Recap for Practical 01 and Practical 02

Ian Wern Han, Lim
lim.wern.han@monash.edu

10 August 2014

Abstract

This is a supplementary document based on my evaluation of your Practical 01 and Practical 02. It is meant to highlight the good practices as examples for future work and also the mistakes I have noticed so that it do not repeat in the future. I hope this would be of help to you in the coming practicals.

Do note that this is just an extra thing that I do and I might not do it consistently on a weekly basis depending on my workload. I will however try my best to do this for every practical.

1 General Remarks

1.1 Reading and Understanding of Questions

There are quite a number of students who did not read the question properly. Do read each and every line carefully as it denotes the requirements for your practicals. For example, in Practical 02, many of you did not read between Task 01 and Task 02 where it stated that 2 test cases per menu command is required in Task 02 and Task 03.

1.1.1 Coming Practicals Expectations

As Practical 02 is the first assessed practical, I was lenient but in the coming practical, you will lose marks during evaluation for not meeting the requirements of the question.

1.2 Preparation

I have noticed that many did not prepare accordingly. Last minute works are fairly obvious in the unit where preparation is required. This is a unit which you have to sink time in order to do well (results scales with time spent). Thus do spend time preparing for the labs. If you prepared enough, you will have more time to ask questions during consultations or via emails.

1.2.1 Coming Practicals Expectations

Spend at least 4 hours preparation on the practical, preferable starting earlier such as on Monday itself so there is time to consult me (email) or your friends.

2 Python

Practical 01 and Practical 02 are basically simple practicals to get you used to Python and some practices (documentation, testing) behind it. From the pre-requisites of the unit (programming and algorithms), you are well equipped to pick up a new language on your own. Besides that, supplementary materials are also provided in Week 0 of Moodle which you are all expected to go through. If you haven, please go through the materials.

While Python may seem difficult, it is a fairly straight forward 4th generation programming language which is fairly close to our normal human language through abstraction. With sufficient practice, it will be second nature to you.

2.1 Getting Python to Work in Command Prompt (CMD)

This subsection is mainly catered for Windows users where you would need to add Python into your environment path (it generally gets handled automatically in Linux upon installation). The simplified steps are as follow:

1. Open command prompt (CMD). An easy way for this is to just go to the address bar and type **cmd**.
2. Add Python into your environment with the following command **setx Path <PyhonLocation>**. For example if I installed Python 3.4 in my computer in **C:\Python34** then my command would be **setx Path C:\Python34**.
3. Restart the command prompt as the previous one is using the old environment.

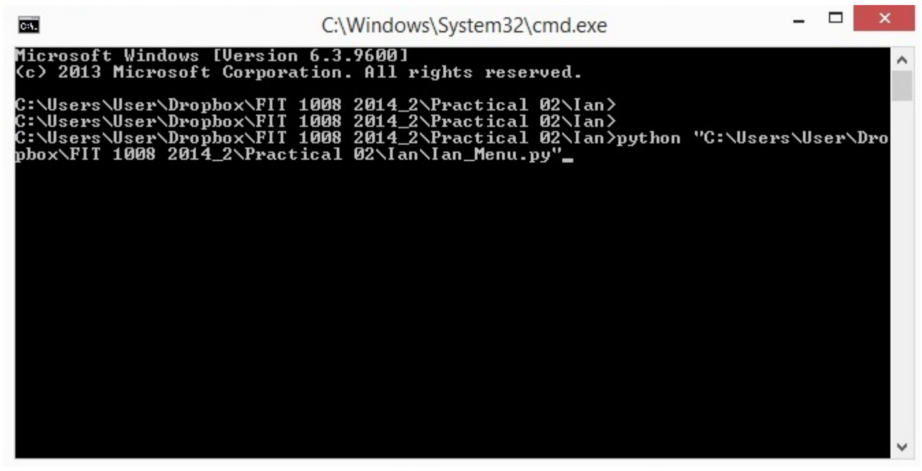
2.1.1 Running CMD vs Python

From the lab sessions, I noticed some of you are still confused what is being run in your command prompt. Thus you ended up giving the wrong commands. In the first Image 1. Notice how there is your current working directory before you are allowed to key in your command. The command that I have entered is running a Python file as shown in Image 1.

On the other hand, when you are running Python itself, you would need to invoke Python before hand. Your command then can be accepted after the >>> symbols as shown in Image 2.

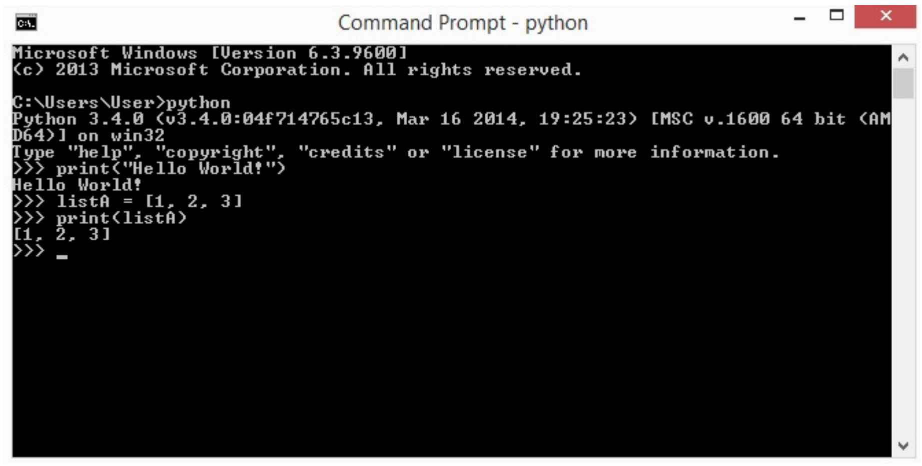
2.1.2 Why Command Prompt?

Being able to use the command prompt is one of the basic knowledge of a computer scientist or information technology professional especially when you are dealing with Linux-based systems. In your future units as well, you will be required to use it. Might as well start early.



```
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\User\Dropbox\FIT 1008 2014_2\Practical 02\Ian>
C:\Users\User\Dropbox\FIT 1008 2014_2\Practical 02\Ian>
C:\Users\User\Dropbox\FIT 1008 2014_2\Practical 02\Ian>python "C:\Users\User\Dro
pbox\FIT 1008 2014_2\Practical 02\Ian\Ian_Menu.py"_
```

Figure 1: Command Prompt and the commands. In this example, I am trying to run a Python file



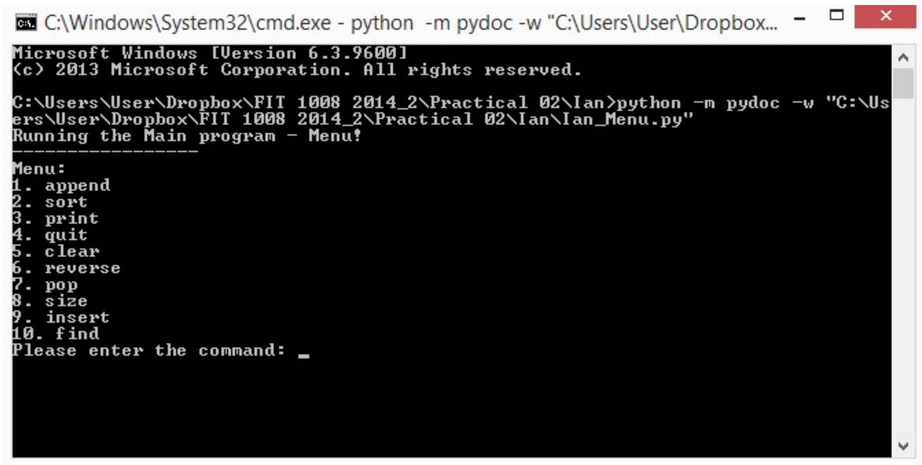
```
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\User>python
Python 3.4.0 (v3.4.0:04f714765c13, Mar 16 2014, 19:25:23) [MSC v.1600 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello World!")
Hello World!
>>> listA = [1, 2, 3]
>>> print(listA)
[1, 2, 3]
>>> _
```

Figure 2: Python running example

Besides that, it is a generalized approach for any machine regardless of whichever IDE that you are using. For example, some IDE are able built-in to run Pydoc while some are not. Example of Pydoc being run are as Image 3.

2.2 Examples of Python coding

I have attached several other files to provide additional example of Python. Do not that you still have to go through the Week 0 Moodle files. These are just extra supplementary materials.


C:\Windows\System32\cmd.exe - python -m pydoc -w "C:\Users\User\Dropbox...

```
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\User\Dropbox\FIT 1008 2014_2\Practical 02\Ian>python -m pydoc -w "C:\Us
ers\User\Dropbox\FIT 1008 2014_2\Practical 02\Ian\Ian_Menu.py"
Running the Main program - Menu!
----------------
Menu:
1. append
2. sort
3. print
4. quit
5. clear
6. reverse
7. pop
8. size
9. insert
10. find
Please enter the command: _
```

Figure 3: Pydoc running example: -m as a Pytho argument for module, -w as a Pydoc argument for write

2.2.1 Functions, Parameters, Returns and Variables

This example Python file (**Ian_Example_Python.py**) demonstrate how are Functions (or Methods) can be used in Python. Go through the example and take note of the following:

- Declaration of functions. It is always a better practice to code everything within a function and just calling it rather than coding everything to run in the Module itself due to the scripting nature of Python and to encourage re-usability.
- Passing of parameters into a function.
- Setting default value for parameters for cases which they are not passed.
- Returning values from functions. Notice how Python can return more than a single value.
- Encapsulation and locality of variables in a function. Notice how the values of variables changes? Note: more to come for the Nested List.

2.2.2 Exception Handling

Exceptions are runtime errors which will crash your running program. Thus, it is something that you would need to ensure to not occur. These errors can be discovered through proper testing (test cases) and through practice, you will be able to identify and use exceptions in Python. Once again, the reading material on exception is provided in Week 0 Moodle.

The attached file **Ian_Menu** is an example based on my simple implementation of Practical 02 Task 02. Error handling is a helpful tool for you work in the future. Things to notice in the file:

- The use of **try-except**. If the exception is known like **ValueError** you can include it and code how would you like to handle the exception. You can leave it without an exception type if you are unaware. Specific error handling allow you to handle exception on specific basis while without stating the type would invoke the exception handling regardless of errors.
- How are the exception handling coded into the code consistently.

Note: Go through this example file to look at how I would attempt Task 02. This is not the best solution from me but it provide a very good reference without me giving out a model solution.

2.2.3 Handling of Nested List

The example in **Ian_Example_NestedList.py** uses nested list to showcase how are some data types handled in Python especially in handling references and mutable data. The commends in the example code should probably suffice without me elaborating further.

3 Good Practices

Practical 01 and Practical 02 also aims to introduce and develop good coding practices amongst the students. This are also essentials when you graduate to work in the industry.

3.1 Documentation

I have gone through how documentation should be done during Practical 01 and likewise it is in the supplementary Week 0 Moodle file (docstring). Documentation is an important practice. The documentation of codes help you understand the code better (also others to understand your code) and note down key portions of your code (which would be useful for you when I evaluate you as well).

The documentation of your work can be anything up to you. However, there are some important documentations which you need to include especially the docstring for your functions (one for each function and one for the entire module/ file) as a requirement of your work. As also covered in your Practical 01, these includes (refer to the examples in **Ian_Menu.py**):

- Author: Who created it? Who modified it?
- Timestamps: Created and modified
- Description: What is it about
- Parameter: If any for the function
- Return: If any for the function
- Pre and Post-conditions: Generally conditions to be satisfied before running the function and the conditions resulting from the function. Read the file in Moodle Week 0 on PrePost Conditions for more detail.
- Complexity: You should have covered the basic of this in Week 02. Do note that having a loop does not always mean a O(N) complexity or a nested loop having a O(N**2) complexity. Likewise, you would need to state what are the parameters for example N is the length of a list? Refer to the provided example file for examples.

3.1.1 Coming Practicals Expectations

Complete and proper documentation. With the examples and materials provided, you should be able to provide a complete one now.

3.2 Testing and Test Cases

Test cases were one of the focus of Practical 02 with a large majority of the marks allocated to it being a key requirement (stated in the practical sheet as well) where you would require at least 2 distinct test cases per menu command. The main reasons for test cases are:

- Ensure that the code is working as intended. This mainly concern the logical errors where your expected output need to be the same as your actual output. For example during the lab, there are students that allow the empty space X to go beyond the boundary for Practical 02 Task 03 which is a wrong implementation. Besides that, some students only allow the user to append String but then only to search for Integers which is just inconsistent and they did not even realize it. Another logical error is allowing items to be **insert(index, item)** with an illegal index such as negative index which I am being lenient this lab as it is in fact wrong.
- Ensure that the code is free from runtime errors (exceptions). I am able to crash many of the students programs by something as simple as doing **pop()** on an empty list. This should not happen if you have properly tested your code. There are countless other exceptions which can caused in the codes which is effectively crashing your program and causing you to score really low marks. Exception errors should not occur and should handled as in Section 2.2.2.

There are some who script their testing. That is not required for now and I will not discuss it in this document.

3.2.1 Test Cases

What are test cases? For those who have took FIT 2001, they would have learnt it. Those who did not have the option of consulting me (only one consulted me before the lab) or their peers before hand so that they would not lose marks here. As the first assessed practical, I was being lenient on this but in the future I will adhere to the question requirements during evaluation.

Generally, how you write your test cases is pretty much up to you as long as you have the core of it. Even if you are unaware of test cases, you can get the answer easily by Google through keywords such as **test case**, **test case template** etc. It is up to you how would you format your test cases such as in a word document, in a table etc. What required (the core) of a test case are:

- Name or ID: The name or ID of the test case so it is identifiable.
- Description: Description about the test case such as your reason for running the test case.
- Steps: Basically you describe the steps taken to build and go through the test cases. For example if you are trying to test insertion of an element into a non-empty list, you would have to populate the list first and then insert the item.
- Expected output: What are the expected output to pass the test case.
- Actual/ observed output: What are the output you obtained running the steps of the test case.

- Result and remark: Did you pass (expected output consistent with the observed output) or fail the test case? Add in any comments if you wished.

The test cases need to be comprehensive enough to test all the possible scenarios associate with your code. For example, appending integer 1 and integer 2 to a list is in fact consider a single test case and on its own is not comprehensive to test out the **append(item)** function. Scenarios in this example include appending a string, appending into an empty list, appending into a non empty list and appending an item already found in the list (determine if your list allow duplicates).

The most common lack of testing in the students code from Practical 02 Task 02 is the **pop()** where it fails the popping operation on an empty list. What's worse is having it on your test case and it still failing to run properly in your code. You code is supposed to account for the test case and ensure that you pass the test case.

3.2.2 Coming Practicals Expectations

Include test cases if required in the practical sheet. If it is not in the practical sheet, make sure your codes are properly tested and the correctness of your code carries a lot of weight in your marks.