CS256 Advanced Programming – Assign4

Submit a compressed tar file **XXX_Assign4.tar.gz**, which include the following files:

1) **typescript**
2) **find_max_min.py**
3) **rearrange.py**

Exercise –4.1 This exercise aims at practicing the usage of pdb in debugging your program.

1: stepping through a program

Copy and paste the code snippet below into a file named pdb_exercise_1.py

```
name = 'alice'
greeting = 'hello ' + name
print(greeting)
```

Now invoke the script using the python debugger via the command below.

```
python -m pdb pdb_exercie_1.py
```

In the above pdb is the three letter acronym for the Python Debugger. You should be greeted by the prompt below.

```
> pdb_exercise_1.py(1)<module>()
-> name = 'alice'
(Pdb)
```

The debugger shows the next line to be executed (-> name = 'alice') as well as the prompt for interacting with the debugger ((Pdb)).

Type in n, short for next, to execute the line displayed. You should now see the output below.

```
(Pdb) n
> pdb_exercise_1.py(2)<module>()
-> greeting = 'hello' + name
```

Let us check the value of the newly assigned name variable. Type in p name(p as in "print"). It should tell you that the name is "alice". Type in n again to execute the next command.
The greeting variable should now have been assigned the string "hello alice".

```
(Pdb) p name
'alice'
(Pdb) n
> pdb_exercise_1.py(3)<module>()
> -> print(greeting)
> (Pdb) p greeting
> 'hello alice'
```

When debugging it is quite easy to lose the frame of reference as to where one is in the code. To put things into context type in l as in list (the source code for the current file). You should see output below.

```
(Pdb) l
  1     name = 'alice'
  2     greeting = 'hello ' + name
  3  -> print(greeting)
[EOF]
```

Type in n again to execute the last command.

```
(Pdb) n
hello alice
```

Finally, type in q to quit the debugger.

2: stepping into functions

Let us create a script with a function. Copy and paste the code snippet below into a file named pdb_exercise_2.py.

```
def greet(name):
    greeting = 'hello ' + name
    return greeting

greeting = greet('alice')
print(greeting)
```

Start the debugger.

```
python -m pdb pdb_exercise_2.py
```

This time, rather than stepping through the program, press c (which stands for "continue execution").

```
(Pdb) c
hello alice
The program finished and will be restarted
> pdb_exercise_2.py(1)<module>()
-> def greet(name):
(Pdb)
```

Basically the program ran from beginning to end, printing out the greeting, and then it restarted itself leaving us at the (Pdb) prompt.

This time use n to walk through the script. Note that you only need to enter n three times to get to the end of the program and that the debugger does not step into the greet() function. You should see the output below.

```
> pdb_exercise_2.py(1)<module>()
-> def greet(name):
(Pdb) n
> pdb_exercise_2.py(5)<module>()
-> greeting = greet('alice')
(Pdb) n
> pdb_exercise_2.py(6)<module>()
-> print(greeting)
(Pdb) n
hello alice
```

In other words n continues execution until the next line in the current function is reached or it returns.

Press c to restart the program and press n once to get to the line where the greet function is about to be called.

```
> pdb_exercise_2.py(1)<module>()
-> def greet(name):
(Pdb) n
> pdb_exercise_2.py(5)<module>()
```

This time we will use s to step into the greet() function, then we will continue walking through the program using n. Note the difference now that you have stepped into the greet() function.

```
(Pdb) s
--Call--
> pdb_exercise_2.py(1)greet()
-> def greet(name):
(Pdb) n
> pdb_exercise_2.py(2)greet()
-> greeting = 'hello ' + name
```

```
(Pdb) n
> pdb_exercise_2.py(3)greet()
-> return greeting
(Pdb) n
--Return--
> pdb_exercise_2.py(3)greet()->'hello alice'
-> return greeting
(Pdb) n
> pdb_exercise_2.py(6)<module>()
-> print(greeting)
(Pdb) n
hello alice
--Return--
> pdb_exercise_2.py(6)<module>()->None
-> print(greeting)
(Pdb)
```

Finally let us have a look at the r command, which stands for return. This is similar to the c command, but rather than continuing to the end of the program runs to the end of the function.

Let us try it out, start off by entering c to restart the program then enter n and s. You should now be in the greet() function.

```
(Pdb) c
The program finished and will be restarted
> pdb_exercise_2.py(1)<module>()
-> def greet(name):
(Pdb) n
> pdb_exercise_2.py(5)<module>()
-> greeting = greet('alice')
(Pdb) s
--Call--
> pdb_exercise_2.py(1)greet()
-> def greet(name):
(Pdb)
```

As a sanity check, use l to list where you are in the code. You should see the below.

```
(Pdb) l
  1  -> def greet(name):
  2         greeting = 'hello ' + name
  3         return greeting
  4
  5     greeting = greet('alice')
  6     print(greeting)
[EOF]
(Pdb)
```

Now press r as in "return".

```
(Pdb) r
--Return--
> pdb_exercise_2.py(3)greet()->'hello alice'
-> return greeting
(Pdb)
```

Note that we are immediately placed at the end of the function where it is about to deliver its return value.

Exercise 3: getting help

When using a tool infrequently it is easy to forget what the commands are named and what they do. However, using the help command it is easy to refresh your memory.

```
(Pdb) help
```

```
Documented commands (type help <topic>):
========================================
EOF    bt         cont      enable  jump  pp       run      unt
a      c          continue  exit    l     q        s        until
alias  cl         d         h       list  quit     step     up
args   clear      debug     help    n     r        tbreak   w
b      commands   disable   ignore  next  restart  u        whatis
break  condition  down      j       p     return   unalias  where

Miscellaneous help topics:
==========================
exec  pdb

Undocumented commands:
======================
retval  rv
```

Let us have a look at the help descriptions of the commands that we have been using so far.

```
(Pdb) help n
n(ext)
Continue execution until the next line in the current function
is reached or it returns.
(Pdb) help s
s(tep)
Execute the current line, stop at the first possible occasion
(either in a function that is called or in the current function).
(Pdb) help c
c(ont(inue))
Continue execution, only stop when a breakpoint is encountered.
(Pdb) help r
r(eturn)
Continue execution until the current function returns.
(Pdb) help l
l(ist) [first [,last]]
List source code for the current file.
Without arguments, list 11 lines around the current line
or continue the previous listing.
With one argument, list 11 lines starting at that line.
With two arguments, list the given range;
if the second argument is less than the first, it is a count.
(Pdb) help help
h(elp)
Without argument, print the list of available commands.
With a command name as argument, print help about that command
"help pdb" pipes the full documentation file to the $PAGER
"help exec" gives help on the ! command
(Pdb)
```

4: interacting with the program under inspection

Up until this point we have not actually had any errors in our scripts to correct. Let us change that. Copy and paste the code below into a file named pdb_exercise_4.py.

```
import sys

def magic(x, y):
    return x + y * 2

x = sys.argv[1]
y = sys.argv[1]
```

```
answer = magic(x, y)
print('The answer is: {}'.format(answer))
```

Suppose that we run this script with the inputs 1 and 50 expecting the result 101.

```
python pdb_exercise_4.py 1 50
The answer is: 111
```

What is going on?

Now, rather than inserting print statements all over the code to work it out, let us examine the code in the debugger.

```
python -m pdb pdb_exercise_4.py 1 50
```

Let us get to the point where we have access to the variables x and y.

```
> pdb_exercise_4.py(1)<module>()
-> import sys
(Pdb) n
> pdb_exercise_4.py(3)<module>()
-> def magic(x, y):
(Pdb) n
> pdb_exercise_4.py(6)<module>()
-> x = sys.argv[1]
(Pdb) n
> pdb_exercise_4.py(7)<module>()
-> y = sys.argv[1]
(Pdb) n
> pdb_exercise_4.py(9)<module>()
-> answer = magic(x, y)
(Pdb)
```

First of all let us see what attributes are available in the scope of the program. We can do this using p for print.

```
(Pdb) p dir()
['__builtins__', '__file__', '__name__', '__package__', 'magic', 'sys', 'x',
 'y']
```

There is also pp for pretty print.

```
(Pdb) pp dir()
['__builtins__',
 '__file__',
 '__name__',
 '__package__',
 'magic',
 'sys',
 'x',
 'y']
```

So what is x?

```
(Pdb) p x
'1'
```

Hey, that looks suspiciously like a string. Note that we can use raw Python within the debugger. Let us find out type x is.

```
(Pdb) type(x)
<type 'str'>
```

The fact that we can execute Python within the debugger means that we can change the input variables dynamically.

```
(Pdb) x = int(x)
```

```
(Pdb) y = int(y)
```

Let us just check the values before we run the program.

```
(Pdb) p x, y
(1, 1)
```

What y is 1 not 50?

Inspecting the code we find that I forgot to update the index when I copied the input parsing line (note line 7 in the code listing below).

```
(Pdb) l
  4         return x + y * 2
  5
  6     x = sys.argv[1]
  7     y = sys.argv[1]
  8
  9  -> answer = magic(x, y)
 10     print('The answer is: {}'.format(answer))
[EOF]
(Pdb)
```

Ok, let us just change the value of y to 50 in the debugger before checking if the code works as expected by letting it run to completion.

```
(Pdb) y = 50
(Pdb) c
The answer is: 101
The program finished and will be restarted
> pdb_exercise_4.py(1)<module>()
-> import sys
(Pdb)
```

5: using breakpoints

So far we have been stepping though the scripts from beginning to end. However, when working on larger programs, this is often not practical. To simulate such a situation, copy and paste the code below into a file named pdb_exercise_5.py.

```python
import time

def slow_subtractor(a, b):
    """Return a minus b."""
    time.sleep(1)
    return a - b

some = slow_subtractor(12, 8)
crazy = slow_subtractor(12, 78)
scientific = slow_subtractor(56, 31)
experiment = slow_subtractor(101, 64)

total = some + crazy + scientific + experiment

experimental_fraction = experiment / total
```

When we run this code we get a ZeroDivisionError.

```
$ python pdb_exercise_5.py
Traceback (most recent call last):
  File "pdb_exercise_5.py", line 15, in <module>
    experimental_fraction = experiment / total
ZeroDivisionError: integer division or modulo by zero
```

Stepping through the code in the debugger would be annoying as you would have to press n every time the slow_subtraction() function was called. Let us instead insert a breakpoint before the line that generates the error.

Use the command b to add a breakpoint at a specified line.
For example,

```
python -m pdb pdb_exercise_5.py
> pdb_exercise_5.py(1)<module>()
-> import sys
(Pdb) n
> pdb_exercise_5.py(3)<module>()
-> def slow_subtractor(a, b):
(Pdb) n
> pdb_exercise_5.py(8)<module>()
-> some = slow_subtractor(12, 8)
(Pdb) n
> pdb_exercise_5.py(9)<module>()
-> crazy = slow_subtractor(12, 78)
(Pdb) b 13
Breakpoint 1 at pdb_exercise_5.py:13
(Pdb) b 15
Breakpoint 2 at pdb_exercise_5.py:15
```

Now use l to list where you are, you will find the program stops at line 9, and there are two breakpoints added to the program on line 13 and 15.

```
4           """Return a minus b."""
5           time.sleep(1)
6           return a - b
7
8       some = slow_subtractor(12, 8)
9   ->  crazy = slow_subtractor(12, 78)
10      scientific = slow_subtractor(56, 31)
11      experiment = slow_subtractor(101, 64)

13  B   total = some + crazy + scientific + experiment

15  B   experimental_fraction = experiment / total
```

Now if continue the execution with command c, the execution will continue until a breakpoint is encountered. You will see the following

```
(Pdb) c
> pdb_exercise_5.py(13)<module>()
-> total = some + crazy + scientific + experiment
(Pdb) c
> pdb_exercise_5.py(15)<module>()
-> experimental_fraction = experiment / total
(Pdb) p total
0
(Pdb) c
```

If you continue the execution, you will see the Traceback which indicates where the error happens.

To remove a breakpoint, use command cl(ear) and specify the line or the breakpoint number. For example, after adding two breakpoints on line 13 and 15, we could remove the first breakpoint as following:

```
(Pdb) b 1
Deleted breakpoint 1 at pdb_exercise_5.py():13
(Pdb) l
4           """Return a minus b."""
5           time.sleep(1)
```

```
6           return a - b
7
8       some = slow_subtractor(12, 8)
9   ->  crazy = slow_subtractor(12, 78)
10      scientific = slow_subtractor(56, 31)
11      experiment = slow_subtractor(101, 64)

13      total = some + crazy + scientific + experiment

15  B   experimental_fraction = experiment / total
(Pdb) c
> pdb_exercise_5.py(15)<module>()
-> experimental_fraction = experiment / total
```

Command disable and enable could temporary disable a breakpoint and enable that breakpoint later. For example, suppose the program is currently at line 13, with the breakpoint 1 at line 13 has already been removed. If we disable the breakpoint 2 at line 15, and enable it immediately, we will see the breakpoint 2 is still in effect.

```
(Pdb) disable 2
Disabled breakpoint 2 at pdb_exercise_5.py:15
(Pdb) enable 2
Enabled breakpoint 2 at pdb_exercise_5.py:15
(Pdb) c
> pdb_exercise_5.py(15)<module>()
-> experimental_fraction = experiment / total
```

Another way to set breakpoint is importing the pdb module and using the pdb.set_trace() function.

```
import time

def slow_subtractor(a, b):
    """Return a minus b."""
    time.sleep(1)
    return a - b

some = slow_subtractor(12, 8)
crazy = slow_subtractor(12, 78)
scientific = slow_subtractor(56, 31)
experiment = slow_subtractor(101, 64)

total = some + crazy + scientific + experiment

import pdb; pdb.set_trace()
experimental_fraction = experiment / total
```

If we run the code now we get dumped into a debugger session before the offending line is executed.

```
python pdb_exercise_5.py
> pdb_exercise_5.py(17)<module>()
-> experimental_fraction = experiment / total
(Pdb) p total
0
(Pdb) p some, crazy, scientific, experiment
(4, -66, 25, 37)
```

6: print the content in the shell window.

You can print out transcripts of your sessions by generating a typescript:

eccs-tools$ script

Script started, output file is typescript

```
eccs-tools $
```
This will copy everything that shows up in your shell window into the file "typescript".

When you are done, type
```
eccs-tools $ exit
exit
Script done, output file is typescript
eccs-tools $
```
You can then print typescript as above.

7: Copy the following code and save it in factorial.py

```
def factorial(n):
  if n==0:
    return 1
  else:
    return n*factorial(n-1)

def print_factorial(n):
  for i in range(n):
    print(factorial(n)

if __name__=='__main__':
  print_factorial(5)
```

Run factorial.py in pdb mode, exercising the following commands:

s(tep)
n(ext)
b(reak)
c(ontinue)
cl(ear)
disable
enable
condition
tbreak
q(uit)

Save the session to a typescript file, and submit the **typescript** file for this exercise. (30 points)
For the commands which are not covered in this exercise, refer to the python pdb document (https://docs.python.org/3/library/pdb.html) or use help command to find out how to use them.

(This exercise is adopted from http://tjelvarolsson.com/blog/five-exercises-to-master-the-python-debugger/)

Note: source file for each one question (include test code for using your function).
C-4.9 (35 points) Write a recursive Python function that finds the minimum and maximum values in a sequence without using any loops.

C-4.19 (35 points) Write a short recursive Python function that re-arrange a sequence of integer values so that all the even values appear before all the odd number.