# Cisco pyATS

## Network Test and Automation Solution

Data-driven and reusable testing for modern networks

JOHN CAPOBIANCO
DAN WADE

ciscopress.com

# Cisco pyATS—Network Test and Automation Solution

## Data-driven and reusable testing for modern networks

John Capobianco

Dan Wade

**Cisco Press**

221 River Street

Hoboken, NJ 07030 USA

# Cisco pyATS—Network Test and Automation Solution

John Capobianco, Dan Wade



Published by:
Cisco Press



$PrintCode

Library of Congress Control Number: 2024907000

ISBN-13: 978-0-13-803167-1

ISBN-10: 0-13-803167-3

## Warning and Disclaimer

This book is designed to provide information about all aspects of Cisco pyATS. Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied.

The information is provided on an "as is" basis. The authors, Cisco Press, and Cisco Systems, Inc. shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book or from the use of the discs or programs that may accompany it.

The opinions expressed in this book belong to the authors and are not necessarily those of Cisco Systems, Inc.

## Trademark Acknowledgments



## Special Sales

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

## Feedback Information

At Cisco Press, our goal is to create in-depth technical books of the highest quality and value. Each book is crafted with care and precision, undergoing rigorous development that involves the unique expertise of members from the professional technical community.

Readers' feedback is a natural continuation of this process. If you have any comments regarding how we could improve the quality of this book, or otherwise alter it to better suit your needs, you can contact us through email at feedback@ciscopress.com. Please make sure to include the book title and ISBN in your message.

We greatly appreciate your assistance.

Please contact us with concerns about any potential bias at https://www.pearson.com/report-bias.html.

**GM K12, Early Career and Professional Learning:** Soo Kang

**Alliances Manager, Cisco Press:** Caroline Antonio

**Director, ITP Product Management:** Brett Bartow

**Executive Editor:** Nancy Davis

**Managing Editor:** Sandra Schroeder

**Development Editor:** Christopher Cleveland

**Senior Project Editor:** Mandie Frank

**Copy Editor:** Bart Reed

**Technical Editors:** Stuart Clark, Charles Greenaway

**Editorial Assistant:** Cindy Teeters

**Designer:** Chuti Prasertsith

**Composition:** codeMantra

**Indexer:** Timothy Wright

**Proofreader:** Barbara Mack

Americas Headquarters
Cisco Systems, Inc.
San Jose, CA

Asia Pacific Headquarters
Cisco Systems (USA) Pte, Ltd.
Singapore

Europe Headquarters
Cisco Systems International BV Amsterdam,
The Netherlands

Cisco has more than 200 offices worldwide. Addresses, phone numbers, and fax numbers are listed on the Cisco Website at **www.cisco.com/go/offices.**

## About the Authors

**John Capobianco** has a dynamic and multifaceted career in IT and networking, marked by significant contributions to both the public and private sectors. Beginning his journey in the field as an aluminum factory worker, Capobianco's resilience and dedication propelled him through college, earning a diploma as a Computer Programmer Analyst from St. Lawrence College. This initial phase set the foundation for a career underpinned by continuous learning and achievement, evident from his array of certifications, including multiple Cisco certifications as well as Microsoft certification.

Transitioning from his early educational accomplishments, Capobianco's professional life has spanned over two decades, featuring roles that showcased his technical prowess and strategic vision. His work has significantly impacted both the public and private sectors, including notable positions at the Parliament of Canada, where he served as a Senior IT Planner and Integrator, and at Cisco, where he began as a Developer Advocate. These roles have been instrumental in shaping his perspective on network management and security, leading to his recent advancement into a Technical Leader role in Artificial Intelligence for Cisco Secure, reflecting his commitment to integrating AI technologies for enhancing network security solutions.

In addition to his professional and technical achievements, Capobianco is also an accomplished author. His book *Automate Your Network: Introducing the Modern Approach to Enterprise Network Management*, published in March 2019, encapsulates his philosophy toward leveraging automation for efficient and effective network management. He is dedicated to lifelong learning and professional development, supported by a solid foundation in education and a broad spectrum of certifications, and now aims to share his knowledge with others through this book, YouTube videos, and blogs. John can be found on X using @john_capobianco.

**Dan Wade** is a Network and Infrastructure Automation Practice Lead at BlueAlly. As part of the Solutions Strategy team at BlueAlly, he is responsible for developing network and infrastructure automation solutions and enabling the sales and consulting teams on delivery of the developed solutions. Solutions may include infrastructure provisioning, configuration management, network source of truth, network observability, and, of course, automated testing and validation. Previous to this role, Dan worked as a consulting engineer with a focus on network automation.

Dan has more than seven years of experience in network automation, having worked with automation tooling and frameworks such as Ansible and Terraform, and Python libraries, including Nornir, Netmiko, NAPALM, Scrapli, and Python SDKs. Dan has been working with pyATS and the pyATS library (Genie) for the past four to five years, which has inspired him to embrace automated network testing. In 2021, Dan contributed to the genieparser library with a new IOS XE parser. He also enjoys creating his own open-source projects focused on network automation. Dan holds two professional-level certifications from Cisco: Cisco DevNet Professional and CCNP Enterprise. He is also a member of the Cisco DevNet 500 and Cisco Champions program.

Dan enjoys sharing knowledge and experience on social media with blog posts and YouTube videos as well as participating in podcast episodes. He's passionate about helping others explore network automation and advocating how network automation can empower, not replace, network engineers. You can find him on social media @devnetdan.

Chapter 4

# AEtest Test Infrastructure

You may have heard about testing code. Code testing allows you to verify your code produces the results you're expecting. This is important, as it helps *minimize*, not remove, bugs in your code. Code testing also encapsulates the idea of regression testing. In the simplest terms, regression testing ensures new code updates do not introduce new bugs. Regression testing becomes more important as the codebase grows. The last concept you may hear about when it comes to testing code is code coverage. Code coverage is the amount of code in your codebase that is "covered" by a test. Many times, it's assumed that more code coverage equals fewer bugs; however, that's simply not true. Tests are only as good as they are written. If your tests are poorly written, then no amount of code coverage can save you from bugs.

Two of the most popular Python testing frameworks are unittest and pytest. unittest is included in the Python standard library and does not require any additional installation. pytest, on the other hand, is a separate library and requires installation. unittest and pytest are different in their own ways, but both have the same goal of allowing developers to write tests and verify their code is running as expected. Now substitute the word "code" with "network"—*write tests and verify your network is running as expected*. Doesn't that sound amazing? This defines Automation Easy Testing (AEtest)—the testing framework for the network.

In this chapter, we will cover the following topics:

- Getting started with AEtest
- Testscript structure
- AEtest object model
- Runtime behavior
- Test parameters
- Test results

- Running testscripts
- Processors
- Testscript flow control
- Reporting
- Debugging

## Getting Started with AEtest

The goal of AEtest is to standardize the definition and execution of testcases against the network. In this section, we are going to cover the basics: ensuring the AEtest module is installed and reviewing the design features and core concepts of the framework.

### Installation

The AEtest module is included as part of the default pyATS installation. To ensure the module is installed, run the **pip list** command within your Python environment. In the list of installed packages, you should see **pyats.aetest** listed, along with many other pyATS modules. If you don't see the pyats.aetest module listed, I recommend reinstalling pyATS as described in Chapter 2, "Installing and Upgrading pyATS."

### Design Features

AEtest drew its design from two popular Python testing tools: unittest and pytest. If you're familiar with either library, the structure and design of AEtest may be familiar. With that said, let's review the design features of AEtest.

AEtest is built with a Pythonic object-oriented approach. This comes from the infamous object-oriented programming (OOP) programming paradigm, where the design is centered around classes and objects rather than functions. From a network perspective, think about all the individual components that make up a network—interfaces, links, devices, and so on. These are all considered "objects" and can be implemented as such in Python using an OOP approach. Moving on, another design feature is using a block-based approach to test sections. We are going to review each test section, but here's a quick breakdown of the approach:

- Common Setup with subsections
- Testcases with setup/tests/cleanup
- Common Cleanup with subsections

Each block listed has a purpose that will be explained further in the chapter. The next design feature is two-fold. AEtest was built to be highly modular and extensible, which, in turn, allows testcase inheritance, dynamic testcase generation, customer runner for

testable objects, and a customizable reporter. The last design feature is what allows the tool to scale and cover multiple network use cases. AEtest provides enhanced looping and testcase parametrization. Enhanced looping allows the same test(s) to be reused with different parameters. This is huge, as looping cuts down on the need to write multiple tests that only require slight variation. Looping and testcase parametrization will be covered in further detail later in the chapter. I hope going through these design features helps set expectations and creates a mold for AEtest. Next, let's look at some of the core concepts of AEtest.

### Core Concepts

The core concepts of AEtest are brief but promote boundaries of the framework. Here are the three core concepts:

- Main sections must be subdivided
- Sections must be explicitly declared
- Import, inspect, and run

Subdividing the main sections enhances the readability of the code. You can also quickly identify the section that failed in results. Imported sections (that is, testcases) must be inherited in the script for them to be included. Inheriting an imported section explicitly tells AEtest to include the imported testcase. Simply importing the testcase into the script does nothing. The last core concept (import, inspect, and run) is interesting and requires a little more explanation. When a testcase or testscript is imported, Python discovers the test sections (classes), which are instantiated and then run by AEtest. This might seem confusing, as you probably expect any test section that is imported, or included in a testscript, to run in the order provided, but that is not the case. The discovery process is similar to how pytest and unittest discovers testcases and will be covered in more detail further in the chapter. With the design features and core concepts covered, let's move on to the structure of testscripts.

## Testscript Structure

Testscripts are the foundation to the AEtest test infrastructure. Testscripts are made up of three main "containers": Common Setup, Testcases, and Common Cleanup. Each container is a Python class with smaller sections (subsections, setup, test, and cleanup) that are defined as methods within the container class. Each method is decorated with a Python decorator that identifies the section type. Python decorators modify the behavior of functions. They can be complex to understand, but they work by passing a function into another function as an argument. The returned output of the function that's passed in as an argument is modified, which ultimately modifies the function's behavior. In AEtest, the different decorators help identify the execution order and resulting rollup of each section based on the decorator. We will dive into these smaller sections later in this chapter, but let's begin by reviewing the main containers.

## Common Setup

The Common Setup container is where pyATS connects to testbed devices, applies base configuration for testing, and other initialization actions. This container essentially sets the stage for testing. Common Setup is not a required container in your testscript, but it's highly recommended. If Common Setup is defined, it will always run first. This is due to the discovery process performed by AEtest, which we will take a look at later on. One key feature of Common Setup is that it's also used to validate any script inputs (arguments) provided to the testscript. This allows your testscript to fail fast before going too far into testing before realizing one of the script inputs is incorrect. To better organize your code, you can break down Common Setup into multiple subsections. Each subsection should perform a specific task—for example, one subsection for connecting to testbed devices, another subsection for applying base configurations, and so on. The goal of the Common Setup container is to house the code that prepares your testbed devices for testing, whether that be connectivity, configuration, or operational state.

### Subsection

Subsections are smaller actionable sections that make up Common Setup and Common Cleanup. Subsections can be seen as independent, as results from a previous subsection do not affect execution of the current subsection. The user can control whether to skip, abort, or continue to the next subsection after an unexpected result. The results of a subsection are rolled up to the parent section (Common Setup/Common Cleanup). Example 4-1 shows what a Common Setup section with two subsections might look like in a testscript.

**Example 4-1** *Common Setup*

```
from pyats import aetest

class MyCommonSetup(aetest.CommonSetup):
      """Common Setup"""

      @aetest.subsection
      def connect_to_devices(self):
      """Code to connect to testbed devices"""
      pass

      @aetest.subsection
      def apply_base_config(self):
      """Code to configure devices with base/initial config"""
      pass
```

## Testcases

The testcase container is made up of smaller tests and is the focal point of testscripts. Testcases are designed to be self-contained, modular, and extensible, which allows network engineers to build a library of testcases for their network testing needs. Testcases can have their own setup and cleanup sections, with an arbitrary number of test sections. Each testcase has a unique identifier (UID), which defaults to the testcase name, which is used for result reporting and other job artifacts. Testcases are run as they are defined in the testscript.

### Setup Section

The setup section within a testcase is optional. If it's defined, there can only be one setup section within a testcase, and it is automatically run before any other sections. If the setup section fails, all test sections within the testcase would be "blocked" from running. The purpose of the setup section is to configure/enable specific features being tested in a particular testcase. The setup section result is rolled up to the parent testcase result.

### Test Section

The test section is a basic building block of testcases. Test sections define the tests run against the network. Each test should test for one specific, identifiable objective—don't try to stuff too much logic or checks within one test section! They are run in the order in which they are defined in the testcase. All test results are rolled up to the parent testcase result.

### Cleanup Section

The cleanup section is an optional section, like the setup section, within a testcase. It removes all configurations and/or features enabled during the setup and test sections of the testcase. Whether tests pass or fail, the goal of the cleanup section is to return the testbed devices back to the same state they were in before the testcase. This allows the testscript to continue executing without any lingering issues from previous testcase manipulation. The cleanup section result is rolled up to the parent testcase result.

Now that we have discussed testcases, and the individual sections, let's take a look at an example. Example 4-2 shows code scaffolding for a testcase with a setup section, two test sections, and a cleanup section.

**Example 4-2** *Testcase*

```
from pyats import aetest

class MyTestcase(aetest.Testcase):
      """Testcase"""
```

```
@aetest.setup
def testcase_setup(self):
    """Code to setup testbed devices for testcase"""
    pass

@aetest.test
def test1(self):
    """Code for first test"""
    pass

@aetest.test
def test2(self):
    """Code for second test"""
    pass

@aetest.cleanup
def testcase_cleanup(self):
    """Code to cleanup config on testbed devices"""
    pass
```

### Common Cleanup

Common Cleanup is much like the cleanup section defined for testcases, but it applies to the entire testscript. It is not required in a testscript, but it's highly recommended. The Common Cleanup is always the last section to run, after all testcases, and removes any configuration and environment changes that occurred during the testscript run. You can think of it as reversing the actions that were done in Common Setup. Like Common Setup, Common Cleanup can be broken down into subsections, which define specific actions. The goal of the Common Cleanup section is to reset the state of the testbed back to what it was before the testscript run. The Common Cleanup result is a combined rollup of results of all of its subsections.

To wrap up, take a look at Figure 4-1, which helps visualize the testscript structure with the different containers and their corresponding sections.

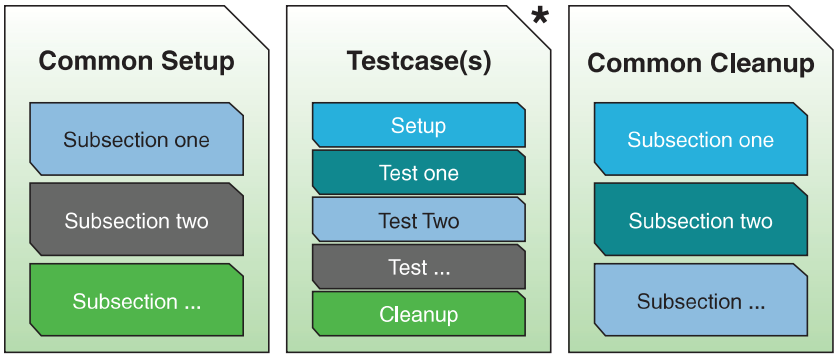


**Figure 4-1** *AEtest Testscript Structure*

## Section Steps

Previously, we discussed how container classes (Common Setup, Testcases, and Common Cleanup) are broken down into smaller sections to help better organize code and the overall testing workflow. However, we can go one step further. *Steps* allow you to break down your individual test sections into more granular actions. Steps are completely optional and should only be used if a test is larger and it makes sense to break it down further versus separating it out into smaller, individual tests.

**Steps** is a reserved parameter in the AEtest infrastructure and must be included as a test function argument in order to be used within the test. The **Steps** object is a Python context manager and is intended to be used by the **with** statement. Example 4-3 shows a simple example of a test section within a testcase broken down into multiple steps.

**Example 4-3** *Steps in Test Section*

```
from pyats import aetest
class Testcase(aetest.Testcase):
          """Testcase with steps"""

          @aetest.test
          def test(self, steps):
          """Code for test section and steps"""
              # steps.start() begins the step
              with steps.start("The first step") as step:
                print("This is step one!")

              with steps.start("The second step") as step:
                print("This is step two!")
```

There are a couple of key points to point out from the example. The step begins with **steps.start()**, which contains a description of the step within the parentheses. The **Steps** object is really implemented as two internal classes. The **Steps** class is considered the base container class and allows the creation, reporting, and handling of multiple nested steps. The **Step** class inherits the base **Steps** class and is meant to be used as a context manager. We can access current step information with the variable name we set after the **as** keyword. For example, steps have an "index" attribute that can be accessed in the step via **step.index**. Table 4-1 shows the complete list of attributes and properties of the **Steps** and **Step** objects.

**Table 4-1** *The steps and step Attributes*

| **Steps** | |
|---|---|
| **Attribute** | **Description** |
| start | Starts a new step. Returns a **Step** instance. |
| result | Rollup result of all steps contained. |
| report | Reports current step details/results to a log file. |
| steps | List of **Step** objects representing each step taken. |
| **Properties** | **Description** |
| details | List of step details using **StepDetail namedtuple**. |
| **Step (basecls: Steps)** | |
| **Attribute** | **Description** |
| start | Starts a new child step. Returns a **Step** instance. |
| result | Rollup result of this step and all child steps contained. |
| report | Reports current step details/results to a log file. |
| steps | List of child **Step** objects. |
| description | Description of this step instance. |
| **Properties** | **Description** |
| details | List of step details using **StepDetail namedtuple**. |
| **Result APIs** | **Description** |
| passed | Provides a passed result to this step. |
| failed | Provides a failed result to this step. |
| aborted | Provides an aborted result to this step. |
| blocked | Provides a blocked result to this step. |
| skipped | Provides a skipped result to this step. |
| errored | Provides an errored result to this step. |

| **Steps** | |
|---|---|
| passx | Provides a passx result to this step. |
| **Built-in** | **Description** |
| __enter____ | Method called with starting step through **with** statement. |
| __exit____ | Method called with exiting step through **with** statement. |

By inheriting the base **Steps** class, the **Step** object is able to nest steps, which provides more granularity during testing. Example 4-4 shows nested steps and the associated output, which expresses how the nested steps show up in the printed results. You'll notice the nested steps are separated from the parent step using "".

**Example 4-4** *Nested Steps*

```
from pyats import aetest

class Testcase(aetest.Testcase):
    @aetest.test
    def test(self, steps):
        # demonstrating a step with multiple child steps
        with steps.start("test step 1") as step:
            with step.start("test step 1 substep a"):
                pass
            with step.start("test step 1 substep b") as substep:
                with substep.start("test step 1 sub-step b sub-substep i"):
                    pass
                with substep.start("test step 1 sub-step b sub-substep ii"):
                    pass
```

The results for each step roll-up to the parent test section. Let's use the previous example (Example 4-4) as an example. If Step 1.b.ii was the only step to fail, the entire test section, which includes the other four steps, would have a "Failed" result. That's why it's crucial to only include related test steps within a single test section. If one fails, the entire section is considered a failure. The roll-up nature of test results will be covered later in the chapter. For reporting, a testscript creates a steps report at the end of each test section that contains steps. During runtime, the steps report can be accessed with the **report()** attribute (**steps.report()**). For additional detail, the **details** attribute can be accessed (**steps.details**),

which will return a list of StepDetail objects. Each StepDetail object is a named tuple containing the current step index, step name, and step result. The details attribute can be useful if you plan to parse and analyze the step results further. Steps are useful when needing to break down a lengthy test into smaller, granular chunks.

## AEtest Object Model

An object model describes the classes and objects that make up a piece of software or system. In this section, we are going to go through the object model of AEtest. This will get into the implementation details of the classes that make up the testscript and different testscript sections previously described. If you are brand new to pyATS and Python, this section may be advanced, but try to stick with it! There's a lot of detail that explains how testscripts and their individual sections are constructed and why they behave the way they do.

### TestScript Class

An AEtest testscript is a standard Python file, but without the .py extension. The file is considered a testscript because it imports the AEtest module from **pyats**. Testscripts are made up of three defined sections: Common Setup, Testcases, and Common Cleanup. During execution, the **aetest** infrastructure internally wraps the running testscript into a **TestScript** class instance. The important piece of the **TestScript** instance is that it stores script arguments as parameters in the instance to use throughout testing, and all the major sections (Common Setup, Testcases, Common Cleanup) point to the **TestScript** instance as their parent during testing (that is, Testcase.parent).

### Container Classes

There are three container classes in AEtest: **CommonSetup**, **Testcase**, and **CommonCleanup**. Conveniently, each of these containers was covered in detail at the beginning of this chapter, but let's now focus on the implementation details. All the container classes are inherited from the **TestContainer** class, which is a base class. Base classes are internal to pyATS and will not be covered further in this book, as there is little reason to access or manipulate base classes. The purpose of container classes is simply to house other sections. Tests are not defined directly in a container class. They must be included in a test section, which is written inside of a container class. Table 4-2 shows the attributes and properties of a container class.

**Table 4-2** *Container Class Attributes/Properties*

| **CommonSetup, CommonCleanup, Testcase (base cls: TestContainer)** | |
|---|---|
| **Attribute** | **Description** |
| source | File/line information where the class was defined |

| CommonSetup, CommonCleanup, Testcase (base cls: TestContainer) | |
|---|---|
| **Properties** | **Description** |
| uid | The common_setup/common_cleanup, or uid of Testcase |
| description | Class header (docstring) |
| parent | TestScript object |
| result | Rolled-up result of this section |
| parameters | Dictionary of parameters relative to this section |

To dive deeper into the technical details, note that container class instances are callable iterables. Let's take a second to break that down. A callable allows you to run, or "call," the code. In Python, functions and classes can be called; hence, they can be referred to as callables. An iterable is an object that can be iterated or looped over. Example 4-5 shows how a Common Setup container instance can be looped over and directly called.

**Example 4-5** *Container Class*

```
from pyats import aetest
# Define a container and two subsections
class MyCommonSetup(aetest.CommonSetup):
    @aetest.subsection
    def subsection_one(self):
        self.a = 1
        print("hello world")

    @aetest.subsection
    def subsection_two(self):
        assert self.a == 1

# Instantiate the class
common_setup = MyCommonSetup()

# Loop through to see what we get:
for i in common_setup:
    print(i)
```

## Function Classes

Function classes are housed within container classes and are what carry out the actual tests. Function classes include the **Subsection**, **SetupSection**, **TestSection**, and **CleanupSection** classes. These class names may look familiar to the different section decorators we discussed previously in "Testscript Structure." Each function class is

short-lived. They are instantiated during runtime and only live as long as the section runs. Table 4-3 shows the attributes and properties of function classes.

**Table 4-3** *Function Class Attributes/Properties*

| **Subsection, [Setup\|Test\|Cleanup]<br>Section (base cls: TestFunction)** | |
|---|---|
| **Attribute** | **Description** |
| function | Function/method that was decorated to be this section |
| source | File/line information where the method was defined |
| **Properties** | **Description** |
| uid | Name of the function/method |
| description | Function header (docstring) |
| parent | Container (**CommonSetup/Testcase/CommonCleanup**) |
| result | Rolled-up result of this test function |
| parameters | Dictionary of parameters relative to this function |

Any class method that has a section decorator is instantiated with their corresponding function class. For example, a class method with the decorator **@aetest.test** instantiates the **TestSection** class. This allows the AEtest infrastructure to manage each section's reporting context, enables result tracking, and other features specific to the test section methods. Example 4-6 shows the internals of each function class within a Testcase class instance.

**Example 4-6** *Function Class*

```
from pyats import aetest

class MyCommonSetup(aetest.CommonSetup):

    # subsection corresponds to Subsection class
    @aetest.subsection
    def subsection_one(self):
        pass

class MyTestcase(aetest.Testcase):

    # setup corresponds to SetupSection class
    @aetest.setup
    def setup(self):
        pass
```

```
    # test corresponds to TestSection class
    @aetest.test
    def test_one(self):
        pass

    # cleanup corresponds to CleanupSection class
    @aetest.cleanup
    def cleanup(self):
        pass

# When container instances are iterated, the returned objects are function
# class instances
tc = MyTestcase()
for obj in tc:
    print(type(obj))
    print(obj.function)

# Printed results:
# <class 'pyats.aetest.sections.SetupSection'>
# <bound method MyTestcase.setup of <class 'MyTestcase' uid='MyTestcase'>>
# <class 'pyats.aetest.sections.TestSection'>
# <bound method MyTestcase.test_one of <class 'MyTestcase' uid='MyTestcase'>>
# <class 'pyats.aetest.sections.CleanupSection'>
# <bound method MyTestcase.cleanup of <class 'MyTestcase' uid='MyTestcase'>>
```

## Runtime Behavior

The AEtest module provides access to objects and attributes that are only available during runtime via the runtime object. The runtime object is available only while the testscript is executing. Currently, **uids** and **groups** are the only two accessible variables; however, this could change in future releases.

The runtime object can be useful for querying and possibly manipulating the execution flow of your testscript. For example, you may want to ensure only certain testcases or sections run during testing. This can be done by querying the testcase UID and/or group. We will talk about groups later in the chapter, but in short, they allow you to arbitrarily label testcases. This allows you to better organize which testcases to run in a testscript. Example 4-7 shows how the testscript will only run testcases that belong to the “L3” group but are not in the “L2” group.

**Example 4-7** *AEtest Runtime*

```
from pyats import aetest

from pyats.datastructures.logic import And, Not

class CommonSetup(aetest.CommonSetup):
    # Allows testcases in "L3" and not in "L2" group to execute
    @aetest.subsection
    def validate_l3_testcases(self):
        aetest.runtime.groups = And("L3", Not("L2"))
        # Print runtime groups
        print(aetest.runtime.groups)
```

Although this example reads straightforward, I do want to touch on the logic expressions used to differentiate the group names. Notice at the beginning of the example that we imported the keywords **And** and **Not** from **pyats.datastructures.logic**. This is another hidden gem within pyATS. The logic module allows you to easily produce logic testing with English keywords. The logic module also allows us to use callables that accept arguments to perform additional logic before returning a value that is used for truth testing.

### Self

In Python, when a class is instantiated, it has the ability to access its attributes and methods through the **self** keyword. The use of **self** is a convention, not a rule, in Python; however, there aren't many instances where you will see any other keyword used to represent an instance of a Python class. In AEtest, container classes (Common Setup, Testcases, and Common Cleanup) are all Python classes. This allows you to get and set class-level attributes during testing. For example, let's say you wanted to use a value from one test section in another, such as the MAC address table. You might have multiple tests that need the results of the **show mac-address table** command. Instead of running the command and collecting the results in each individual test, you can run it once and save it as a class attribute using **self**. Example 4-8 shows a testcase that executes the **show mac-address table** command, collects the output, and uses that output in a future test.

**Example 4-8** *Self Testcase*

```
from pyats import aetest

class L2Testcase(aetest.Testcase):
    @aetest.setup
    def collect_l2_info(self, device):
```

```
        # Collect MAC address table entries
        self.mac_table = device.execute("show mac-address table")

    @aetest.test
    def confirm_mac_addresses(self):
        # Confirm the "important" MAC address is found in the MAC address
        important_mac = "0123.4567.0987"
        assert important_mac in self.mac_table
```

Obviously, this is not the best way to go about checking for a particular MAC address in the MAC address table, but you can clearly see how the MAC address table output is collected in the setup section and used in a separate test section within the testcase. Being able to get and set class attributes using **self** can be a powerful tool by allowing testcases to run more efficiently and remove redundancies.

## Parent

At the beginning of the chapter, we touched on the testscript structure and the concept of parent-child object relationships. Let's dive a bit further into those topics. Besides the **TestScript** class, all other classes in the AEtest object model have a parent class. Figure 4-2 shows a graphical representation of the parent-child relationships among the different AEtest class objects.

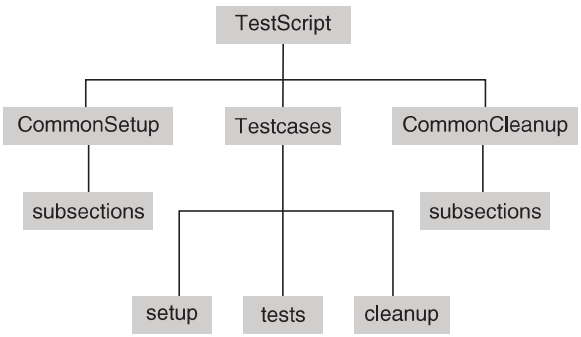


**Figure 4-2** *Parent-Child Object Relationships*

The **parent** attribute is accessible during testing by using **self.parent** within the container/section. It's not used often but may be useful if you are trying to access a parent's parameters (for example, a **Testcase** class accessing the parameters assigned to the **TestScript** class).

### Section Ordering

Testscripts have a logical, reproducible order in which container classes (Common Setup, Testcases, and Common Cleanup) and sections within the container classes (setup, test, cleanup) are discovered and executed. This allows for every testscript to run the same, regardless of the execution environment. Container classes execute in the following order:

- Common Setup always runs first.
- Testcases run in the order in which they appear in the script.
- Common Cleanup always runs last.

Within a container class, there can be multiple child methods: setup, subsection, test, and cleanup. The child methods execute in the following order:

- Setup always runs first (if defined).
- The subsection and test methods run as they appear in the script.

  If the parent class is inherited, the parent class's subsection and tests run first.
- Cleanup always runs last.

The execution order might seem apparent when looking at examples in this book or the library's documentation, but the guaranteed ordering provides uniformity to testscript execution and leaves nothing to chance.

## Test Results

Now that we know about the structure of a testscript, all the different containers and sections, the order they are executed, and even the runtime behavior, let's talk about test results. Test results may seem straightforward—the test either passes or fails—but that's not all. Pass, fail, and error may work for traditional software testing frameworks, but we are dealing with a network infrastructure. As network engineers, we all know the engineering slogan: "It depends." To accommodate for potential unrelated network failures, poor design, and so on, AEtest has added some additional result types, such as skipped and errored, and exception handling to better describe and gain a better understanding of the test results. Along with test results, understanding how the results roll up and affect test reporting is crucial.

### Result Objects

Before we dive deeper into understanding test results, let's list the different result objects with a short description of each:

- **Passed**: Test was successful.
- **Failed**: Test was not successful.

- **Aborted**: Something started but did not finish.
- **Blocked**: A test dependency was not met, and the test could not start.
- **Skipped**: A test was not executed and omitted.
- **Errored**: A mistake or unexpected exception occurred. The difference from Failed is that the test ran but did not meet expectations. Errored indicates something went wrong during testing and it could not be completed.
- **Passx**: Short for "pass with exception." Essentially remarks a Failed result with Passed based on an expected exception.

## Result Behavior

By default, all test results are "Passed." A testcase could be empty, and as long as no exceptions are thrown, it will pass. This type of behavior is standard among other Python testing frameworks. However, when exceptions are thrown, AEtest has the capability to catch exceptions and assign a result to the corresponding section. AEtest can catch and handle the following exceptions:

- **AssertionError**: A built-in Python exception raised when an assert statement fails. AEtest will catch this exception and assign a Failed result to the test section.
- **Exception**: The base Exception class in Python. All built-in Python exceptions are derived from this class. AEtest will catch any exception and assign an Errored result, indicating an unhandled error to the developer. This should allow the developer to quickly locate any unhandled errors during testing and fix them or catch them properly.

To avoid these exceptions being caught by AEtest, you should use **try**…**except** blocks to catch any exceptions and handle them appropriately. For example, if you expect that a device or set of devices will not parse a particular **show** command due to no output, you should try to catch the SchemaParserEmpty Exception and handle it by either skipping or blocking the test from running for those devices. Otherwise, AEtest will catch the exception for you and assign an Errored result to the test section.

To be more granular and assign explicit results to a test section, you simply call the result object and provide a reason for the result. Example 4-9 shows an example of three tests marked with Passed, Skipped, and Errored.

**Example 4-9** *Test Result Assignment*

```
from pyats import aetest

class ResultTestcase(aetest.Testcase):

    @aetest.test
```

```
    def subsection_that_passes(self):
        self.passed("This test passed!")

    @aetest.test
    def subsection_that_skips(self):
        self.skipped("This test skipped!")

    @aetest.test
    def subsection_that_is_errored(self):
        self.errored("This test errored!")
```

Along with a reason that is printed with the result, results can also include a few more optional arguments:

- **Reason:** Describes to the user why a test result occurred (shown in Example 4-9).
- **Goto:** List of sections to "go to" after this section. This is essentially a one-way ticket to another test section in the testscript.
- **From_exception:** Accepts an exception object and will add the traceback to the reason.
- **Data:** A dictionary of data relevant to the result. This data is passed to and stored with the Reporter object for further processing.

Once a result is determined for the current section, AEtest moves on to the next test section. Any code that is included after the result is determined in a test section will not be executed.

## Interaction Results

As a network engineer, you might still want or require some control of the testing that occurs. "What if I need to move patch cables during testing? I don't want to sit in a testing lab all day waiting around to pause testing, move one silly patch cable, and click a button to continue testing." Automation can't solve all our problems, but it can take them into consideration! AEtest offers the **WebInteraction** class, which pauses test execution and can notify a user (via email) that input is required via a web page. The web page is a form for the user to submit a test result for the test(s) that required intervention. You can even customize the email body and HTML web page using Jinja2 templates to really make it your own!

## Result Rollup

Result rollup involves combining the results of many child sections into one summary result. The simplest example is a testcase with two test sections. If one of the two test sections fails, the testcase fails. The rollup concept is easy to understand, but it can be complex when dealing with many testcases and multiple test sections as well as adding a

few steps in each test section. Table 4-4 shows a lookup table for a summary result when combining multiple results. This table can be used to check which result "wins" over the other. For example, let's say you have a testcase with three test section results: Passed, Skipped, and Passx, in that order. Starting from the top, you can compare the Passed and Skipped results and see in the table that the summary result for the testcase would be Passed. Next, compare the Passed and Passx results. This is interesting because now the summary result changes to Passx instead of Passed. This is due to the fact that the summary result is meant to inform the user of any negative results or exceptions caught during testing, as these issues would need fixed. If the summary result would have resulted in Passed, we wouldn't have seen the test section that passed with an exception (Passx result).

**Table 4-4** *Result Roll-up Table*

| Results | Failed | Passed | Aborted | Blocked | Skipped | Errored | Passx |
|---|---|---|---|---|---|---|---|
| **Failed** | Failed | Failed | Aborted | Failed | Failed | Errored | Failed |
| **Passed** | Failed | Passed | Aborted | Blocked | Passed | Errored | Passx |
| **Aborted** | Aborted | Aborted | Aborted | Aborted | Aborted | Aborted | Aborted |
| **Blocked** | Failed | Blocked | Aborted | Blocked | Blocked | Errored | Blocked |
| **Skipped** | Failed | Passed | Aborted | Blocked | Skipped | Errored | Passx |
| **Errored** | Errored | Errored | Aborted | Errored | Errored | Errored | Errored |
| **Passx** | Failed | Passx | Aborted | Blocked | Passx | Errored | Passx |

# Processors

Processors are functions that are executed before or after a given section. Processors are optional in AEtest but can be used to perform helpful checks before and after testing, such as collecting test environment information, taking snapshots, validating section results, or executing debug commands and collecting dump files. The possibilities are endless, as they are simply Python functions. In the following sections, we will cover the different processor types, including how to define and use them in a testscript.

## Processor Types

There are three types of processors:

- Pre-processors
- Post-processors
- Exception processors

Pre-processors are executed before a given section and may be used to take snapshots of the current environment or determine whether a test should run. Post-processors are executed after a section and may be used to validate the test results or collect debug information or dump files. Exception processors are kicked off when an exception

occurs. They may be used to collect debug information when an exception occurs or to suppress exceptions that are raised within a section and assign a proper result for the section. Because these processors are just Python functions, you can get creative with the logic and data collected for each processor type.

## Processor Definition and Arguments

All the processor types (pre/post/exception) can be applied to test containers (Common Setup, Testcases, and Common Cleanup) and test sections (subsections, setup, test, and cleanup). Each section may have one or more processors, in the form of a list, that execute in the order they appear. A processor can be applied to a test container or section with the **@aetest.processors** decorator. Example 4-10 shows pre-, post-, and exception processors applied to a testcase with two tests.

**Example 4-10** *Testcase Processors*

```
from pyats import aetest

# Print section uid
def print_uid(section):
    print("current section: ", section.uid)

# Print section result
def print_result(section):
    print("section result: ", section.result)

# Print the exception message and suppress the exception
def print_exception_message(section, exc_type, exc_value, exc_traceback):
    print("exception : ", exc_type, exc_value)
    return True

# Use the above functions as pre/post/exception processors to a Testcase
#   pre-processor  : print_uid
#   post-processor : print_result
#   exception-processor : print_exception_message
@aetest.processors(pre = [print_uid],
                   post = [print_result],
                   exception = [print_exception_message])
class Testcase(aetest.Testcase):

    @aetest.test
    def test(self):
        print("First test section...)
```

```
    @aetest.test
    def testException(self):
        raise Exception("Exception raised during testing... ")
```

Processors may have parameters propagated to them via a datafile or parent containers/sections. Processor arguments must have the same name as the parameters being passed in. Some default parameters included are section, processor, and steps. Exception processors have additional default parameters, which include the exception type (**exc_type**), exception value (**exc_value**), and exception traceback (**exc_traceback**). These default parameters can be very powerful when determining test section results or troubleshooting issues during testing. Example 4-10 shows the different attributes of the section parameter (**uid** and **result**).

## Context Processors

Context processors are more advanced processors that act similar to Python context managers in the sense that they handle the pre-, post-, and exception-handling processors within a single class instead of a single Python function. When you're creating a context processor, the pre-processor actions are defined in the **__enter__** method. Actions defined in the **__exit__** method handle the post- and exception-processor logic. A context processor class has the results API available, so calling **self.failed()** within the context processor class would be like **processor.failed()** and fail the processor.

## Global Processors

Global processors are processors that run automatically before and after each test container and section defined in a testscript. They do not require the **@aetest.processors** decorator to be applied to each container/section. To use global processors in your testscript, you must define a Python dictionary in your testscript called global_processors. In this dictionary, you must specify the following keys to represent the different processor types: pre, post, and exception. A list of processors can be specified as the value for each processor type key. Example 4-11 shows how global processors are defined in a testscript.

**Example 4-11** *Global Processors*

```
from pyats import aetest

# Print section uid
def print_uid(section):
    print("current section: ", section.uid)

# Print section result
```

```
def print_result(section):
    print("section result: ", section.result)

# Print the exception message and suppress the exception
def print_exception_message(section, exc_type, exc_value, exc_traceback):
    print("exception : ", exc_type, exc_value)
    return True

# Use the above functions to define global pre/post processors
#   global pre-processor  : print_uid
#   global post-processor : print_result
#   global exception-processor : print_exception_message
global_processors = {
    "pre": [print_uid,],
    "post": [print_result,],
    "exception": [print_exception_message,],
}

class Testcase(aetest.Testcase):

    @aetest.test
    def test(self):
        print('running testcase test section')

<rest of testscript omitted for brevity>
```

### Processor Results

Like test sections, processors have their own result and can be marked as passed, failed, skipped, and so on. The result will roll up to the parent object, the same as any other child result; however, the processor result can directly affect the parent test section's result. Because processors have access to the section object via the default parameters provided to processors, a processor can alter a section's result by calling the section results API. For example, to fail a test section, simply call **section.failed()** in a processor function. Once the processor is run, it will fail the parent test section. For pre-processors, this will block the execution of the test section and set the result as failed. For post-processors, this will override the existing results of the test section and mark it as failed.

## Data-Driven Testing

AEtest testscripts and testcases are intended to be driven dynamically by data. Dynamic data that alters and affects the behavior of testscripts and testcases is called a parameter. Test parameters are meant to be dynamic in nature and can be provided to testscripts in

the form of input arguments or generated during runtime. In this section, we will go over test parameter relationships, their properties, calling parameters, parametrization, and reserved parameters. Beyond test parameters, we will also take a look at datafile inputs and looping sections. If it isn't already apparent, the focus in these upcoming sections is how to *dynamically* affect the execution and runtime behavior of your testscript. By utilizing these concepts, your testscript organization and testing logic will greatly mature.

## Test Parameters

Parameters are variables used to access input data (arguments) in Python functions and methods. In the context of AEtest, a parameter will take the value of a testbed argument, which is passed to a testscript, to instruct the testscript as to which testbed to connect to for testing. If the testbed argument is not available, a testscript would have to be hardcoded with the testbed name, which eliminates the dynamic nature of testing and makes it impossible to scale. This simple example, which outlines one way to pass data to AEtest through script arguments, allows you to understand the importance of test parameters. In the following sections, we are going to take a look at the relationships, properties, and ways to call different test parameters during testing.

### Parameter Relationships

Test parameters are relative to the test section. The test parameters for a given test section are a combination of local parameters in the section and any parent parameters. Going back to the AEtest object model and how the container and sections relate to one another, we see that parameters are inherited the same way. Figure 4-3 shows a visual of the parameter relationship model and how parameters are inherited from their parent container/sections.

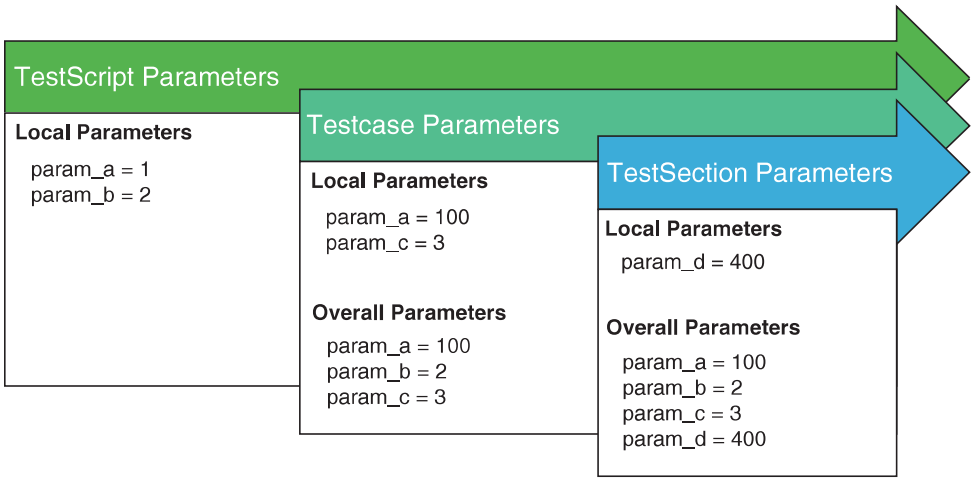


**Figure 4-3** *Parameter Relationship Model*

You can see how the list of overall parameters continues to grow as you move from the TestScript parameters to the Testcase parameters, to the TestSection parameters, so that parameters defined at the TestScript level are made available at the TestSection level. Another key point is that test parameters can be overwritten. For example, **param_a = 1** at the TestScript level was changed to **param_a = 100** at the Testcase level and is presented as such to the TestSection. This can be key if you're planning to implement a parameter that you expect will change during testing. Initialize the parameter at the highest possible level to make it available throughout testing and then alter it as needed.

### Parameter Properties

Each top-level object (TestScripts and Testcases) in AEtest has a special **parameters** property that represents the different test parameters for that particular object. The **parameters** property is a Python dictionary that stores the test parameters as key-value pairs. They can store default values for the test parameters, which can be changed by accessing the test parameter during runtime. Function classes such as Subsection, SetupSection, TestSection, and CleanupSection have a **parameters** property as well, but these class instances only exist briefly during runtime, so we cannot statically set a dictionary of default parameters for these sections. It's recommended that you consolidate test parameters for these sections and include them in the parent TestContainer **parameters** property.

### Parameter Types

There isn't an official list of parameter "types," but it's important to understand the different ways in which a parameter can be included in testing. Test parameters can be added to a testscript as script arguments, function arguments, or callables. Let's take a look at each one.

Script arguments are any arguments passed directly to a testscript before startup. This includes arguments passed by a jobfile, command-line arguments, or even updating the **parameters** property of a TestScript object within the testscript code with a dictionary of parameters. Example 4-12 shows a simple example of how testscript parameters can be updated within code.

**Example 4-12** *TestScript Test Parameters*

```
# The following parameters were already defined
parameters = {
    "arg_a": 1,
    "arg_b": 2,
}

# The following inputs were passed as arguments to the testscript
script_arguments = {
    "arg_a": 100,
    "arg_c": 3,
}
```

```
# The TestScript parameters would be built as follows
testscript.parameters = parameters
testscript.parameters.update(script_arguments)

# Result - you'll notice that arg_a was updated by the script arguments
testscript.parameters
# {"arg_a": 100,
#  "arg_b": 2,
#  "arg_c": 3}
```

Parameters can also be passed as function arguments. Input parameters passed to the testscript as script arguments can be explicitly passed to a function as an argument. This is due to the parent-child object model in AEtest. During runtime, all function arguments are filled with the corresponding parameter value, with the argument and parameter names matching. It is preferred to explicitly pass each parameter as function arguments, as it makes the code easier to understand and allows you to call each function with different arguments during testing or debugging. Example 4-13 shows how parameters can be passed down to child containers/sections and changed in a testscript.

**Example 4-13** *Parameters—Function Arguments*

```
from pyats import aetest

# Script-level parameters
parameters = {
    "param_A": 1,
    "param_B": dict(),
}

class Testcase(aetest.Testcase):

    # "param_B" is passed to the setup section as a function argument
    @aetest.setup
    def setup(self, param_B):

        # param_B is a dictionary and can be changed (mutable)
        # Any changes are persist throughout the testscript
        param_B['new_key'] = "a key added during setup section"

    # "param_A" and "param_B" are passed to the test section
    @aetest.test
    def test_one(self, param_A, param_B):
        print(param_A)
        # 1
        print(param_B)
        # {'new_key': 'a key added during setup section'}
```

The last way a test parameter can be provided to a testscript is via a callable. As mentioned earlier in the chapter, functions and classes are considered callables. In the context of AEtest, many times we are dealing with functions when talking about callables. A callable parameter must evaluate to True to be valid, which means the callable can't return a boolean of False, None, empty strings, or a numeric value of 0. Callables are passed as function arguments and are "called" during runtime. The return value of the callable is used as the actual parameter. The one limitation to callables is that they cannot have arguments of their own, as AEtest will not pass any arguments to the callable.

### Parameter Parametrization

Parametrized parameters are identical to callables, but they enable you to create "smarter" functions by allowing you to introduce more dynamic parameter values. Parametrized functions are declared using the **@aetest.parameters.parametrize** decorator. Unlike normal callables, arguments can be passed to the parametrized function with the **@aetest.parameters.parametrize** decorator. Along with allowing arguments, a special argument named **section** can be passed to a parametrized function that allows you to access the current section object. This includes access to the current section's properties, such as **uid** and **result**. By having access to the current section object, you can dynamically change the return value based on the parent section's result or in combination with the test parameters available to the current section.

### Reserved Parameters

AEtest has reserved parameters that are generated during runtime. They are not available when you're accessing the parameters property but can be used to access internal objects to AEtest. They are only accessible if their name is provided as a keyword argument to a test section. Reserved parameters take precedence over normal parameters if there is a normal parameter with the same name. The purpose of reserved parameters is to provide a mechanism for engineers to access and dive deeper into the internals of AEtest within a testscript. It's highly recommended that you only access the reserved parameters if required and to never modify a reserved parameter, as it could lead to unexpected behavior.

## Datafile Input

Up to this point, testscripts are defined as static files with multiple test containers/sections that can be altered by test parameters. However, what if we want more flexibility and the ability to dynamically update the testscript without having to manually change the code? *Datafiles* are YAML input files that can be passed to AEtest and allow you to dynamically update testscript parameter values. They are completely optional but allow you to easily change testscript values without having to modify the original testscript code. With datafiles being written in YAML, they are easily readable, even by non-programmers, which empowers users of pyATS to easily modify testscript values and feel confident doing so.

Datafile inputs directly update the testscript's module parameters before runtime. Only container classes can be updated via datafiles: **CommonSetup**, **Testcases**, and **CommonCleanup**. However, due to the parent-child relationship of these container classes to the individual test sections, the datafile values can be used in the individual test sections. The common and testcase container classes defined in the testscript must have matching names in the datafile. For example, a testcase defined in the testscript as **class BGPTestcase** that requires dynamic values from the datafile must include a **bgp-testcase:** section in the datafile. If a section is looped, which will be discussed in the next section, only the base class attributes are changed. If values must change on *n* iterations, you must pass those values as loop parameters. The last key point to datafiles is that only one can be provided to a testscript; however, you may *extend* another datafile. Much like Jinja2 template inheritance, a base datafile can be extended to create more modular datafiles for testing. The extensibility reduces the amount of redundant datafiles that need to be created and can help promote others to build their own datafiles by simply extending a core datafile.

There is a defined datafile schema in the pyATS documentation, but for brevity, Example 4-14 shows two datafiles, base.yaml and datafile.yaml, that can be passed to a testscript (see Example 4-15). It's assumed that a pyATS testbed file (testbed.yaml) exists in the same directory.

**Example 4-14** *The base.yaml and datafile.yaml Datafiles*

```
---
# base.yaml
# testscript parameters
parameters:
    cloudflare_dns: 1.1.1.1

...
---
# datafile.yaml
extends: base.yaml

# testscript parameters
parameters:
  google_dns: 8.8.8.8
  # Adds to existing cloudflare DNS from base.yaml

testcases:
  BGPTestcase:
    # testcase uid
    uid: routing_test_1
```

```
    # list of groups that testcase belongs to
    groups: [routing]

    # testcase parameters
    parameters:
      local_asn: 65000
      remote_asn: 65001

    # testcase class variable
    expected_routes: 5

  ExternalConnectivity:
    # testcase uid
    uid: ext_dns_test

    # list of groups that testcase belongs to
    groups: [routing]

    # testcase parameters
    parameters:
      alt_google_dns: 8.8.4.4
...
```

**Example 4-15** *Testscript with Datafile Input*

```
! Make into code block
import logging
from genie.utils import Dq
from pyats import aetest
from unicon.core.errors import ConnectionError

logger = logging.getLogger(__name__)
logger.setLevel("INFO")

class CommonSetup(aetest.CommonSetup):
    @aetest.subsection
    def connect_to_devices(self, testbed):
        """Connect to all testbed devices"""
        try:
            testbed.connect()
```

```
        except ConnectionError:
            self.failed(f"Could not connect to all devices in {testbed.name}")

        # Print log message confirming all devices are in a 'connected' state
        logger.info(f"Connected to all devices in {testbed.name}")

class BGPTestcase(aetest.Testcase):
"""Test BGP operational state"""

    @aetest.test
    def check_bgp_routes(self, testbed):
        """Check number of BGP neighbors equals expected number of routes in
        datafile."""

        # Print all class variables (as a Python dictionary)
        print(f"All class variables: {vars(BGPTestcase)}")
        # Example Output
# {'__module__': '__main__', 'check_bgp_routes': <function
# BGPTestcase.check_bgp_routes at 0x10bf30430>,
# '__parameters__': {'local_asn': 65000, 'remote_asn': 65001},
# '__doc__': None, 'source': <pyats.aetest.base.Source object at 0x10beeebb0>,
# '__uid__': 'routing_test_1', 'groups': ['routing'], 'expected_routes': 1,
# 'uid': <property object at 0x108195b30>}

# Print available test parameters (provided by datafile) - includes TestScript
# and Testcase-level parameters
        print(f"Available testcase parameters: {self.parameters}")
# Example Output
# ParameterMap({'local_asn': 65000, 'remote_asn': 65001},
# {'cloudflare_dns': '1.1.1.1', 'google_dns': '8.8.8.8',
# 'testbed': <Testbed object 'Cat8k Lab' at 0x108cfa490>})
        # Parse 'show up route bgp' command output using Genie parsers
        r1_bgp_routes = testbed.devices["cat8k-rt1"].parse("show ip route bgp")
        r2_bgp_routes = testbed.devices["cat8k-rt2"].parse("show ip route bgp")

# Capture the number of BGP routes in the routing table using the Genie Dq library
        self.r1_route_count = (
        (len(Dq(r1_bgp_routes).contains("routes").get_values("route")))
        )
```

```
        self.r2_route_count = (
        (len(Dq(r2_bgp_routes).contains("routes").get_values("route")))
        )

        # Confirm number of BGP routes equals the expected number of routes
        # provided as a class variable in the datafile
        if self.r1_route_count == self.expected_routes:
            self.passed(
            "There were the correct number of expected BGP routes on router 1."
            )
        else:
            self.failed(f"Router 1 does not have the expected number of BGP \
            routes ({self.expected_routes}). Instead, there are \
            {self.r1_route_count} BGP routes.")
        if self.r2_route_count == self.expected_routes:
            self.passed("There were the correct number of expected BGP routes \
            on router 2.")
        else:
            self.failed(f"Router 2 does not have the expected number of BGP \
            routes ({self.expected_routes}). Instead, there are \
            {self.r2_route_count} BGP routes.")

class ExternalConnectivity(aetest.Testcase):
"""Test external connectivity by pinging external DNS servers (Google and
Cloudflare) using pyATS device Ping API.
There are no pass/fail conditions in this testcase, as the goal is to
illustrate the use of datafile input parameters. All tests will pass.
"""

    @aetest.test
    def ping_cloudflare_dns(self, testbed):
        """Ping Cloudflare DNS servers"""

# Print all TestScript-level parameters - you'll notice the BGPTestcase
# parameters are not included, as they are Testcase-level parameters
# You'll also notice the addition of the 'alt_google_dns' parameter, as that
# is a Testcase-level parameter
        print(self.parameters)
# Example Output
# ParameterMap({'alt_google_dns': '8.8.4.4'}, {'cloudflare_dns': '1.1.1.1',
```

```
# 'google_dns': '8.8.8.8', 'testbed': <Testbed object 'Cat8k Lab' at
# 0x10a485310>})

# Use 'cloudflare_dns' TestScript-level parameter to ping Cloudflare DNS servers
# (found in base.yaml)
        testbed.devices["cat8k-rt1"].api.ping(self.parameters["cloudflare_dns"])

    @aetest.test
    def ping_google_dns(self, testbed):
        """Ping Google DNS servers"""

# Use 'google_dns' TestScript-level parameter and 'alt_google_dns' Testcase-level
# parameter to ping Google DNS servers (both found in datafile.yaml)
        testbed.devices["cat8k-rt1"].api.ping(self.parameters["google_dns"])
        testbed.devices["cat8k-rt1"].api.ping(self.parameters["alt_google_dns"])

class CommonCleanup(aetest.CommonCleanup):
    @aetest.subsection
    def disconnect_from_devices(self, testbed):
        """Disconnect from all devices"""
        testbed.disconnect()
        logger.info(f"Disconnected from all devices in {testbed.name}")

if __name__ == "__main__":

    from pyats.topology.loader import load

    # Load testbed object from testbed file
    tb = load("../testbed.yaml")
    # Run with standalone execution
    aetest.main(datafile="datafile.yaml", testbed=tb)
```

The datafile in Example 4-14 shows a few different values that can be changed, including the testcase UID, groups, test parameters, and class-level variables. The class variables are accessible via the **self** keyword within the respective testcase. For example, in the **BGPTestcase** class definitions, you can access **the expected_routes** class variable via **self.expected_routes**. It's important to verify the number of expected routes being received from BGP because if there are not enough or too many BGP routes being received, it can lead to abnormal routing in your network. If you're a service provider, it can be more detrimental to your business, as this can lead to outages across multiple customers.

Datafiles are extremely powerful and allow pyATS testscripts to be more modular and dynamic in nature without altering any testscript code. The only requirement is to be able to read and update a YAML-based file. Being able to hand over the keys to the testing framework to the engineer testing their changes increases adaptability and confidence of test-driven network automation.

## Looping Sections

AEtest provides the ability to loop over test sections with different parameters for each loop iteration. This is another feature of AEtest, along with datafiles, that allows the testing infrastructure to be dynamic. You can reuse test section code without having to edit the code. Only certain test sections can be looped: subsections within **CommonSetup/CommonCleanup** and **Testcases** and test sections within **Testcases**.

### Defining Loops

Sections that are decorated with the **@aetest.loop** decorator are marked for looping. The looping parameters are provided as decorator arguments. During runtime, if a test section is marked for looping, an instance of the test section is created for each loop iteration. As a convenience, you may also use the following decorators on the subsection and test sections, respectively: **@aetest.subsection.loop** and **@aetest.test.loop**. These decorators essentially combine the two decorators you normally would have to mark each section with—**@aetest.{subsection | test}** and **@aetest.loop**.

### Loop Parameters

Looping over a test section is only useful if different test parameters are provided. These parameters are passed in as arguments to the **@aetest.loop** decorator. The test parameters are propagated to the test section as local parameters. There are two methods to providing loop parameters:

- Providing a list of parameters and another list of parameter values (uses args and argvs)
- Providing each parameter as a keyword argument and a list of the parameter values as the value to the argument

There isn't a suggested method, as both methods produce the same results, but it's up to the specific use case as to whether one method should be used over the other. Example 4-16 shows both methods being used on two different test sections.

**Example 4-16** *Looping Parameters*

```
from pyats import aetest

class Testcase(aetest.Testcase):

    # Method 1 - args and argvs - the positions of each value match to its arg name
```

```
    @aetest.test.loop(args=('a', 'b', 'c'),
                      argvs=((1, 2, 3),
                             (4, 5, 6)))
    def test_one(self, a, b, c):
        print("a=%s, b=%s, c=%s" % (a, b, c))

    # Method 2 - keyword args - each argument in the lists are provided indepen-
  dently
    @aetest.test.loop(a=[1,4],
                      b=[2,5],
                      c=[3,6])
    def test_two(self, a, b, c):
        print("a=%s, b=%s, c=%s" % (a, b, c))

# OUTPUT GENERATED IF TESTCASE IS EXECUTED:
# testcase output:
#   a=1, b=2, c=3
#   a=4, b=5, c=6
#   a=1, b=2, c=3
#   a=4, b=5, c=6
#
#  SECTIONS/TESTCASES                                                    RESULT
# ------------------------------------------------------------------------------
#  .
#  '-- Testcase                                                          PASSED
#      |-- test_one[a=1,b=2,c=3]                                        PASSED
#      |-- test_one[a=4,b=5,c=6]                                        PASSED
#      |-- test_two[a=1,b=2,c=3]                                        PASSED
#      '-- test_two[a=4,b=5,c=6]                                        PASSED
```

Along with test parameters, you may also pass in alternative UIDs to identify each looped section. When you're using loop parameters, the number of iterations depends on a couple different factors. If alternative UIDs are provided, the number of iterations is equal to the number of UIDs provided. If there are more loop parameter values than UIDs, the extra values are discarded. If there aren't any alternative UIDs provided, the number of iterations is equal to the number of loop parameter values.

Loop parameters can also be a callable, iterable, or generator. If the argument value is a callable, the return value from the callable is used as the loop argument value. If the argument value is an iterable or generator, only one element is used at a time for each loop iteration until the iterable or generator is exhausted. Example 4-17 shows a callable (function) and generator being used as loop parameter values.

**Example 4-17** *Loop Parameters: Callable and Generator*

```
from pyats import aetest

# callable function
def my_function():
    value = [1, 2, 3]
    print("returning %s" % value)
    return value

# generator
def my_generator():
    for i in [4, 5, 6]:
        print("generating %s" % i)
        yield i

class Testcase(aetest.Testcase):

    # creating test section with parameter "a" as a function
    # note that the function object is passed, not its values
    @aetest.test.loop(a=my_function)
    def test_one(self, a):
        print("a = %s" % a)

    # creating a test section with parameter "b" as a generator
    # note that the generator is a result of calling my_generator(), not
    # the function itself.
    @aetest.test.loop(b=my_generator())
    def test_two(self, b):
        print("b = %s" % b)

# OUTPUT GENERATED IF TESTCASE IS EXECUTED:
#   returning [1, 2, 3]
#   a = 1
#   a = 2
#   a = 3
#   generating 4
#   b = 4
#   generating 5
#   b = 5
#   generating 6
#   b = 6
```

You might notice that the callable is run and the return value is captured before the looped sections are created, while the generator is only queried before the next section needs created. This is important because the generator is only queried before each test iteration; it allows a generator to dynamically generate loop iterations based on the current test environment instead of providing one return value before test iteration.

### Dynamic Looping

Up to this point, we've discussed how to statically mark different test sections for looping and provide loop parameters in a testscript, but what if we wanted to loop a section based on a runtime variable? For example, what if we only want to mark a test for looping based on a certain condition or calculated value that can only be determined during runtime. *Dynamic looping* offers the ability to mark a specific test section for looping using the **loop.mark()** function. Example 4-18 shows how you can mark a test section for looping in the setup section of a testcase.

**Example 4-18** *Dynamic Looping*

```
from pyats import aetest

class Testcase(aetest.Testcase):

    @aetest.setup
    def setup(self):
        # mark the next test for looping
        # provide it with two unique test uids.
        # (self.simple_test is the next test method)
        aetest.loop.mark(self.simple_test, uids=["test_one", "test_two"])

    # note: the simple_test section is not directly marked for looping
    # instead, during runtime, its testcase's setup section marks it for
    # looping dynamically.

    @aetest.test
    def simple_test(self, section):
        # print the current section uid
        # by using the internal parameter "section"
        print("current section: %s" % section.uid)

# OUTPUT GENERATED IF TESTCASE IS EXECUTED:
#   current section: test_one
#   current section: test_two
#
#  SECTIONS/TESTCASES                                                    RESULT
# ------------------------------------------------------------------------------
#  .
#  '-- Testcase                                                          PASSED
#      |-- setup                                                         PASSED
#      |-- test_one                                                      PASSED
#      '-- test_two                                                      PASSED
```

The **loop.mark()** function is identical to the **@aetest.loop** decorator, with the exception that the first argument must be the target test section/class. For example, to mark a BGP

testcase that uses different ASNs for each loop iteration, you would use the following syntax in a preceding class or section:

```
loop.mark(BGPTestcase, asn=[65000, 65001, 65002])
```

# Running Testscripts

Now time for what you've been waiting for… running a testscript! Testscripts can be run using one of two execution methods: Standalone or Easypy execution. Before reviewing each execution method, let's dive into the AEtest Standard Arguments and how arguments are parsed and propagated from the command line.

## Testing Arguments

Test arguments provide a way to supplement and influence the execution of your testscript. AEtest has a set of standard arguments, called Standard Arguments, along with the ability to accept arguments from the command line when running a testscript. In the following sections, you'll see the Standard Arguments provided by AEtest and how we can use the Python argparse standard library module to parse command-line arguments and propagate them to individual test sections.

### Standard Arguments

AEtest has a number of standard arguments, referred to as Standard Arguments, used to influence/change testscript execution. Standard Arguments can be provided as command-line arguments or keyword arguments to **aetest.main()** in Standalone execution or **easypy.run()** in Easypy execution. Table 4-5 shows the available AEtest Standard Arguments.

**Table 4-5** *AEtest Standard Arguments*

| Keyword | Command Line | Description |
|---|---|---|
| n/a | **-help** | Display help information |
| **uids** | **-uids** | Specify the list of section UIDs to run (logic expression) |
| **groups** | **-groups** | Specify the list of testcase groups to run (logic expression) |
| **datafile** | **-datafile** | Input datafile/value for this script |
| **random** | **-random** | Flag to enable testcase randomization |
| **random_seed** | **-random_seed** | Testcase randomization seed |
| **max_failures** | **-max_failures** | Max acceptable number of failures |
| **Pdb** | **-pdb** | Start interactive debugger on failure |

| Keyword | Command Line | Description |
|---|---|---|
| step_debug | -step_debug | Step debug input file |
| pause_on | -pause_on | Pause on phrase input string/file |
| loglevel | -loglevel | AEtest logging level |
| submitter | -submitter | Submitter of this script (defaults to current user) |

You may remember the **datafile** argument from the previous "Datafile Input" section. Datafiles are provided to a testscript as a standard argument. If you recall, datafiles provide dynamic test parameters and updates to test sections, which influences the behavior and execution of testscripts. That is the overall goal of Standard Arguments—to influence the execution of testscripts.

### Argument Propagation

AEtest parses and propagates all command-line arguments using the Python standard library argparse module. The argparse module makes it easy to write command-line interfaces in Python. Using the argparse module, AEtest parses the argument values stored in sys.argv, which is a list of command-line arguments passed to a Python script. You may be wondering, what if you pass in only Standard Arguments? How are they parsed? Here's the process of parsing command-line arguments:

- All standard arguments are parsed and removed from the sys.argv list.
- All unknown arguments, which are arguments that aren't part of the Standard Arguments, are parsed by sys.argv and the argparse module.

Argument propagation allows users to pass in additional arguments to the testscript via the command line, but a custom argument parser must be created to use those arguments in test sections. The custom argument parser can simply use **argparse.ArgumentParser** in a Python script to parse known arguments passed to the script. We will look at examples in each execution method section.

## Execution Environments

AEtest can execute testscripts using one of two methods: Standalone execution or Easypy execution. Standalone execution is meant for testing scripts and rapid development, while Easypy execution is meant for production script where proper reporting and log archiving are required. In the following sections, we will go over the different execution methods and how they are run.

### Standalone Execution

Standalone execution is meant to be used during script development and allows the user to have full control of the execution environment, including logging and reporting. All logging is redirected to standard output (stdout) and standard error (stderr). Reporting is

handled by the Standalone Reporter, which tracks results and prints a summary at the end of testing to standard output (stdout). Many of the examples in this chapter have shown results from Standalone execution. No TaskLog, result report, or archives are generated during Standalone execution.

Testscripts are run as standalone when one of the two following methods is used to execute the script:

- Directly calling **aetest.main()** within a user script
- Indirectly calling **aetest.main()** by invoking Python's **__main__** mechanism

Example 4-19 shows a testscript executed as standalone, and Example 4-20 shows the accompanying results printed to standard output (stdout).

**Example 4-19** *Standalone Execution*

```
import logging
from pyats import aetest

class CommonSetup(aetest.CommonSetup):
    # Subsection 1
    @aetest.subsection
    def subsection_one(self):
        pass

    # Subsection 2
    @aetest.subsection
    def subsection_two(self):
        pass

class Testcase(aetest.Testcase):
    # Test 1
    @aetest.test
    def test_one(self):
        pass

    # Test 2
    @aetest.test
    def test_two(self):
        pass
```

```
    # Test 3
    @aetest.test
    def test_three(self):
        pass

# add the following as the absolute last block in your testscript
if __name__ == '__main__':

    # control the environment
    # e.g., change some log levels for debugging
    logging.getLogger(__name__).setLevel(logging.DEBUG)
    logging.getLogger('pyats.aetest').setLevel(logging.DEBUG)

    # aetest.main() api starts the testscript execution.
    # defaults to aetest.main(testable = '__main__')
    aetest.main()
```

**Example 4-20** *Standalone Execution Results*

```
    +------------------------------------------------------------------------------+
    |                            Starting common setup                             .
    |
    +------------------------------------------------------------------------------+
    +------------------------------------------------------------------------------+
    |                       Starting subsection subsection_one
    |
    +------------------------------------------------------------------------------+
    The result of subsection subsection_one is => PASSED
    +------------------------------------------------------------------------------+
    |                       Starting subsection subsection_two
    |
    +------------------------------------------------------------------------------+
    The result of subsection subsection_two is => PASSED
    The result of common setup is => PASSED
    +------------------------------------------------------------------------------+
    |                            Starting testcase Testcase
    |
    +------------------------------------------------------------------------------+
    +------------------------------------------------------------------------------+
    |                            Starting section test_one
    |
```

```
+------------------------------------------------------------------------------+
The result of section test_one is => PASSED
+------------------------------------------------------------------------------+
|                          Starting section test_two
|
+------------------------------------------------------------------------------+
The result of section test_two is => PASSED
+------------------------------------------------------------------------------+
|                         Starting section test_three
|
+------------------------------------------------------------------------------+
The result of section test_three is => PASSED
The result of testcase Testcase is => PASSED
+------------------------------------------------------------------------------+
|                               Detailed Results
|
+------------------------------------------------------------------------------+
 SECTIONS/TESTCASES                                                    RESULT
------------------------------------------------------------------------------
 .
 |-- common_setup                                                     PASSED
 |   |-- subsection_one                                               PASSED
 |   '-- subsection_two                                               PASSED
 '-- Testcase                                                         PASSED
     |-- test_one                                                     PASSED
     |-- test_two                                                     PASSED
     '-- test_three                                                   PASSED
+------------------------------------------------------------------------------+
|                                  Summary
|
+------------------------------------------------------------------------------+
 Number of ABORTED                                                        0
 Number of BLOCKED                                                        0
 Number of ERRORED                                                        0
 Number of FAILED                                                         0
 Number of PASSED                                                         2
 Number of PASSX                                                          0
 Number of SKIPPED                                                        0
------------------------------------------------------------------------------
```

The **aetest.main()** function provides the entry point and is what starts the script execution. Standard Arguments can be passed to **aetest.main()** as keyword arguments. Any other unknown keyword arguments are propagated as script arguments. If any unknown keyword arguments are passed as command-line arguments, you'll need to create a custom argument parser. This might sound like a lot, but you may use the argparse module to create an **ArgumentParser** object, add arguments, parse the arguments, and add them to **aetest.main()** as keyword arguments. Example 4-21 shows how to pass two command-line arguments, **testbed** and **vlan**, as keyword arguments to **aetest.main()**, which in turn makes them testscript parameters.

**Example 4-21** *Standalone Execution—Input Arguments*

```
from pyats import aetest

class Testcase(aetest.Testcase):

    # defining a test that prints out the current parameters
    # in order to demonstrate argument passing to parameters
    @aetest.test
    def test(self):
        print('Parameters = ', self.parameters)

# do the parsing within the __main__ block,
# and pass the parsed arguments to aetest.main()
if __name__ == '__main__':

    # local imports under __main__ section
    # this is done here because we don't want to pollute the namespace
    # when the script isn't run under standalone
    import sys
    import argparse
    from pyats import topology

    # creating our own parser to parse script arguments
    parser = argparse.ArgumentParser(description = "standalone parser")
    parser.add_argument('--testbed', dest = 'testbed',
                        type = topology.loader.load)
    parser.add_argument('--vlan', dest = 'vlan', type = int)

    # do the parsing
    # always use parse_known_args, as aetest needs to parse any
    # remainder arguments that this parser does not understand
    args, sys.argv[1:] = parser.parse_known_args(sys.argv[1:])
```

```
    # and pass all arguments to aetest.main() as kwargs
    aetest.main(testbed = args.testbed, vlan = args.vlan)


# Let's run this script with the following command
#   example_script.py --testbed /path/to/my/testbed.yaml --vlan 50

# output of the script:
#
#   +------------------------------------------------------------------------------+
#   |                           Starting testcase Testcase                         |
#.  |
#   +------------------------------------------------------------------------------+
#   +------------------------------------------------------------------------------+
#   |                             Starting section test                            |
#.  |
#   +------------------------------------------------------------------------------+
#   Parameters = {'testbed': <Testbed object at 0xf717578c>, 'vlan': 50})
#   The result of section test is => PASSED
#   The result of testcase Testcase is => PASSED
```

Standalone execution provides the user ultimate control and is great when going through the trial-and-error process of writing code. However, what do we do if we want to run our testscripts in production and require proper logging and reporting? Easypy execution provides the answer.

### Easypy Execution

Easypy execution is used when testscripts are executed with the Easypy runtime environment. With this execution method, the Easypy runtime environment controls the environment and provides the following features:

- Multiple testscripts can be run together in a job file.
- Logging configuration is done by Easypy.
- TaskLog, result reporting, and archives are generated.
- Reporter is used for reporting and result tracking as well as generating a YAML result file, results details file, and a summary XML file.

Example 4-22 shows an Easypy job file running two testscripts. Each testscript that's run within a job file is called a *task*.

**Example 4-22** *Easypy Execution*

```
from pyats.easypy import run

# job file needs to have a main() definition
# which is the primary entry point for starting job files
def main():

    # run testscript 1
    run(testscript='/path/to/your/script1.py')

    # run testscript 2
    run(testscript='/path/to/your/script2.py')
```

To run the Easypy job, you must run **pyats run job** *jobfile-name*.**py --testbed-file** */path/to/***testbed.yaml** from the terminal. The **–testbed-file** loads the testbed as a testbed object and is propagated to the testscript as a script argument named **testbed.**

If no **–testbed-file** is passed to **pyats run job**, the **testbed** argument is set to **None.**

In addition to the **–testbed-file** option, all AEtest Standard Arguments are accepted as keyword arguments and propagated to the testscript as script arguments. Any unknown keyword arguments provided to **easypy.run()** are also propagated to the testscript as script arguments. Example 4-23 shows how unknown keyword arguments are propagated to the testscript as script arguments.

**Example 4-23** *Easypy Execution—Script Arguments*

```
from pyats.easpy import run

def main():
    run(
        testscript="standalone_exec_input_args.py",
        pyats_is_awesome=True,
        aetest_is_legendary=True
    )

# Run the easypy job
# pyats run job easypy_script_args.py --testbed ../testbed.yaml

#   +------------------------------------------------------------------------------+
#   |                           Starting testcase Testcase
#   |
#   +------------------------------------------------------------------------------+
```

```
#   +------------------------------------------------------------------------+
#   |                            Starting section test
#.  |
#   +------------------------------------------------------------------------+
#   Parameters = {'testbed': <Testbed object at 0xf742f74c>,
#                 'pyats_is_awesome': True,
#                 'aetest_is_legendary': True}
#   The result of section test is => PASSED
#   The result of testcase Testcase is => PASSED
```

Along with having the ability to run multiple testscripts in a single job, another major benefit of Easypy is the standardization of logging, reporting, and archiving. After a job file is run, a zipped archive folder is created in the user home directory under .pyats/archive/YY-MM (~/.pyats/archive/YY-MM). You can specify archives to not be created by specifying the **–no-archive** option. Table 4-6 shows a list of the files generated by Easypy job files.

**Table 4-6** *Easypy Job Files*

| Filename | Purpose |
|---|---|
| <*job-name*>.py | Copy of the jobfile that ran. |
| <*job-name*>.report | Copy of the email notification sent to the submitter. This report looks very much like the Standalone Reporter results. |
| TaskLog.<*task-id*> | One TaskLog is generated per jobfile task and is where all messages generated in the task are stored. |
| JobLog.<*job-name*> | Overall pyats.easypy module log. |
| testbed.static.yaml | Contents of the testbed file provided by the user. |
| testbed.clean.yaml | Contents of the clean file provided by the user. |
| env.txt | A dump of environment variables and CLI arguments of the Easypy run. |
| reporter.log | Reporter server log file. Contains a trace of XML-RPC calls. |
| results.json | JSON result summary file generated by Reporter. |
| xunit.xml | xUnit-style results reports and information required by Jenkins. Only generated if **–xunit** argument is provided. |
| ResultsSummary.xml | XML result summary file generated by Reporter. |
| ResultsDetails.xml | XML result details file generated by Reporter. |
| CleanResultsDetails.yaml | YAML clean result details file generated by Kleenex. |

| Filename | Purpose |
|---|---|
| Kleenex.<*device-name*>.log | Job-scope clean details for this device. |
| Kleenex_<*task-id*>.<*device-name*>.log | Task-scope clean details for this device. |

As you can see from Table 4-6, many different files are generated and archived from an Easypy job. The archived files can be used for additional regression and sanity testing.

### Testable

A testable in AEtest is any object that can be loaded into a **TestScript** class instance by the aetest.loader module and executed without any errors. The following are acceptable as testables:

- Any path to a Python file ending with .py
- Any module name that is part of the current PYTHONPATH
- Any non-built-in module objects (instances of **types.ModuleType**)

Testables are not the same as testscripts. Testscripts run tests and generate results. Testables can be meaningless modules to AEtest, such as the urllib module. It is a valid testable but produces zero test results.

## Testscript Flow Control

AEtest provides many mechanisms to control the execution flow of testscripts. Different mechanisms include skipping testcases, jumping ahead in the testscript, grouping testcases, and only executing testcases by UID.

### Skip Conditions

AEtest comes with built-in preprocessors that can be used to skip test sections, sometimes based on a condition. The following decorators and functions can be used to skip a test section:

- **@aetest.skip(reason = 'message'):** Unconditionally skip the decorated section. **reason** should describe why that section is being skipped.
- **aetest.skip.affix(testcase, reason):** Same as the skip decorator but can be used on the fly to skip other testcases depending on one testcase result.
- **@aetest.skipIf(condition, reason = 'message'):** Skip the decorated test section if **condition** is True.

- **aetest.skipIf.affix(testcase, condition, reason)**: This can be used to assign the **skipIf** decorator to the testcases; **condition** can be a callable or a boolean.
- **@aetest.skipUnless(condition, reason = 'message')**: Skip the decorated test section unless **condition** is True.
- **aetest.skipUnless.affix(testcase, condition, reason)**: Can be used on the fly to assign decorators to the testcases.

Example 4-24 shows some of these decorators and functions used to skip testcases.

**Example 4-24** *AEtest Skip Conditions*

```
from pyats import aetest

# Custom library used for testing
class mylibrary:
    __version__ = 0.1

# skip testcase intentionally
@aetest.skip('because we had to')
class Testcase(aetest.Testcase):
    pass

class TestcaseTwo(aetest.Testcase):

    # skip test section using if library version < some number
    @aetest.skipIf(mylibrary.__version__ < 1,
                   'not supported in this library version')
    @aetest.test
    def test_one(self):
        pass

    # skip unless library version > some number
    @aetest.skipUnless(mylibrary.__version__ > 3,
                       'not supported in this library version')
    @aetest.test
    def test_two(self):
        pass

    @aetest.test
    def test_three(self):
        aetest.skip.affix(section = TestcaseTwo.test_four,
                          reason = "message")
        aetest.skipIf.affix(section = TestcaseTwo.test_five,
```

```
                            condition = True,
                            reason = "message")
        aetest.skipUnless.affix(section = TestcaseThree,
                                condition = False,
                                reason = "message")

    @aetest.test
    def test_four(self):
        # will be skipped because of test_three
        pass

    @aetest.test
    def test_five(self):
        # will be skipped because of test_three
        pass

    @aetest.test
    def test_six(self):
        # will be skipped because of test_three
        pass

class TestcaseThree(aetest.Testcase):
    # will be skipped because of TestcaseTwo.test_three
    pass
```

## Running Specific Testcases

You might want to run only specific testcases. To do that, you can specify a testcase UID (or UIDs) as a Standard Argument when running the script or by setting a runtime variable (**runtime.uids**) dynamically during execution. The **uids** argument accepts a callable (function) that returns a truthy value. The list of test section UIDs present in the testscript are passed as arguments to the callable. If the callable returns True, the respective test section is run. Logic testing can also be used to evaluate test section UIDs. The running section UIDs are also accessible via the **runtime.uids** variable during runtime. Runtime variables are only accessible during runtime, so the UIDs will have to be dynamically set in the testscript versus passed in as a Standard Argument. Example 4-25 shows how a callable can be used to determine whether a UID should be run.

**Example 4-25** *Running Specific Testcases*

```
from pyats.easypy import run

# function determining whether we should run testcase_A
# currently executing uids is always a list of:
# [ <container uid>, <section uid>]
# e.g., ['common_setup', 'subsection_one']
# thus varargs (using *) is required for the function input.
def run_only_testcase_one(*uids):
    # check that we are running TestcaseOne
    return "TestcaseOne" in uids

# run only TestcaseOne and its contents (using callable)
# executing uids has TestcaseOne:
def main():
    run("example_script.py", uids=run_only_testcase_one)
```

## Testcase Grouping

Testcase grouping allows you to tag testcases that are similar in nature and may be run together by adding them to a *group*. By default, testcases do not belong to any groups. You may add testcases to groups by adding them in the testscript itself by assigning a list of groups to a group variable within the testcase. Testcases can also be grouped by specifying groups in datafile input that is provided as a Standard Argument. Example 4-26 shows an example of assigning a group named "traffic" to Testcase One and "sanity" to a Testcase Two.

**Example 4-26** *Testcase Grouping*

```
from pyats import aetest

class TestcaseOne(aetest.Testcase):
    """Testcase One"""

    groups = ["traffic"]

    <TestcaseOne tests...>

class TestcaseTwo(aetest.Testcase):
    """Testcase Two"""

    groups = ["sanity"]

    <TestcaseTwo tests...>
```

Once the testcases are grouped, you can specify which testcase groups run using Standard Arguments (**--group**) or using the **runtime.groups** variable dynamically in your testscript. Just like how you specify which testcases to run using their UID, you pass a callable to the **groups** argument to determine whether the group(s) should run. The callable accepts a list of each of the testcase's group values and will return True if the testcase group(s) should run. Logic testing can also be used to evaluate whether a group value should run. Groups can also be evaluated at runtime using the **runtime.groups** variable within the testscript. The **runtime.groups** variable is dynamically set by performing logic testing. Example 4-27 shows how to filter certain testcase groups using a callable in Standard Arguments and also dynamically using the runtime variable.

**Example 4-27** *Testcase Group Filtering*

```
from pyats.easypy import run
# import the logic objects
from pyats.datastructures.logic import And, Not

# create a function that tests for testcase groups
# this api tests that a testcase belongs to sanity but not traffic.
# note that varargs (using *) is required, as the list of groups to each
# testcase is unknown.
def non_traffic_sanities(*groups):
    # Runs testcases in "sanity" group and not in "traffic" group
    return "sanity" in groups and "traffic" not in groups

# Runs the testscript as two tasks using different logic testing
def main(runtime):
    ### Using function testing to evaluate testcase groups ###
    # Only runs Testcase Two
    run(testscript="example_script.py", runtime=runtime, groups=non_traffic_sani-
  ties)

    ### Using logic testing to evaluate testcase groups ###
    # Only runs Testcase One
    run("example_script.py", groups=And("sanity", Not("traffic")))
```

## Must-Pass Testcases

If there are testcases that must pass during testing, AEtest allows you to set a class attribute called **must_pass** to True. If a must-pass testcase fails during testing, the testscript will immediately jump to the Common Cleanup section, using the **goto** statement, and block any remaining testcases. The **goto** statement was touched on earlier in the chapter,

but to recap, it allows you to jump to another section within a testscript. The **goto** target must be further in the testscript—you can't go back to a previously executed section. The available targets include the testcase's cleanup section (**cleanup**), the next testcase (**next_tc**), the Common Cleanup section (**common_cleanup**), or exiting the testscript completely (**exit**). Example 4-28 shows how to use the **goto** statement directly, and Example 4-29 shows how to set a testcase as "must pass" and what happens if that testcase fails. You'll notice the following testcase is blocked and the testscript jumps to the Common Cleanup section using the **goto** statement under the hood.

**Example 4-28** *The goto Statement*

```
from pyats import aetest

class CommonSetup(aetest.CommonSetup):
    @aetest.subsection
    def subsection(self):
        # goto with a message
        self.errored('setup error, abandoning script', goto = ['exit'])

# ------------------------------------------------------------------------
class TestcaseOne(aetest.Testcase):
    @aetest.setup
    def setup(self):
        # setup failed, go to cleanup of testcase
        self.failed('test failed', goto = ['cleanup'])

# ------------------------------------------------------------------------
class TestcaseTwo(aetest.Testcase):
    # test failed, move onto next testcase
    @aetest.test
    def test(self):
        self.failed(goto = ['next_tc'])

# ------------------------------------------------------------------------
class TestcaseThree(aetest.Testcase):
    @aetest.setup
    def setup(self):
        # setup failed, move onto cleanup of this testcase, then
        # jump to common_cleanup directly.
        self.failed(goto=['cleanup','common_cleanup'])
```

**Example 4-29** *The Must-Pass Testcase*

```
from pyats import aetest

class TestcaseOne(aetest.Testcase):

    must_pass = True

    @aetest.test
    def test(self):
        self.failed('boom!')

class TestcaseTwo(aetest.Testcase):
    pass

class CommonCleanup(aetest.CommonCleanup):

    @aetest.subsection
    def subsection(self):
        pass

# output result
#
#  SECTIONS/TESTCASES                                                    RESULT
# ------------------------------------------------------------------------------
#  .
#  |-- TestcaseOne                                                       FAILED
#  |   '-- test                                                          FAILED
#  |-- TestcaseTwo                                                     BLOCKED
#  '-- common_cleanup                                                    PASSED
#      '-- subsection                                                    PASSED
```

## Testcase Randomization

By default, AEtest runs the Common Setup section first, then each testcase in the order they are defined, and finally wraps up with the Common Cleanup section. Testcase execution can be randomized by setting the random standard argument to True (**random=True**). Common Setup and Common Cleanup are not randomized and will always be executed first and last, respectively. Example 4-30 shows a basic example of randomizing testcases.

**Example 4-30** *Testcase Randomization*

```
from pyats import aetest

# define a couple testcases
class TestcaseOne(aetest.Testcase):
    pass

class TestcaseTwo(aetest.Testcase):
    pass

class TestcaseThree(aetest.Testcase):
    pass

if __name__ == "__main__":
    aetest.main(random = True)

# output result
#
#  SECTIONS/TESTCASES                                                    RESULT
# ------------------------------------------------------------------------------
#  .
#  |-- TestcaseTwo                                                       PASSED
#  |-- TestcaseOne                                                       PASSED
#  '-- TestcaseThree                                                     PASSED
```

## Maximum Failures

Let's say you are testing many network features and have a long-running testscript. By default, AEtest will run each testcase sequentially and record the respective result. However, if testcases begin to fail, wouldn't you want the ability to stop testing and figure out what's going on without waiting for the testscript to finish executing? AEtest provides a method to set a maximum threshold for testcase failures during a testscript run. The **max_failures** standard argument can provide the number of testcase failures before aborting the rest of the testcases and jumping to the Common Cleanup section of the testscript. The **goto** statement is used once again to jump to the Common Cleanup section. Example 4-31 shows how when one testcase fails, the testscript blocks execution of the other testcases and jumps to the Common Cleanup section before exiting.

**Example 4-31** *Maximum Failures*

```
from pyats import aetest

class TestcaseOne(aetest.Testcase):

    @aetest.test
    def test(self):
        self.failed()

class TestcaseTwo(aetest.Testcase):

    @aetest.test
    def test(self):
        self.failed()

class TestcaseThree(aetest.Testcase):
    pass

class CommonCleanup(aetest.CommonCleanup):
    pass

# set max failure to 1 and run the testscript
if __name__ == "__main__":
    aetest.main(max_failures = 1)

# output result
#
# Max failure reached: aborting script execution
#
#  SECTIONS/TESTCASES                                                    RESULT
# ------------------------------------------------------------------------------
#  .
#  |-- TestcaseOne                                                       FAILED
#  |-- TestcaseTwo                                                     BLOCKED
#  |-- TestcaseThree                                                   BLOCKED
#  '-- common_cleanup                                                   PASSED
```

## Custom Testcase Discovery

Customizing testcase discovery is an advanced topic but should be covered at a high level. Testcases are discovered using the **ScriptDiscovery** class in the discover module of the AEtest package. You can customize the testcase discovery process at the following

levels: script discovery, testcase discovery, and common discovery. The **ScriptDiscovery** class finds the testcases within a testscript, the **TestDiscovery** class finds the test sections (setup, test, cleanup) within a testcase, and the **CommonDiscovery** class finds subsections within the common sections (**CommonSetup** and **CommonCleanup**). To override the discovery process at each level, you can create a new class that inherits the respective default discovery class. The new discovery class must have specific methods to enable the custom discovery logic. The **runtime.discoverer** properties can be configured in the testscript to use the new discovery classes instead of the default classes. Along with custom discovery, you can also customize the ordering of sections. If you would like to customize the discovery or ordering of sections, it's recommended that you reference the pyATS documentation for more details.

## Reporting

AEtest provides reporting of testscript results, including which tests ran during testing and their associated results. The format and level of details in the report depend on the execution mode used for testing (Standalone or Easypy execution). In the following sections, you'll see the different report options available and dive into the reporting details of each execution mode.

### Standalone Reporter

The Standalone Reporter is used when testscripts are run directly from the command-line using Standalone execution (via **aetest.main()**). Testcase, section, and step results are printed to standard output (stdout) in a tree-like format. All examples in this chapter, and most examples in this book, use the Standalone Reporter to showcase testscript results. Example 4-32 shows testscript results presented by the Standalone Reporter.

**Example 4-32** *Standalone Reporter*

```
+------------------------------------------------------------------------------+
|                              Detailed Results                                |
+------------------------------------------------------------------------------+
 SECTIONS/TESTCASES                                                    RESULT
--------------------------------------------------------------------------------
.
|-- common_setup                                                      PASSED
|   |-- sample_subsection_1                                           PASSED
|   '-- sample_subsection_2                                           PASSED
|-- tc_one                                                            PASSED
|   |-- prepare_testcase                                              PASSED
|   |-- simple_test_1                                                 PASSED
|   |-- simple_test_2                                                 PASSED
|   '-- clean_testcase                                                PASSED
```

```
|-- TestcaseWithSteps                                                ERRORED
|   |-- setup                                                         PASSED
|   |   |-- Step 1: this is a description of the step                 PASSED
|   |   '-- Step 2: another step                                      PASSED
|   |-- step_continue_on_failure_and_assertions                       FAILED
|   |   |-- Step 1: assertion errors -> Failed                        FAILED
|   |   '-- Step 2: allowed to continue executing                     FAILED
|   |-- steps_errors_exits_immediately                               ERRORED
|   |   '-- Step 1: exceptions causes all steps to skip over         ERRORED
|   '-- steps_with_child_steps                                        PASSED
|       |-- Step 1: test step one                                     PASSED
|       |-- Step 1.1: substep one                                     PASSED
|       |-- Step 1.1.1: subsubstep one                                PASSED
|       |-- Step 1.1.1.1: subsubsubstep one                           PASSED
|       |-- Step 1.1.1.1.1: running out of indentation                PASSED
|       |-- Step 1.1.1.1.1.1: definitely gone too far...              PASSED
|       |-- Step 1.2: substep two                                     PASSED
|       |-- Step 2: test step two                                     PASSED
|       |-- Step 2.1: function step one                               PASSED
|       |-- Step 2.2: function step two                               PASSED
|       '-- Step 2.3: function step three                            PASSED
'-- common_cleanup                                                    PASSED
    '-- clean_everything                                              PASSED
```

## AEtest Reporter

The AEtest Reporter, known as the Reporter, is used when testscripts are executed via Easpy execution mode. The Reporter creates a package of test result artifacts. It contains information such as the section hierarchy, section results, and even the amount of time each section took during testing. The main files in the package are results.json and results.yaml, which contains hierarchical information about the job. The top-level is TestSuite, which contains high-level information about the entire job. Under TestSuite is the Task level. The Task level represents each testscript that is executed in the job. If you remember, in an Easypy job, multiple testscripts can be executed. Each testscript executed is called a task. As you can imagine, below each Task are the different container classes—Common Setup, Testcase, and Common Cleanup. Following the AEtest section hierarchy, each container class has child sections, including SetupSection, TestSection, CleanupSection, and Subsection. Optionally, these child sections can contain steps represented as Step. Each level to the report has information relevant to that section. Example 4-33 shows the different fields for each level represented in the report.

**Example 4-33** *AEtest Report Structure*

| Section | Field | Description |
|---|---|---|
| TestSuite | type | Identifier that this section is the root TestSuite |
| | id | Unique ID for this job execution |
| | name | Name from jobfile |
| | starttime | Timestamp when job execution began |
| | stoptime | Timestamp when job execution ended |
| | runtime | Duration of execution |
| | cli | Command that started Easypy |
| | jobfile | Location of jobfile |
| | jobfile_hash | SHA256 hash of the jobfile contents |
| | pyatspath | Python environment executing pyATS |
| | pyatsversion | Version of pyATS installed |
| | host | Name of host machine |
| | submitter | User that started execution |
| | archivefile | Path to generated archive file |
| | summary | Combined summary of all Tasks |
| | details | Details about any exceptions or errors |
| | extra | Map of extra info about the TestSuite |
| | tasks | List of child Tasks |
| Task | type | Identifier that this section is a Task |
| | id | Unique ID for this Task |
| | name | Name of TestScript |
| | starttime | Timestamp when execution began |
| | stoptime | Timestamp when execution ended |
| | runtime | Duration of execution |
| | description | Description of TestScript |
| | logfile | Path to logfile for this Task |
| | testscript | Path to testscript |
| | testscript_hash | SHA256 hash of the testscript contents |
| | datafile | Path to the data file |
| | datafile_hash | SHA256 hash of the data file contents |
| | parameters | Any parameters passed to this Task |
| | summary | Summary of results |
| | details | Details about any exceptions or errors |
| | extra | Map of extra info about the Task |
| | sections | List of child Sections |

| Section | type | Specific type of section being represented |
|---|---|---|
| | id | Unique ID for this Section |
| | name | Name of this Section |
| | starttime | Timestamp when this Section began |
| | stoptime | Timestamp when this Section ended |
| | runtime | Duration of execution |
| | description | Description of this Section |
| | xref | XReference to the code defining this Section |
| | source_hash | SHA256 hash of the source code for this Section |
| | data_hash | SHA256 hash of the data file input |
| | logs | Path to logfile showing execution of this Section, as well as the beginning byte and size in bytes |
| | parameters | Any parameters passed to this Section |
| | processors | Lists of processors that ran for this section, both before and after |
| | result | The test result of this Section |
| | details | Details about any exceptions or errors |
| | extra | Map of extra info about this Section |
| | sections | Any child sections of this section (Testcases have TestSections, which can have Steps, etc.) |

All levels below the TestSuite level are considered *sections*, as the information gathered from each level is about the same. The **type** identifies the section type. If there are any unique differences between sections, they will be saved under the **extras** key. Example 4-34 shows an abbreviated results.yaml file that shows the different levels to the report—TestSuite, Task, and the Common Setup section of the first task. The section types are highlighted to help identify the different levels.

**Example 4-34** *results.yaml*

```
version: '2'
report:
  type: TestSuite
  id: example_job.2019Sep19_19:56:06.569499
  name: example_job
  starttime: 2019-09-19 19:56:07.603283
```

```
stoptime: 2019-09-19 19:56:19.951458
runtime: 12.35
cli: pyats run job job/example_job.py --testbed-file etc/example_testbed.yaml
  --no-mail
jobfile: /Users/user/examples/comprehensive/job/example_job.py
jobfile_hash: 2a452a8683f4f5e5c146d62c78a9a5253198e19c3fb6c8c1771bdf0eea622086
pyatspath: /Users/user/env
pyatsversion: '19.11'
host: HOSTNAME
submitter: user
archivefile: /Users/user/env/users/user/archive/
19-09/example_job.2019Sep19_19:56:06.569499.zip
summary:
  passed: 13
  passx: 0
  failed: 1
  errored: 12
  aborted: 0
  blocked: 4
  skipped: 0
  total: 30
  success_rate: 43.33
extra:
  testbed: example_testbed
tasks:
  - type: Task
    id: Task-1
    name: base_example
    starttime: 2019-09-19 19:56:08.432390
    stoptime: 2019-09-19 19:56:08.617640
    runtime: 0.19
    description: |+
      base_example.py

      This is a comprehensive example base script that walks users through AEtest
      infrastructure features, what they are for, how they are used, how it
impacts
      their testing, etc.

    logfile: TaskLog.Task-1
    testscript: /Users/user/examples/comprehensive/base_example.py
    testscript_hash:
```

```
2938f2d2efbf9be144a9fe68667dd1c12753b84017a56e7d04caefe46edc0602
parameters:
  labels: {}
  links: []
  parameter_A: jobfile value A
  routers: []
  testbed: <pyats.topology.testbed.Testbed object at 0x106da92b0>
  tgns: []
summary:
  passed: 3
  passx: 0
  failed: 0
  errored: 3
  aborted: 0
  blocked: 0
  skipped: 0
  total: 6
  success_rate: 50.0
sections:
  - type: CommonSetup
    id: common_setup
    name: common_setup
    starttime: 2019-09-19 19:56:08.434411
    stoptime: 2019-09-19 19:56:08.458939
    runtime: 0.02
    description: |+
      Common Setup Section
          This is the docstring for your common setup section.
          Users should document the number of common setup subsections
          so that by reading this block of comments, it gives a generic
          feeling as to how CommonSetup is built and run.

    xref:
      file: /Users/user/examples/comprehensive/base_example.py
      line: 191
      source_hash:
      c366a269e45838deb9bed54d28fef648b921c4f19a1753fc1e46e4c9ba3f9264
    logs:
      begin: 0
      file: TaskLog.Task-1
      size: 4317
    parameters:
      labels: {}
```

```
    links: []
    parameter_A: jobfile value A
    parameter_B: value B
    routers: []
    testbed: <pyats.topology.testbed.Testbed object at 0x105ea92b0>
    tgns: []
  result:
    value: passed
```

Along with the results.yaml and results.json files, the Reporter also generates XML files named ResultsDetails.xml and ResultsSummary.xml for the aggregated results. The AEtest Reporter also provides the ability to subscribe to live result updates. The Reporter uses a Unix socket client/server model to collect information about each section during the job run, allowing the Reporter Client to subscribe to the Reporter Server for live updates on runtime details of each section. The subscribe functionality only works as an async function, and you should be familiar with the Python asyncio library (https://docs.python.org/3/library/asyncio.html) before proceeding with testing this feature. The asyncio library is part of the Python standard library and is used to write concurrent code using the async/await syntax and is well-suited for I/O-bound tasks. The client subscribes to the server and runs a callback each time event data is received. Table 4-7 shows the different values that can be extracted from event data.

**Table 4-7** *Reporter Event Data*

| Key | Description |
|---|---|
| **event** | A string to specify what kind of event occurred, such as **start_task** or **stop_section** |
| **type** | The type of section that triggered this event |
| **seq_num** | A unique number specific to the section triggering this event |
| **parent_seq_num** | The **seq_num** of the parent section if there is one |
| **id** | The ID of the related section |
| **name** | The name of the related section |
| **starttime** | A timestamp for when the section started |
| **stoptime** | A timestamp for when the section ended |
| **runtime** | How many seconds the section ran for |
| **result** | The result of the section |
| **logfile** | The name of the log file |

| Key | Description |
|---|---|
| logs | A mapping of log file name and offset of the relevant section of logs |
| xref | The location of this section in the script |

The last interesting piece of information the Reporter collects and adds to the report package is git information. Git information, including the repo, file, branch, and commit hash, are added to the report. This can be helpful for regression testing. Let's say the testsuite, or part of the testsuite, broke and you want to quickly figure out when it last worked. By recording the git information captured by the Reporter, including the commit hash, you can quickly identify the last commit when the test worked.

The reporting features in the AEtest test infrastructure provide options to quickly review test results with the Standalone Reporter or to a complete reporting package provided by the AEtest Reporter. The data points, metrics, and other rich data that can be extracted from the AEtest Reporter reporting package provide endless options for further data analysis and visualization of the Easypy job results. It's really up to you on how you want to utilize the captured results!

## Debugging

As with all code, you will find yourself debugging your AEtest testscripts. Python has a debugger module as part of the standard library called Python Debugger (pdb). The pdb debugger (https://docs.python.org/3/library/pdb.html) is used to set breakpoints, step through source code, line by line, and provide other debugging functions in your Python code. The one caveat to using pdb is when multiprocessing is involved. When multiprocessing is used, child processes are forked and break the functionality of pdb. Since AEtest uses multiprocessing, most notably with Easypy execution, AEtest built pdb debugging functionality into the framework.

When running AEtest testscripts, you can pass **pdb=True** as a Standard Argument and whenever an error, failure, or exception occurs, the testing engine pauses and starts an interactive post_mortem debugging section. The post_mortem functionality is built natively into pdb. Also, pdb can be passed as a command-line flag (**--pdb**) when you're running a job via the pyATS **run job** command.

Another pdb debugging feature in AEtest is "pause on phrase," which is the ability to pause test execution based on any log messages generated by the script. The log messages include Python logs, CLI output from devices, and any other logs captured by the root logger (logging.root). The following actions are supported when a script is paused on phrase:

- **Email**: Creates a pause file and emails the user. The script may continue to run once the pause file is deleted or when the timeout limit has been reached.
- **Pdb**: Pauses and opens a pdb debugger.
- **Code**: Pauses and opens a Python interactive shell.

To enable this feature, you must pass the **pause_on** Standard Argument to the script run with a value that provides a path to a YAML pause file that follows a specific schema to define the actions to take when the script is paused. Example 4-35 shows a YAML pause file.

**Example 4-35** *YAML Pause File*

```
timeout: 600        # pause a maximum of 10 minutes

patterns:
    - pattern: '.*pass.*'           # pause on all log messages including
                                    # .*pass.* in them globally

    - pattern: '.*state: down.*'    # pause whenever  'state: down' is found
      section: '^common_setup\..*$' # enable for all common_setup sections

    - pattern: '.*should pause.*'      # pause whenever 'should pause' is found
      section: '^TestcaseTwo\.setup$'  # pause on TestcaseTwo setup section
```

The default action is to email the user. When a log message matches a pattern defined in the YAML pause file, the user is notified via email with the log phrase captured and instructions on how to remove the pause and continue testing. To change the action to one of the other two supported actions (pdb or code), you simply need to specify an "action:" key with one of those values in the YAML pause file.

## Summary

This chapter covered the different components that make up the AEtest test infrastructure. The AEtest test infrastructure is the core to pyATS and provides the foundation for all testing. We reviewed the structure of a testscript, which went through the different sections, including common setup, testcases, and common cleanup. The AEtest object model reviewed the Python classes that are the base classes of the different sections from the testscript structure. The base classes include the TestScript classes, container classes, function classes, TestItem classes, and TestContainer classes. After understanding the testscript structure and the base classes of the AEtest object model, we dove into the behavior of test results. Section test results can be determined automatically using assertions or manually using the different result APIs available (passed, failed, errored, skipped, blocked, aborted, and passx). AEtest allows functions or methods to run before or after testscripts using pre-processors, post-processors, or exception processors. Pre- and post-processors can be helpful for checking the environment before a test executes, validating section results, and even taking and comparing snapshots before and after a test section. Exception processors can take post-execution snapshots of the test environment if an exception is raised in a test section or execute debug commands and collect dump files when an exception occurs.

Once we reviewed the intricacies of the AEtest testscript structure, object model, and how the results are determined, we dove into the extensibility of the test infrastructure, including datafiles and test parameters. Datafiles are YAML files that can provide dynamic test parameters to a testscript, making them more robust and reusable. Datafiles can be a gamechanger and should be used when possible to avoid static test parameters. After seeing how we can make testscripts more extensible and dynamic, we reviewed how to run testscripts using Standalone execution or through Easypy. Standalone execution is recommended for development purposes with all logging outputs being sent to standard output (stdout). The Easypy execution environment is recommended for "official" test execution, running testscripts as tasks in a jobfile. The jobfile produces logs and archives and is best suited for sanity and regression testing where reporting and archiving are required.

AEtest provides many ways to control the flow of testscript execution, including running specific testcases (using a UID or a group of testcases), declaring testcases as "must pass," randomizing testcase execution, declaring a maximum number of failures per testscript execution, and even customizing testcase discovery. We then wrapped up the chapter by reviewing the reporting mechanisms and how to debug testscripts. AEtest reporting mechanisms include the Standalone Reporter, which is used with Standalone execution, and the AEtest Reporter, which is used with the Easypy execution environment and produces a results.json file that includes all the details of the test execution.

It's recommended that you continue referencing the information from this chapter as you go through the rest of the book, as many topics and features are built on the topics discussed in this chapter.