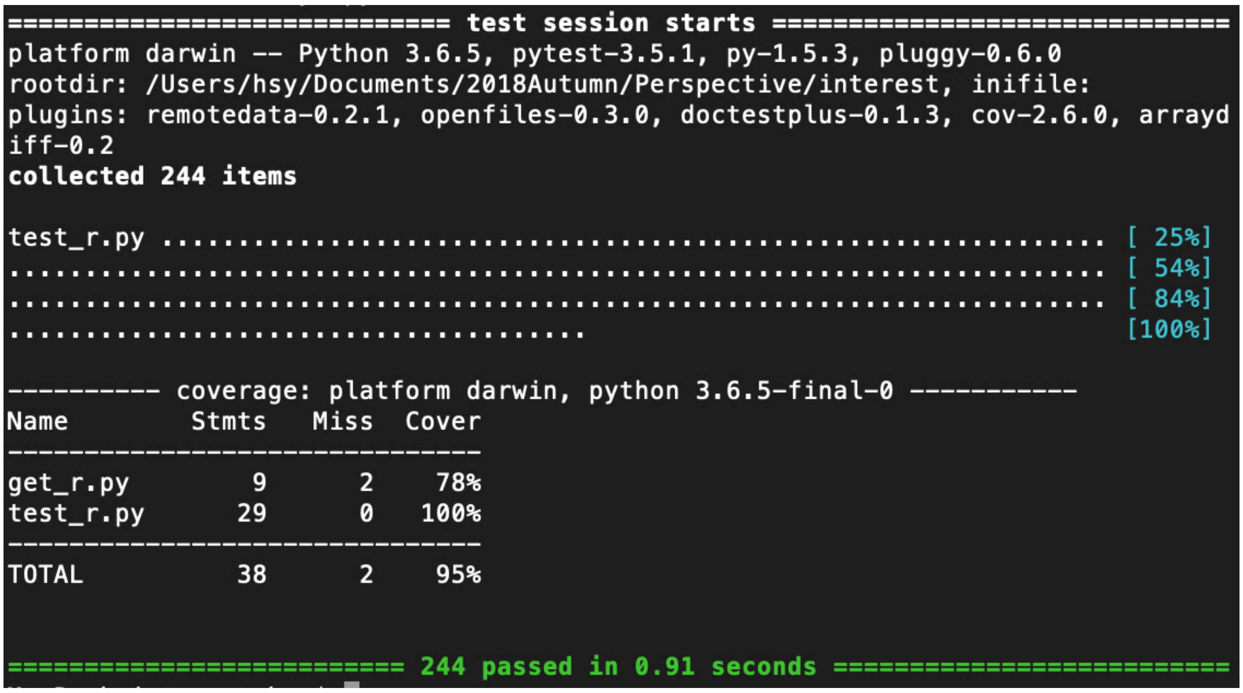
```
import numpy as np

def get_r(K, L, alpha, Z, delta):
    '''
    This function generates the interest rate or vector of interest rates
    '''
    if (type(alpha) != float) or (type(delta) != float) or (type(Z) != 
float and type(Z) != int):
        raise TypeError("alpha and delta must be floats, Z must be float 
or int")
    if (type(K) != float and type(K) != int and type(K) != np.ndarray) or 
(type(L) != float and type(L) != int and type(L) != np.ndarray):
        raise TypeError("K and L must be floats or int or numpy.ndarray")
#    if (np.isscalar(K) and not np.isscalar(L)) or (np.isscalar(L) and not 
np.isscalar(K)) or (not np.isscalar(L) and not np.isscalar(K) and L.shape 
!= K.shape):
#        raise ValueError("dimensions of K and L must be the same")
#    if (alpha >= 1) or (delta >= 1) or (alpha <= 0) or (delta <= 0):
#        raise ValueError("alpha and delta must be between 0 and 1")
#    if np.isscalar(K):
#        if K<=0 or L<=0:
#            raise ValueError("K and L must be positive")
#    else:
#        if sum(K<=0)>0 or sum(L<=0)>0:
#            raise ValueError("K and L must be positive")
    r = alpha*Z*((L/K)**(1-alpha)) - delta
    return r
```

I wrote restrictions on the parameters and error handling as described in the problem. But there were test cases that violated these restrictions (delta=0, K is a scalar but L is a vector), so I had to comment out these codes.

(c)

By simply implementing the function in one line without any error handling would pass all the test cases. And the coverage of the module get_r was 78%.


```
============================== test session starts ===============================
platform darwin -- Python 3.6.5, pytest-3.5.1, py-1.5.3, pluggy-0.6.0
rootdir: /Users/hsy/Documents/2018Autumn/Perspective/interest, inifile:
plugins: remotedata-0.2.1, openfiles-0.3.0, doctestplus-0.1.3, cov-2.6.0, arrayd
iff-0.2
collected 244 items

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

---------- coverage: platform darwin, python 3.6.5-final-0 -----------
Name        Stmts   Miss  Cover
-------------------------------
get_r.py        9      2    78%
test_r.py      29      0   100%
-------------------------------
TOTAL          38      2    95%


========================== 244 passed in 0.91 seconds ===========================
```

3. Watts (2014)

(a)

When rational choice theory was first introduced in the 1960s, there were criticisms about the assumptions of it. Critics argued that the assumptions "about the preferences, knowledge, and computational capabilities of the actors in question" were "implausible or empirically invalid" (Watts, 2014: p.320). Others criticized that the predictions of rational choice theory were not consistent with reality.(Watts, 2014: p.320)

(b)

According to Watts, the main pitfall of commonsense theories of action is that given an explanation from commonsense that fits with our everyday observations, we can not guarantee that this explanation can be generalized to be a "causal mechanism" in a broader context. In fact, commonsense explanations are conceived to be correct in our everyday life because the mistakes they make are too minor or people quickly correct these mistakes by substituting a wrong prior reason with a correct one based on the outcome. So our commonsense knowledge is not as accurate as we conceived, so theorizing it "can undermine the scientific validity of the resulting explanations"(Watts, 2014: p.327).

(c)

The major issue with rational choice modeling and causal explanation lies in that rational choice theorists rely more on "empathetic view of explanation"(Watts, 2014: p.321). So the solution to this issue, as Watts points out, is to conceptually realize "the difference between empathetic and causal explanation and thereby to produce more scientifically rigorous if not more satisfying explanations"(Watts, 2014: p.335). Specifically, Watts suggets that sociologists should "rely more on experimental methods"(Watts, 2014: p.335) and statistical models for causal explanation. He also suggests that explanations should be evaluated based on their predictive power.

(d)

Using theoretical models with simplifications and specific assumptions about mechanisms is the only way that we can start to study the real world. Because most social science problems are too complicated to study if we don't make any simplifications. And making assumptions does not mean that we will stick to them. We make assumptions because we want a easy start point for our research, and we expect to relax these assumptions and refine our models in the future so that they can be more consistent with reality, have more predictive power, and lead to causal inference. For example, at the very beginning of economics, theories assumed that agents were perfectly rational, and that the underlying mechanism was utility maximization. Early economists made these assumptions not because they were realistic, but that the models based on these assumptions could lead to many intuitive predictions. But there were many other outcomes not consistent with reality, so over the years economists are trying to adjust the assumptions and have developed behavial economics, information economics, etc. So in most cases, we use models with simplifications and specific assumptions about mechanisms because it's the best we can do for the moment.