MACS30000 Assignment 7

Dr. Richard Evans

Submitted by: Junho Choi

Due November 26, 2018 (11:30 AM)

Problem 1

For this question, I note that all of the codes are in the folder **Problem1** of **A7**, which is divided into folders **Problem1_1**, **Problem1_2**, **Problem1_3**. However, because it may be bothersome for viewers to open each of the `.py` files, I have separately created a `.ipynb` file (with the filename `junhoc_A7.ipynb` with its `.pdf` counterpart `junhoc_A7_ipynb.pdf`) in the **A7** folder that aggregates all the necessary codes for this problem.

Problem 1 of Chapter 7 in Humpherys and Jarvis (2018)

For this question, I assumed away that the input n is already a natural number. Therefore, I did not test for the type or sign of n (e.g. whether n is a string, or whether it is negative or not).

The most problematic (yet subtle) error in the original code for the function `smallest_factor()` seems to have to do with squares of prime numbers (e.g. 4, 9, 25, 49, and so forth). For n above 1, the for-loop runs from 2 (inclusive) to floor($\sqrt{n}$) (exclusive).[1] Then, for instance, with the case of 9, the for-loop will run from 2 (inclusive) to 3 (exclusive), meaning that it will never be the case in which 3 is even tested as the greatest factor. One way to surely fix this problem would be to run for-loop from 2 (inclusive) to floor($\sqrt{n}$) + 1 (or `int(sqrt ** 0.5) + 1` in Python code) (exclusive).

In light of this, `test_small.py` (in the folder **Problem1_1**) checks whether the function `smallest_factor()`, with the input of 25, has an output of 5. In fact, the original code for `smallest_factor()` has an output of 25 due to not checking 5 as its factor.

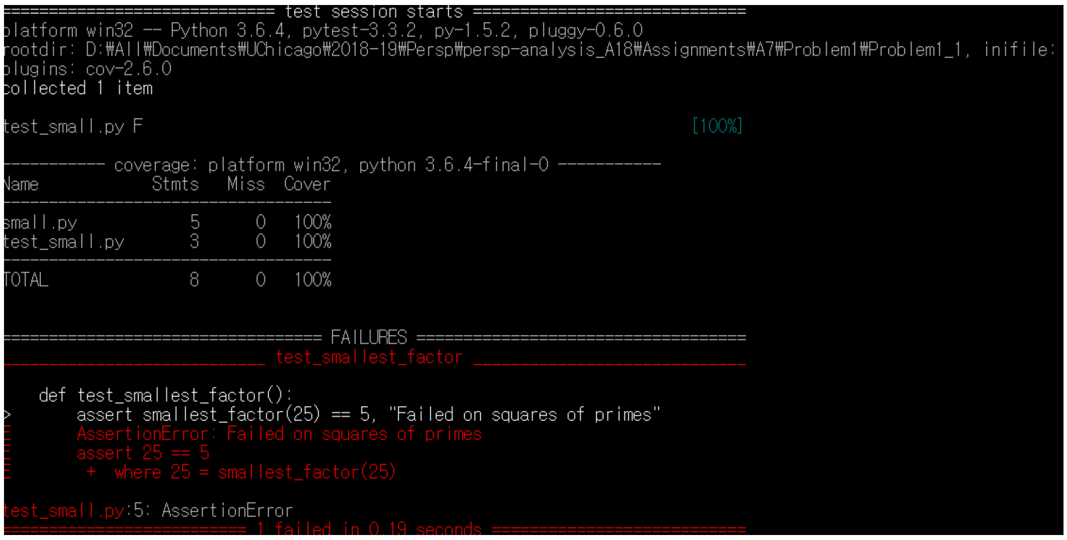
Problem 2 of Chapter 7 in Humpherys and Jarvis (2018)

Firstly, the question asks us to report the coverage of `smallest_factor()`. In addition, if the original test from the previous problem did have full coverage, then it asks us to add lines of code to the test so that it does have full coverage. In the case of `test_small.py` that I wrote, it indeed had full coverage; this can be confirmed from Figure 1. In addition, because the initial code for `smallest_factor()` was error-prone, we see that it has failed to pass the test.

For the second part of this problem, I note that the code for a comprehensive unit test for the function `month_length()` can be found in `test_monlen.py` (in the folder

[1]This is due to the fact that, in Python, casting `int` over positive number will *round down* the number to the nearest integer below it.

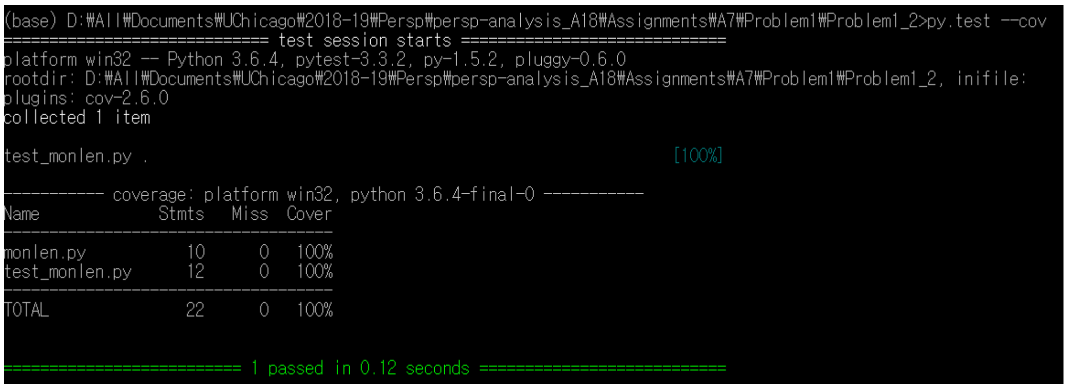
Figure 1: Screen Capture of Coverage Test for `smallest_factor()`



```
=========================== test session starts ============================
platform win32 -- Python 3.6.4, pytest-3.3.2, py-1.5.2, pluggy-0.6.0
rootdir: D:₩All₩Documents₩UChicago₩2018-19₩Persp₩persp-analysis_A18₩Assignments₩A7₩Problem1₩Problem1_1, inifile:
plugins: cov-2.6.0
collected 1 item

test_small.py F                                                       [100%]

----------- coverage: platform win32, python 3.6.4-final-0 -----------
Name            Stmts   Miss  Cover
-----------------------------------
small.py            5      0   100%
test_small.py       3      0   100%
-----------------------------------
TOTAL               8      0   100%


================================= FAILURES =================================
___________________________ test_smallest_factor ___________________________

    def test_smallest_factor():
>       assert smallest_factor(25) == 5, "Failed on squares of primes"
E       AssertionError: Failed on squares of primes
E       assert 25 == 5
E        +  where 25 = smallest_factor(25)

test_small.py:5: AssertionError
========================= 1 failed in 0.19 seconds =========================
```

Problem1_2). It checks whether the function `month_length()` produces the correct output (as the number of days) when given the name of a certain month. In addition, when the input is not a name of a certain month (say, some string like `Something_Else` that I used for the test), it checks whether the output is `None`. Just in case it needs to be checked whether this test produces full coverage, I provide the report in Figure 2.

Figure 2: Screen Capture of Coverage Test for `month_length()`



```
(base) D:₩All₩Documents₩UChicago₩2018-19₩Persp₩persp-analysis_A18₩Assignments₩A7₩Problem1₩Problem1_2>py.test --cov
=========================== test session starts ============================
platform win32 -- Python 3.6.4, pytest-3.3.2, py-1.5.2, pluggy-0.6.0
rootdir: D:₩All₩Documents₩UChicago₩2018-19₩Persp₩persp-analysis_A18₩Assignments₩A7₩Problem1₩Problem1_2, inifile:
plugins: cov-2.6.0
collected 1 item

test_monlen.py .                                                      [100%]

----------- coverage: platform win32, python 3.6.4-final-0 -----------
Name             Stmts   Miss  Cover
------------------------------------
monlen.py           10      0   100%
test_monlen.py      12      0   100%
------------------------------------
TOTAL               22      0   100%


========================= 1 passed in 0.12 seconds =========================
```

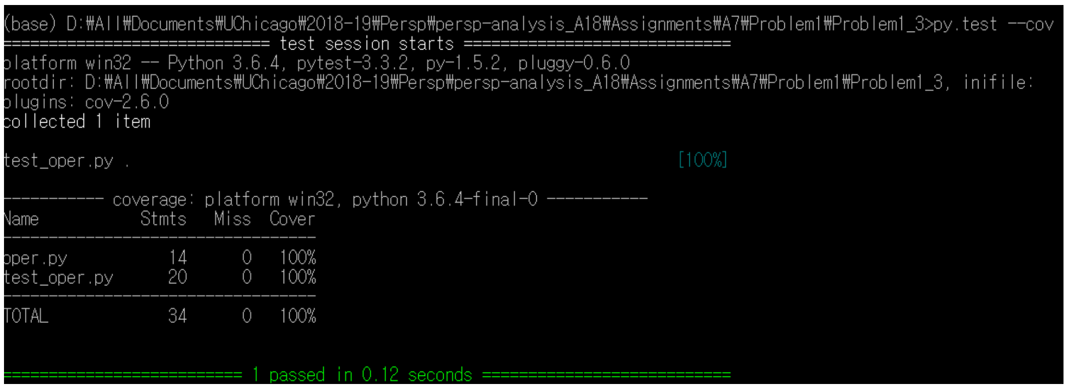
Problem 3 of Chapter 7 in Humpherys and Jarvis (2018)

For this question, I have noticed that there are potentially seven cases (and if including sub-cases, possibly more) to check for. These are as follows: 1) addition, 2) subtraction, 3) multiplication, 4) division where the divisor is non-zero, 5) case of dividing by zero, 6) case in which the operand is not a string, 7) case in which the operand is a string but not one of +, -, /, or *. As for the latter three, they are cases in which specific exception errors are raised and, as Humpherys and Jarvis (2018) elaborates, should use the code `with pytest.raises()` to explicitly check for (p. 102).

Therefore, in `test_oper.py` (in the folder **Problem1_3**) was written to check all of such cases. For the former four cases (in which exception errors are *not* raised), each case is divided into two sub-cases in which tests operation that produces an `int` output and one that produces a `float` output. This was inspired by the function `test_divide()` in Humpherys and Jarvis (2018, p. 102).

The coverage report using the line `py.test --cov` can be found in Figure 3. In addition, `.html` version of the coverage report can be found in the file `test_oper_py.html` of the sub-folder **htmlcov** (in the folder **Problem1_3**).

Figure 3: Screen Capture of Coverage Test for `operation()`



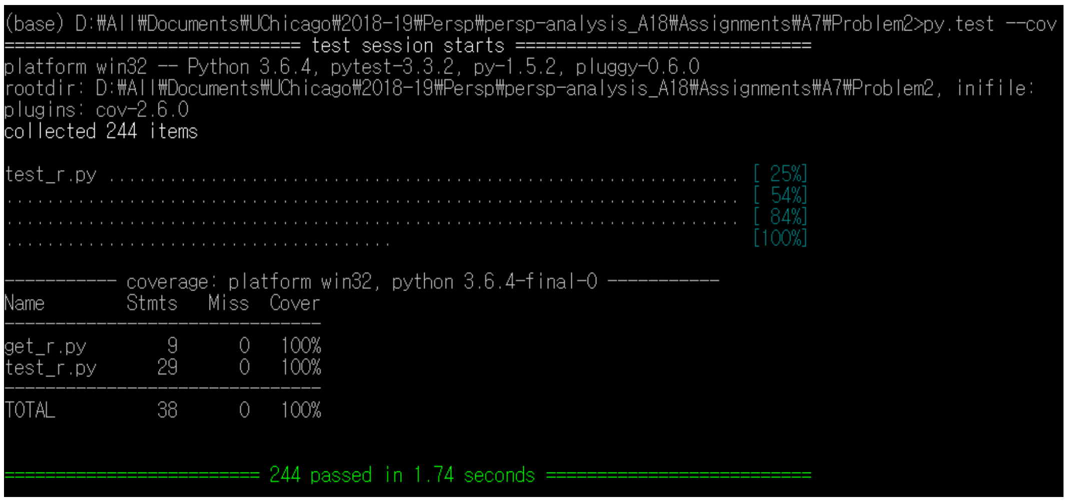
```
(base) D:₩All₩Documents₩UChicago₩2018-19₩Persp₩persp-analysis_A18₩Assignments₩A7₩Problem1₩Problem1_3>py.test --cov
============================= test session starts =============================
platform win32 -- Python 3.6.4, pytest-3.3.2, py-1.5.2, pluggy-0.6.0
rootdir: D:₩All₩Documents₩UChicago₩2018-19₩Persp₩persp-analysis_A18₩Assignments₩A7₩Problem1₩Problem1_3, inifile:
plugins: cov-2.6.0
collected 1 item

test_oper.py .                                                           [100%]

----------- coverage: platform win32, python 3.6.4-final-0 -----------
Name           Stmts   Miss  Cover
----------------------------------
oper.py           14      0   100%
test_oper.py      20      0   100%
----------------------------------
TOTAL             34      0   100%


========================== 1 passed in 0.12 seconds ===========================
```

Problem 2

For this question, I note that all of the necessary files (including the most important `get_r.py` and `test_r.py`) are in the folder **Problem2** of **A7**. However, once again, because it may be bothersome for viewers to open each of the `.py` files, I have copied and pasted the codes into the aforementioned `.ipynb` file (`junhoc_A7.ipynb` and its `.pdf` counterpart `junhoc_A7_ipynb.pdf`) in the **A7** folder.

In order to confirm that all the tests have been passed and that the `get_r` function I have written has full coverage, I provide the report in Figure 4.

Figure 4: Screen Capture of Coverage Test for `get_r()`



```
(base) D:₩All₩Documents₩UChicago₩2018-19₩Persp₩persp-analysis_A18₩Assignments₩A7₩Problem2>py.test --cov
============================ test session starts =============================
platform win32 -- Python 3.6.4, pytest-3.3.2, py-1.5.2, pluggy-0.6.0
rootdir: D:₩All₩Documents₩UChicago₩2018-19₩Persp₩persp-analysis_A18₩Assignments₩A7₩Problem2, inifile:
plugins: cov-2.6.0
collected 244 items

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

----------- coverage: platform win32, python 3.6.4-final-0 -----------
Name        Stmts   Miss  Cover
-------------------------------
get_r.py        9      0   100%
test_r.py      29      0   100%
-------------------------------
TOTAL          38      0   100%


========================= 244 passed in 1.74 seconds ==========================
```

Problem 3

In Watts (2015), the author argues that, in spite of the belief that sociological methodologies elucidate what may be considered against the common sense, researchers in sociology indeed rely on common sense more than what they are led to believe (p. 313). While underlining this fact, the author makes note of the overarching concept in which many sociological theories fall under – rationalizable actions – and how this can be regarded less as a scientific idea and more as a "commonsense" or "folk" theory (Watts 2015, p. 314).

The author's criticism towards the aforementioned rationalizable actions is exemplified with the case of rational choice theory. Rational choice theory was sharply criticized by academics across a variety of fields, mostly due to its assumptions (such as those on individual preferences, information, or ability to compute) or not being able to yield much empirical evidence. For instance, the assumptions were deemed unrealistic or unsupported by actual evidence (Watts 2015, p. 320). Along this trend, the cornerstones of rational choice theory, such as the principle of utility maximization (along with its weaker versions), have decayed as well (Watts 2015, p. 320). Yet the aforementioned are not exactly within the scope of criticism that the author seeks to provide. Rather, the author's point is that rational choice theory has moved on from "emphasis on prediction and deduction to an emphasis on understandability and sense making" (Watts 2015, p. 321).

Essentially, the author makes a criticism towards the tendency in sociology (and broadly, in social sciences as a whole) that rather than attempting to scientifically predict outcomes, it now seeks to explain in an empathetic manner what resonates with common sense (Watts 2015, p. 321). Extending this argument, the author points to the crux of the problem in using commonsense theories: that while common sense seems to be generally valid, it is not unequivocally valid in *every* situation (Watts 2015, p. 327). While general validity calls for understandability, "universal validity" corresponds to scientific precision and predictiveness. The author also explicates that this danger of using commonsense theory is amplified by the fact that, within in the field of sociology, there seems to be a "conflation," or a confusion of ideas, between understandability ("emphathetic explanation") and prediction ("scientific explanation") (Watts 2015, p. 327).

The author's suggested solution to the reliance on commonsense theories is using statistical, econometric, or mathematical methods for prediction, which essentially would replace models that emphasize understandability (Watts 2015, p. 336). Preceding this notion, however, is the re-definition of the word prediction, which can be wrongly interpreted as something that is deterministic, only about the future, and only involving specific instances (Watts 2015, p. 337). In fact, the author refutes the preconceived notion of prediction by elaborating that predictions can be probabilistic, about the analysis of the past, and about "stylized" cases in which extension to specific cases may not be feasible (Watts 2015, pp. 337-339). By allowing this "extended notion" of prediction, a number of methods can be evaluated, including large-N observational studies, small-N comparative studies, and mathematical and agent-based models (Watts 2015, p. 340).

Such solutions, while being able to partially solve the problems in the theories of rationalizable actions (including rational choice theory), almost belittles the potential importance that models with specific assumptions or mechanisms can have. Models are simplifications of actual phenomena that can exactly fend off what the author warns researchers of: the danger in the usage of common sense. Often, researchers and academics may use computer simulations or mathematical models to find intuition about the world that may be *counter-intuitive* at first. Such intuitions are not innate, but rather trained so that they guide one's design of models to empirically test such models. Therefore, the sole emphasis on prediction – even in the extended sense – may not be adequate.

References

Humpherys, Jeffrey and Tyler J. Jarvis. 2018. "'Labs for Foundations of Applied Mathematics: Python Essentials," creative commons, open access.

Humpherys, Jeffrey, Tyler J. Jarvis, and Emily J. Evans. 2017. *Foundations of Applied Mathematics: Mathematical Analysis*, Vol. 1, SIAM: Society for Industrial and Applied Mathematics.

Watts, Duncan J. 2015. "Common Sense and Sociological Explanations." *American Journal of Sociology* 120 (2): 313-351.