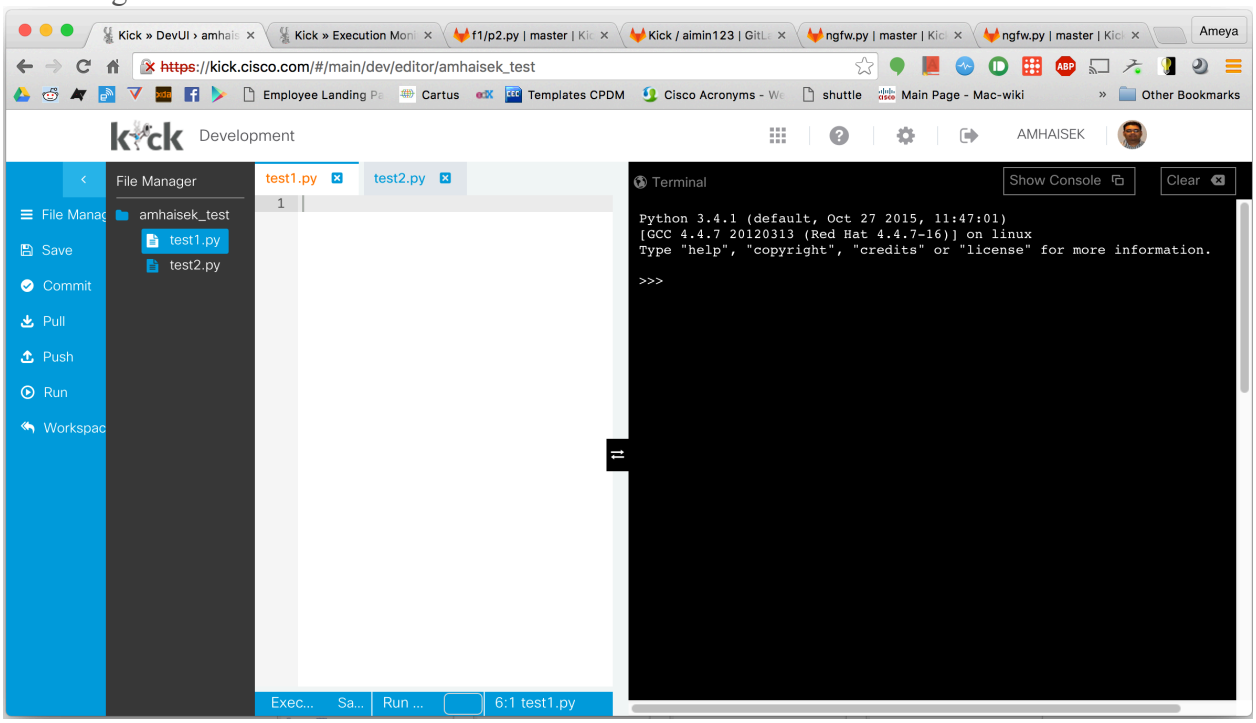
Basics of python and PyATS

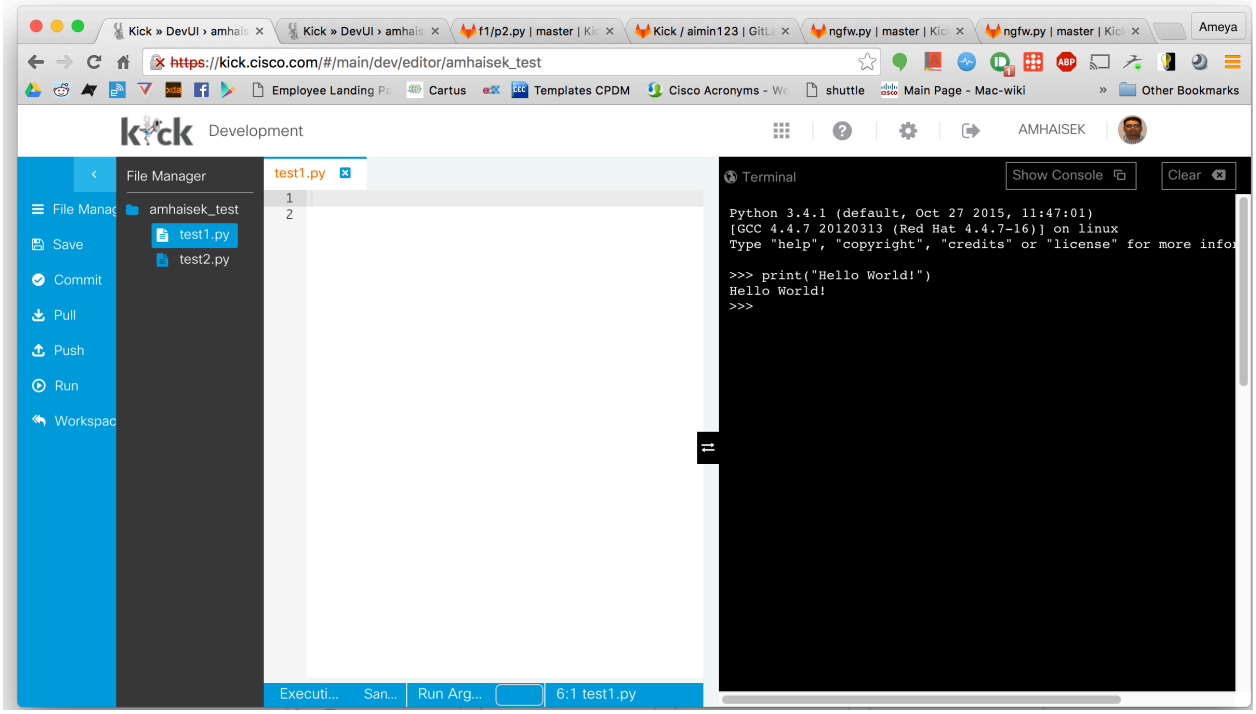
Python Basics for Kick!

1. Python is a high level scripting language with object oriented features. For a quick getting started refer to this **quick guide** and for basic command references checkout this **cheatsheet**.
2. This document would cover the basics of python use in reference to Kick! UI and terminal.
3. You can access the python Editor and terminal by going to https://kick.cisco.com --> Development. Once you select your workspace, you should now be able to see the following screen.

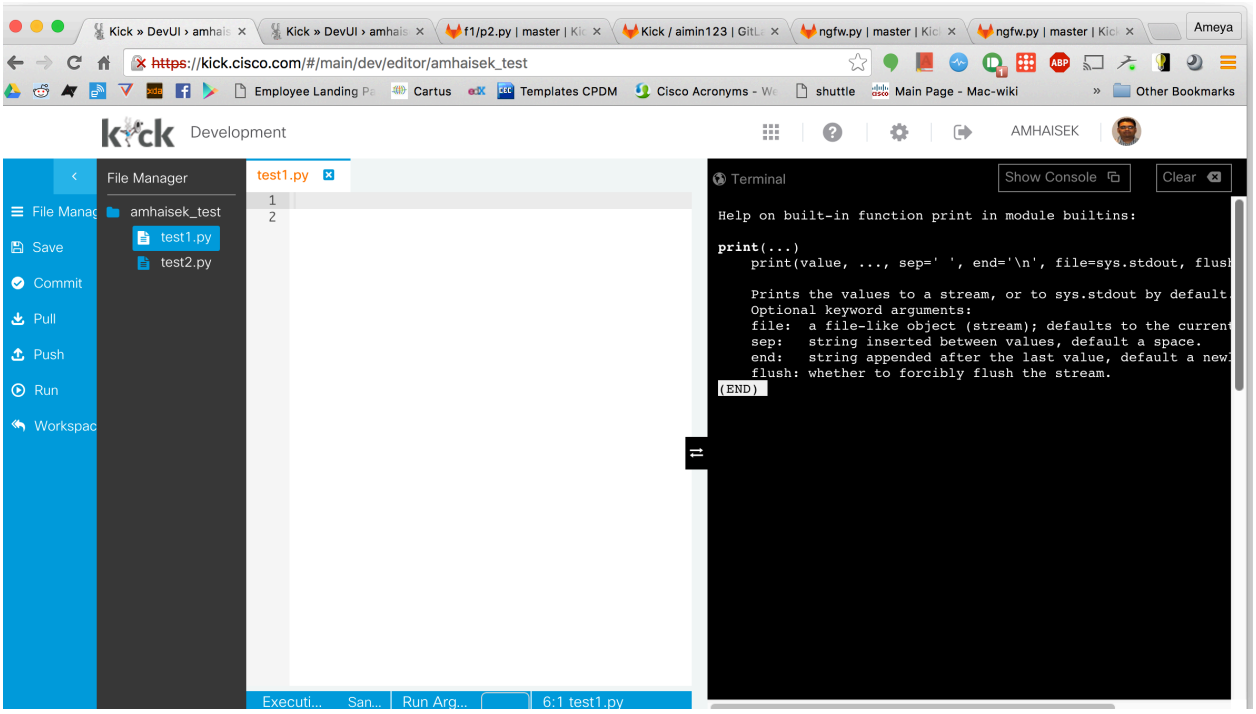


4. You can now write python code by creating a new file with **".py" extension** (python file) or editing one of the existing files.
5. On the right you can switch between the **console**, where you will see the output of running the file, and **terminal,** where you can write commands in python shell.
6. A simple way to test a python command is by running it first in the terminal. You can see the python and linux version for the execution environment on the terminal as shown in the image below. The terminal is also helpful in checking help, namespace and function details for runtime objects. For example, here is a simple command to print "Hello

World!".



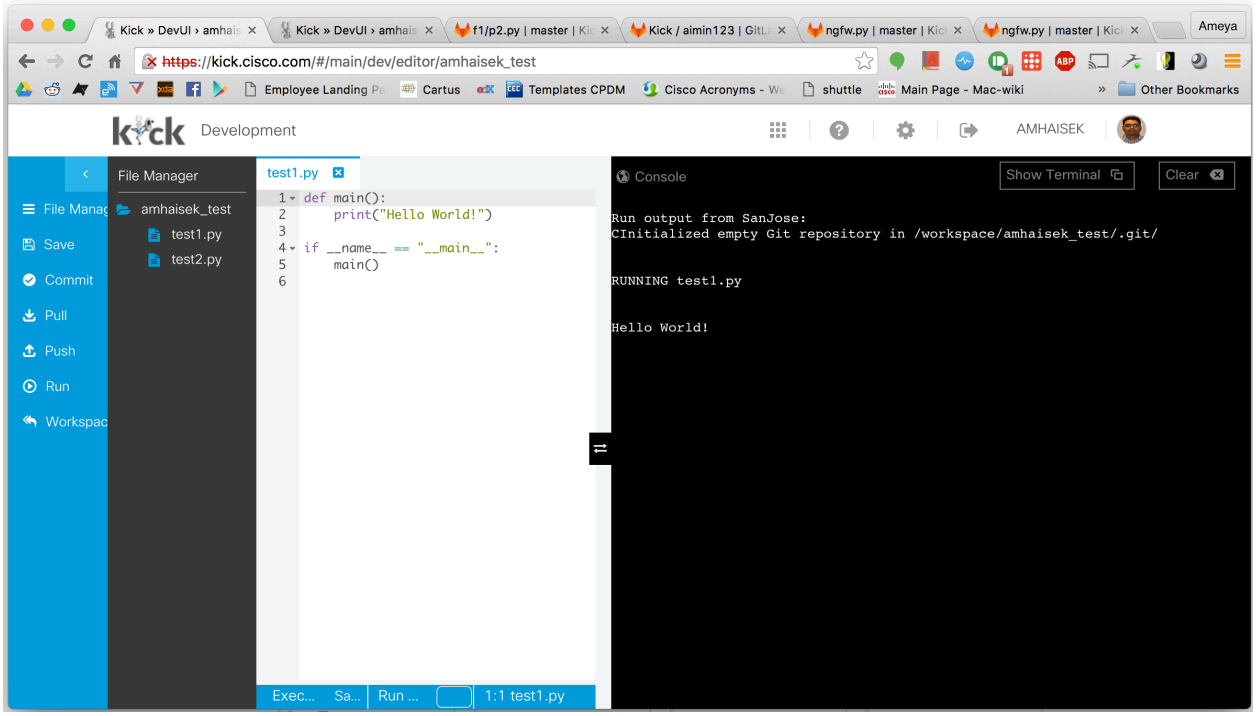
7. In the above example, you can run the print command next to the prompt ">>>" and see the output for the same in the next line.
8. You can get additional details on any python command/module by typing help(command/module). For example, to get the details on the print command type **help(print)** and you should see the output as shown below.



9. You can also run python commands by adding them in the editor to the file and then running it. Here is a simple python program to print Hello World! -

   Hello World!

10. In **Line 1**, we define a function called main() and we write the function details after the colon [:].
11. **Line 2** is indented to show that it belongs to function defintion of main(). This follows the standard convention of python to use whitespace as a means to mark boundaries and scope.
12. In **Line 4 and 5**, tell us that this file, (line 1-5, in our case test1.py) can be used as a module or as a stand-alone python program. In case it is imported as a module only the function definition would be inherited else it would also execute the print command.
13. You can run the file below by clicking on the file, and then clicking on the "Run" on the sidebar. You should now be able to see the output on the console.

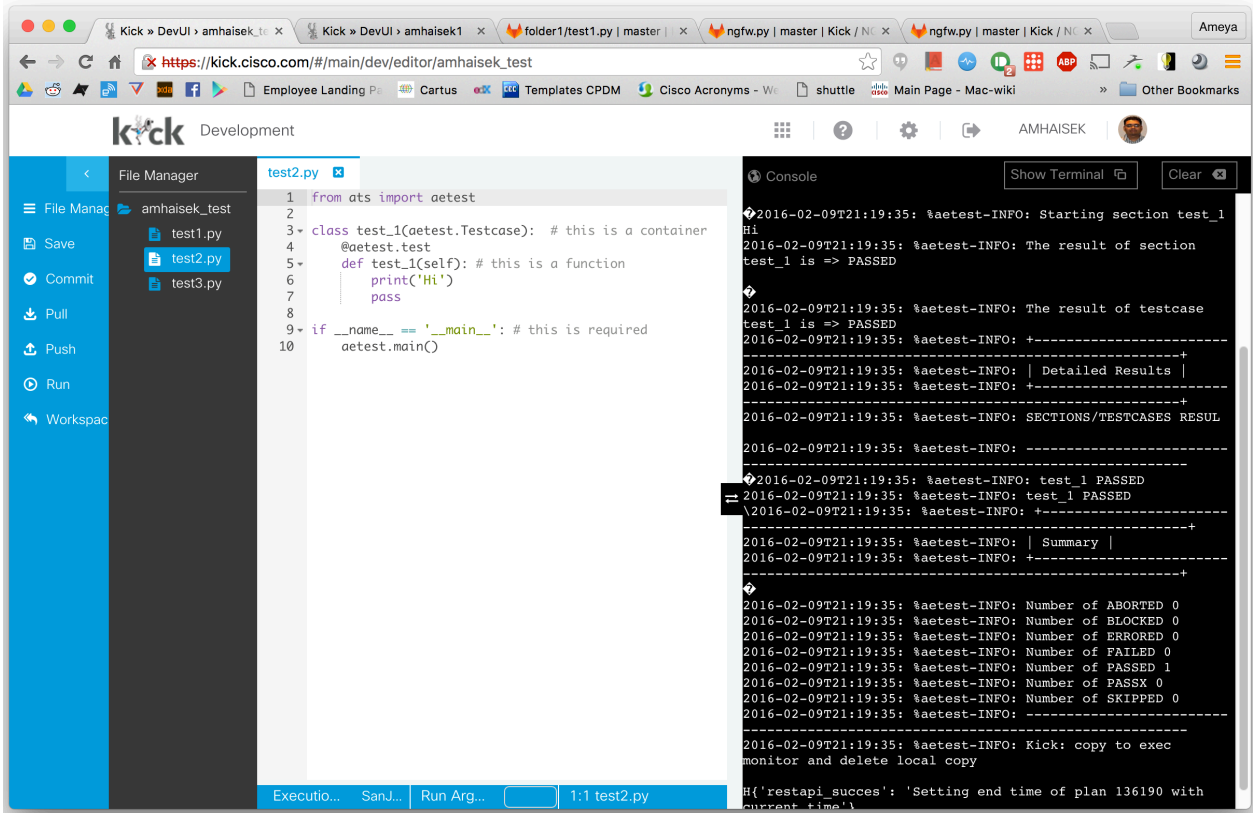


Basics of PyATS script

1. Shown below is a sample PyATS testscript.
2. AEtest (Automation Easy Testing) is the standard Cisco engineering test automation harness. AEtest is a standard component (aetest) in pyATS for standardizing the definition and execution of testcases & testscripts. It also adds new features such as logging as a standard option.
3. Typically a PyATS script would include the following key sections -

1. **Imports -** This includes importing any supporting modules and also import the definitions from aetest. This also include setting up any global variables such as for logging (Line 1 in the above script)
2. **Container/s -** This contains the definition for the test cases which include the functions to perform the tests (Line 3-7 in the above script).
3. **Script run -** This is the section that tells the script what to do when the file is imported as a module or when run as a program (Line 9-10 in the above script). This part is mandatory.

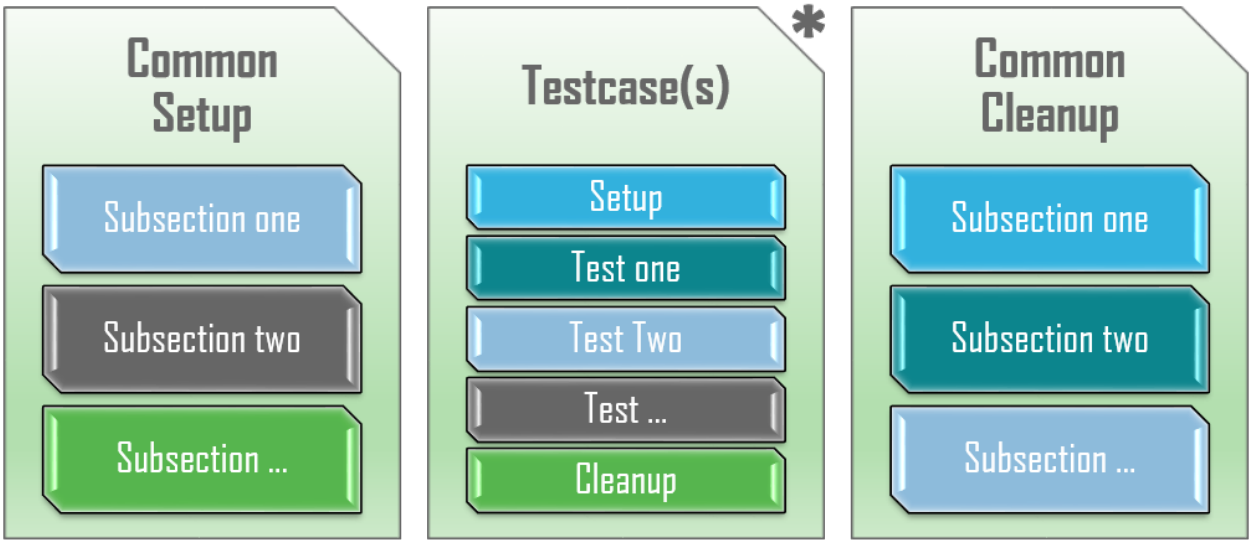
4. Here is the output for running this sample script on the kick UI.



5. The output for the testcase and status of the testcases can be seen on the console.
6. **Additional Notes regarding the program -**
    1. **aetest -** Is one of the most commanly used modules and will be used frequently as a standard for the running testcases.
    2. The classes (containers for the testcases) need to **inherit** from aetest classes, in order to inherit the common functionalities and register as test cases (**aetest.Testcase**).
    3. The **@aetest.test** registers the functions as a test that is to be run.
    4. The call for **aetest.main()** mandatory as this tells the interpreter to run the script.

PyATS script Structure

1. We will be following the standard AEtest test script structure for writing test cases. A typical script would is split into three major container sections which are then further broken down into smaller, method sections (as shown in the image). These are explained in detail as below



2. **Common Setup -**
    1. This is where all the common configurations, prerequisites and initializations shared between the script's testcases should be performed. **`CommonSetup`** is always run first, before all testcases.
    2. This ensures that the test cases have all the necessary configuration and initialization setup.
    3. **`CommonSetup`** is an optional container section within each testscript. It is defined by inheriting the **`aetest.CommonSetup`** class, and declaring one or more subsections inside.
    4. **`CommonSetup`** section is unique within each testscript: only one may be defined, and regardless of the class name used, its reporting/result id will always be `common_setup`.
    5. **@aetest.subsection** is used to decorate definitions of functions that demarcate a subsection. Splitting the content into separate functions helps in better organization of code.

3. **Common Cleanup -**
    1. **`CommonCleanup`** is the last section to run within each testscript. Any configurations, initializations and environment changes that occured during this script run should be cleaned up (removed) here.
    2. **`CommonCleanup`** is an optional container section within each testscript. It is defined by inheriting the **`aetest.CommonCleanup`** class, and declaring one or more subsections inside.

3. **`CommonCleanup`** section is unique within each testscript: only one may be defined, and regardless of the class name used, its reporting/result id will always be `common_cleanup`.
4.

4. **Testcase(s) -**
    1. **`Testcase`** is a container/collection of smaller tests. Testcases are the workhorse of every testscript, carrying out the assessments that determines the quality of the product under scrutiny.
    2. Each Testcase is defined by inheriting **aetest.Testcase** class, and defining one or more **Test Sections** inside. Optionally, each Testcase may also have a single **Setup Section** and a single **Cleanup Section**.
    3. Testcases are run in the order as they are defined/appear in the testscript.
    4. Testcases are unique: each `Testcase` is associated with a unique ID. This defaults to the testcase's class name, and can be changed by setting the **`Testcase.id`** attribute. This testcase ID will be used for result reporting purposes.
    5. **`Testcase`** are independent: the testing code of a **`Testcase`** instance should be entirely self-contained, such that it can be run either in isolation or in arbitrary combination with any number of testcases. Each testcase should test a unique aspect of the product, is self-reliant, and its result can be separated from all other testcases.

5. **Subsections -**
    1. Subsections are the bricks-and-mortars that make up `CommonSetup` and `CommonCleanup`. Within these class definitions, any methods decorated with `@subsection` decorator is marked to be a subsection. Each subsection should be an identifiable action to be completed as part of the greater section.
    2. When a `CommonSetup` or `CommonCleanup` class method is decorated with `@subsection`, the corresponding method name is used as the subsection name for result reporting.
    3. Subsections are indepedent: each subsection will run regardless of any previous section's result. The control of whether to abort/skip/continue after an unexpected result is entirely in the hands of the user.

6. **Sections under a Testcase -**
    1. **Setup Section -**
        1. **`setup`** is an sub-division section, available for **`Testcase`**. It can be used to perform all the common configuration, prerequisites and initializations specific to that testcase.
        2. **`setup`** section is defined by decorating a **`Testcase`** class method with **`@aetest.setup`** decorator. It is optional to each testcase: if defined, it will always run before all other sections.
        3. **`setup`** is unique: each **`Testcase`** may only have one method decorated to be its setup section. Regardless of this method's function name, its reporting/result id will always be named **`setup`**.

2. **Test Section -**
    1. **`test`** sections are the smallest units of testing and the most basic building block that makes up **`Testcase`**. Each **`test`** should carry out a single identifiable check/evaluation to be completed as part of the greater section.
    2. **`test`** section is defined by decorating a **`Testcase`** class method with **`@aetest.test`** decorator. The corresponding method name is used as the test name for result reporting. Each testcase must have at least one or more `test` section.
    3. **`test`** sections normally run in the order of definition, and will always run regardless of previous test section results.
3. **Cleanup Section -**
    1. **`cleanup`** is the last sub-division section within each **`Testcase.`** Any configurations, initializations & changes that occured during this testcase should be cleaned up (removed) here.
    2. **`cleanup`** section is defined by decorating a **`Testcase`** class method with **`@aetest.cleanup`** decorator. It is optional to each testcase: if defined, it will always run after all other sections.
    3. **`cleanup`** is unique: each **`Testcase`** may only have one method decorated to be its cleanup section. Regardless of this method's function name, its reporting/result id will always be named `cleanup`
    4. **Note** - that `cleanup` section should be catch-all: regardless of whether all `tests` before it passed or failed, it should be still able to return the environment to its original state.

7. Please refer to the Sample pyATS test script to see a sample script.

References -

- http://wwwin-pyats.cisco.com/documentation/html/aetest/structure.html

PyATS Sample Script

1. Here is a sample test script (Aimin's git repo with sample testcases - http://gitlab.cisco.com/kick/aimin1234) with detailed comments on each section -

```
#!/bin/env python
##################################################################
# basic_example.py : A very simple test script example which include:
#     common_setup
#     Tescases
#     common_cleanup
# The purpose of this sample test script is to show the "hello world"
# of aetest.
##################################################################

# To get a logger for the script
import logging
import sys

# Needed for aetest script
from ats import aetest

# Get your logger for your script
log = logging.getLogger(__name__)
log.setLevel(logging.INFO)

##################################################################
###                  COMMON SETUP SECTION                      ###
##################################################################

# This is how to create a CommonSetup
# You can have 0 or 1 CommonSetup
# CommonSetup can be named whatever you want
class common_setup(aetest.CommonSetup):
    """ Common Setup section """

    # CommonSetup have subsection.
    # You can have 1 to as many subsection as wanted
    # here is an example of 2 subsections

    # First subsection
    @aetest.subsection
    def sample_subsection_1(self):
        """ Common Setup subsection """
        log.info("Aetest Common Setup ")

    # If you want to get the name of current section,
    # add section to the argument of the function.

    # Second subsection
    @aetest.subsection
    def sample_subsection_2(self, section):
        """ Common Setup subsection """
        log.info("Inside %s" % (section))

        # And how to access the class itself ?

        # self refers to the instance of that class, and remains consistent
        # throughout the execution of that container.
        log.info("Inside class %s" % (self.id))
```

```
54
55 ##################################################################
56 ###                     TESTCASES SECTION                        ###
57 ##################################################################
58
59 # This is how to create a testcase
60 # You can have 0 to as many testcase as wanted
61
62 # Testcase name : tc_one
63 class tc_one(aetest.Testcase):
64     """ This is user Testcases section """
65
66     # Testcases are divided into 3 sections
67     # Setup, Test and Cleanup.
68
69     # This is how to create a setup section
70     @aetest.setup
71     def prepare_testcase(self, section):
72         """ Testcase Setup section """
73         log.info("Preparing the test")
74         log.info(section)
75
76     # This is how to create a test section
77     # You can have 0 to as many test section as wanted
78
79     # First test section
80     @ aetest.test
81     def simple_test_1(self):
82         """ Sample test section. Only print """
83         log.info("First test section ")
84
85     # Second test section
86     @ aetest.test
87     def simple_test_2(self):
88         """ Sample test section. Only print """
89         log.info("Second test section ")
90
91     # This is how to create a cleanup section
92     @aetest.cleanup
93     def clean_testcase(self):
94         """ Testcase cleanup section """
95         log.info("Pass testcase cleanup")
96
97 class tc_two(aetest.Testcase):
98     """ This is user Testcases section """
99
100    # Testcases are divided into 3 sections
101    # Setup, Test and Cleanup.
102
103    # This is how to create a setup section
104    @aetest.setup
105    def prepare_testcase(self, section):
106        """ Testcase Setup section """
        log.info("Preparing the test")
        log.info(section)
```

```
107
108     # This is how to create a test section
109     # You can have 0 to as many test section as wanted
110
111     # First test section
112     @ aetest.test
113     def simple_test_1(self):
114         """ Sample test section. Only print """
115         log.info("First test section ")
116
117     # Second test section
118     @ aetest.test
119     def simple_test_2(self):
120         """ Sample test section. Only print """
121         log.info("Second test section ")
122
123     # This is how to create a cleanup section
124     @aetest.cleanup
125     def clean_testcase(self):
126         """ Testcase cleanup section """
127         log.info("Pass testcase cleanup")
128 ##################################################################
129 ####                       COMMON CLEANUP SECTION                 ###
130 ##################################################################
131
132 # This is how to create a CommonCleanup
133 # You can have 0 , or 1 CommonCleanup.
134 # CommonCleanup can be named whatever you want :)
135 class common_cleanup(aetest.CommonCleanup):
136     """ Common Cleanup for Sample Test """
137
138     # CommonCleanup follow exactly the same rule as CommonSetup regarding
139     # subsection
140     # You can have 1 to as many subsection as wanted
141     # here is an example of 1 subsections
142
143     @aetest.subsection
144     def clean_everything(self):
145         """ Common Cleanup Subsection """
146         log.info("Aetest Common Cleanup ")
147
148 if __name__ == '__main__': # pragma: no cover
149     aetest.main()
```