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

Asynchronous Library (Parallel Call)

Asynchronous (async) execution defines the ability to run programs and functions in a non-blocking manner. In pyATS, it's recommended that you execute in parallel using multiprocessing. The proper use of multiprocessing can greatly improve the performance of a program and is only bounded by the physical number of CPUs and I/O limits. Multiprocessing is recommended with pyATS and test-driven automation for a variety of reasons:

- Separate memory space, meaning no race conditions (except with external systems)
- Very simple, straightforward code
- No global interpreter lock, taking full advantage of multiple CPU/cores
- Easily interruptible/killable child processes

pyATS also provides an API from the async module known as Parallel Call, or pcall, to further abstract and simplify parallel calls. pcall supports calling procedures and functions in parallel using multiprocessing fork, without you having to write boilerplate code to handle the overhead of process creation, waiting, and terminations. pcall supports calling all procedures, functions, and methods in parallel, if the return of the called target is a pickleable object. In Python, "pickleable" refers to the capability of an object to be serialized into a byte stream (and subsequently deserialized back into an object). Serialization with the pickle module allows objects to be saved to a file, transmitted over a network, or, as in the case of pyATS's pcall, passed between processes.

Consider pcall as a shortcut library to multiprocessing, intended to satisfy most users' need for parallel processing. However, for more custom and advanced use cases, you should stick with direct usage of multiprocessing.

The pcall API allows users to call any functions/methods (a.k.a. targets) in parallel. It comes with the following built-in features:

- Builds arguments for each child process/instance
- Creates, handles, and gracefully terminates all processes
- Returns target results in their called order
- Reraises child process errors in the parent process

Data Structures

New data structures have been introduced and maintained as part of the pyATS infrastructure. These new data structures are used as part of the pyATS source code and may prove to be useful in our users' day-to-day coding:

- **Attribute dictionaries:**
  - **AttrDict:** This is the exact same as the Python native dictionary, except that in most cases you can use the dictionary key as if it was an object attribute instead.

    - **NestedAttrDict**: This is a special subclass of **AttrDict** that recognizes when its key values are other dictionaries and auto-converts them into further **NestedAttrDict** instances.
- **Weak list references**: A standard list object stores every internal object as a direct reference. That is, if the list exists, its internally stored objects exist.
- **Dictionary represented using lists**: Accessing nested dictionaries often calls for recursive functions in order to properly parse and/or walk through them. This isn't always easy to code around. **ListDict** provides an alternative view on nested dictionaries, breaking down the value nested within keys to a simple concept of path and value. This flattens the nesting into a linear list, greatly simplifying the coding around nested dictionaries.
- **Orderable dictionary**:
    - Python's built-in **collections.OrderedDict** only remembers the order in which keys were inserted into it and does not allow users to reorder the keys and/or insert new keys into arbitrary positions in the current key order.
    - **OrderableDict** (Orderable Dictionary) is almost exactly the same as Python's **collections.OrderedDict**, with the added ability to order and reorder the keys inserted into it.
- **Logic testing**: Boolean algebra is sometimes confusing when used in the context of the English language. The goal of this module is to standardize how to represent and evaluate logical expressions within the scope of pyATS as well as to offer standard APIs, classes, and behaviors for users to leverage.
- **Configuration container**:
    - The **Configuration** container is a special type of **NestedAttrDict** intended to store Python module and feature configurations.
    - Avoid confusing Python configurations with router configurations. Python configurations tend to be key-value pairs that drive a particular piece of infrastructure, telling it how its behavior should be.

TCL Integration

This module effectively enables you to make Tcl calls in the current Python process and is 100% embedded. There's no child Tcl process. The actual Tcl interpreter is embedded within the current Python process, and the process ID (PID) of both Python and Tcl is the same.

Part of the goal of pyATS is to enable the testing community to leverage existing Tcl-based scripts and libraries. In order to make the integration easier, the Tcl module was created to extend the native Python-Tcl interaction:

- **Interpreter class**: Extends the native Tcl interpreter by providing access to ATS-tree packages and libraries.

- **Two-way typecasting:** APIs and Python classes enable typecasting Tcl variables to the appropriate Python objects and back. This includes, but is not limited to, int, list, string, array, and keyed lists.
- **Call history:** An historical record of Tcl API calls is maintained for debugging purposes.
- **Callbacks:** Callbacks from Tcl to Python code enable closer coupling.
- **Dictionary access:** You can access Tcl variables as if accessing a Python dictionary.
- **Magic Q calls:** You can call Tcl APIs as if calling a Python object method, with support for Python ***args** and ****kwargs** mapping to Tcl positional and dashed arguments.

Logging

A log is a log, regardless of what kind of prefixes each log message contains and in what format it ended up as, as long as it is human-readable and provides useful information to the user.

The Python logging module's native ability to handle and process log messages is more than sufficient for any logging needs and has always been suggested as the de facto logging module to use.

Therefore, for all intents and purposes, users of the pyATS infrastructure should always use just the native Python logging module as-is in their scripts and testcases. Example 6-7 demonstrates some simple logging functions.

Example 6-7 *pyATS Logging Functions*

```
#   import the logging module at the top of your script
#   setup the logger

import logging

# always use your module name as the logger name.
# this enables logger hierarchy
logger = logging.getLogger(__name__)

# use logger:
logger.info('an info message')
logger.error('an error message')
logger.debug('a debug message')
```

Result Objects

In most test infrastructures, such as pytest and unittest, test results are only available as pass, fail, or error. This works quite well in unit and simplistic testing. The downside of having only three result types, however, is the inability to describe testcase result relationships, or distinguish a test's genuine failure, versus a failure of the testscript caused by poor design/coding (for example, the testcase encountered a coding exception).

To accommodate complex test environments, pyATS supports more complicated result types such as "test blocked," "test skipped," "test code errored," and so on, and it uses objects and object relationships to describe them. These objects simplify the whole result tracking and aggregation infrastructure and grant the ability to easily roll up results together. Here are the available result objects:

- **Passed**: Indicates that a test was successful, passed, the result accepted, and so on.
- **Failed**: Indicates that a test was unsuccessful, fell short, the result was unacceptable, and so on.
- **Aborted**: Indicates something was started but was not finished, or was incomplete and/or stopped at an early or premature stage. For example, a script was killed via a **Ctrl+C** keypress.
- **Blocked**: Used when a dependency was not met and the following event could not start. Note that a "dependency" doesn't strictly mean order dependency or setup dependency. It could also mean cases where the next event to be carried out is no longer relevant.
- **Skipped**: Used when a scheduled item was not executed and was omitted. The difference between skipped and blocked is that skipped is voluntary, whereas blocked is collateral.
- **Errored**: A mistake or inaccuracy (for example, an unexpected exception). The difference between failed and errored is that failed represents carrying out an event as planned with the result not meeting expectation, whereas errored means something went wrong in the course of carrying out that procedure.
- **Passx**: Short for "passed with exception." Use passx with caution because you are effectively re-marking a failed result as passed, even though there was an exception.

Reporter

Reporter is a package for collecting and generating test reports in YAML format. This results file contains all the details about execution (section hierarchy, time, results, and so on).

The results.json report contains hierarchical information about the pyATS job executed. The top level is the TestSuite, which contains information about the job as a whole. Under

the TestSuite are all of the tasks executed as a part of the job. Each task then has the various sections of testing underneath CommonSetup, CommonCleanup, and Testcases. These then have child sections, such as TestSection, SetupSection, CleanupSection, and Subsection. The children of these would be steps, which can be nested with their own children steps.

Being able to parse the generated test reports (results.json) allows you to further dig into and programmatically analyze the test results. This allows you to take further action based on the testing results using an automated workflow.

Utilities

pyATS comes with a variety of additional utilities to enhance and support the framework:

- **Find:** Used to search and filter against objects
- **Secret Strings:** Used to protect and encrypt strings (such as passwords)
- **Multiprotocol file transfer utilities:** Used to transfer files to/from a remote server
- **Embedded pyATS file transfer server:** Supports FTP, TFTP, SCP, and HTTP
- **Import utilities:** Used to translate "x.y.z"-style strings into "from x.y import z" and then return z
- **YAML File Markups:** pyATS-specific YAML markup; similar to Django template language

Robot Framework Support

Robot Framework is generic Python/Java test automation open-source framework that focuses on acceptance test automation using an English-like keyword-driven test approach. https://robotframework.org/

You can now freely integrate pyATS into the Robot Framework, and vice versa:

- You can run Robot Framework scripts directly within Easypy, save runtime logs under the runinfo directory, and aggregate results into Easypy report.
- You can leverage the pyATS infrastructure and libraries within Robot Framework scripts.

However, because Robot Framework support is an optional component under pyATS, you must install the package explicitly before being able to leverage it. For more information, see Chapter 22, "Robot Framework."

Manifest

The pyATS Manifest is a file with YAML syntax describing how and where to execute a script. It is intended to formally describe the execution of a single script, including the runtime environment, script arguments, and the profile(s) that define environment specific settings and arguments. Profiles allow the same script to be run against multiple environments or run with different input parameters—for example, using multiple testbeds representing different environments (testing/production) or different scaling numbers to test scalability and resiliency. A script can be executed via the manifest using the **pyats run manifest** command. Manifest files use the file extension .tem, which stands for Test Execution Manifest. Manifest files can be tracked via source control, which can help standardize testing environments across multiple testing scenarios.

Summary

Adopting network automation has never been easier with the introduction of a modern tool like pyATS, and its associated framework, and the test-driven development methodology. TDD emerged from the Agile manifesto and is a common form of software development that can be extended to network automation. Network engineers will gather business requirements and transform them into testcases. These testcases are small unit tests that at first fail when they are written. The smallest, simplest amount of code possible is applied to make the test pass, while all other tests remain passing, and is refactored until the developer is satisfied with the passing test. This iterative approach is performed for each use case until test coverage is established for all business requirements.

The iterative essence of TDD fosters a disciplined, incremental, and feedback-driven approach to both software and network automation development. This iterative process, initiated from gathering business requirements, which are then transmuted into test cases, is at the heart of promoting a robust and reliable network automation culture. The rhythm of writing a failing test, making it pass with the simplest code, and then refining the code, embodies a cycle of continuous improvement and validation:

- **Evolution of tests**: As development progresses, tests evolve in tandem. Initially, tests are rudimentary, focusing on basic functionalities. Over time, as more features are integrated and complexities arise, tests become more comprehensive and nuanced. This evolution of tests is a natural reflection of the growing understanding and unfolding of business requirements.
- **Improved code quality**: One of the stellar benefits of this iterative approach is the uplift in code quality. Each cycle of TDD pushes the code through a crucible of validation, ensuring that it not only meets the immediate requirement but is also robust and resilient to potential issues. The refactoring stage, an integral part of the TDD cycle, further polishes the code, enhancing its readability, efficiency, and maintainability.