Test Shell for Interactive Environments

This document describes how to use the `TestShell` submodule in the functional test suite.

The `TestShell` submodule extends the `BitcoinTestFramework` functionality to external interactive environments for prototyping and educational purposes. Just like `BitcoinTestFramework`, the `TestShell` allows the user to:

- Manage regtest bitcoind subprocesses.
- Access RPC interfaces of the underlying bitcoind instances.
- Log events to the functional test logging utility.

The `TestShell` can be useful in interactive environments where it is necessary to extend the object lifetime of the underlying `BitcoinTestFramework` between user inputs. Such environments include the Python3 command line interpreter or Jupyter notebooks running a Python3 kernel.

1. Requirements

- Python3
- `bitcoind` built in the same repository as the `TestShell`.

2. Importing `TestShell` from the Bitcoin Core repository

We can import the `TestShell` by adding the path of the Bitcoin Core `test_framework` module to the beginning of the PATH variable, and then importing the `TestShell` class from the `test_shell` sub-package.

```
>>> import sys
>>> sys.path.insert(0, "/path/to/bitcoin/test/functional")
>>> from test_framework.test_shell import TestShell
```

The following `TestShell` methods manage the lifetime of the underlying bitcoind processes and logging utilities.

- `TestShell.setup()`
- `TestShell.shutdown()`

The `TestShell` inherits all `BitcoinTestFramework` members and methods, such as: * `TestShell.nodes[index].rpc_method()` * `TestShell.log.info("Custom log message")`

The following sections demonstrate how to initialize, run, and shut down a `TestShell` object.

3. Initializing a `TestShell` object

```
>>> test = TestShell().setup(num_nodes=2, setup_clean_chain=True)
20XX-XX-XXTXX:XX:XX.XXXXXXX TestFramework (INFO): Initializing test directory /path/to/bitco
```

The `TestShell` forwards all functional test parameters of the parent `BitcoinTestFramework` object. The full set of argument keywords which can be used to initialize the `TestShell` can be found in section #6 of this document.

Note: Running multiple instances of `TestShell` is not allowed. Running a single process also ensures that logging remains consolidated in the same temporary folder. If you need more bitcoind nodes than set by default (1), simply increase the `num_nodes` parameter during setup.

```
>>> test2 = TestShell().setup()
TestShell is already running!
```

4. Interacting with the `TestShell`

Unlike the `BitcoinTestFramework` class, the `TestShell` keeps the underlying Bitcoind subprocesses (nodes) and logging utilities running until the user explicitly shuts down the `TestShell` object.

During the time between the `setup` and `shutdown` calls, all `bitcoind` node processes and `BitcoinTestFramework` convenience methods can be accessed interactively.

Example: Mining a regtest chain

By default, the `TestShell` nodes are initialized with a clean chain. This means that each node of the `TestShell` is initialized with a block height of 0.

```
>>> test.nodes[0].getblockchaininfo()["blocks"]
0
```

We now let the first node generate 101 regtest blocks, and direct the coinbase rewards to a wallet address owned by the mining node.

```
>>> address = test.nodes[0].getnewaddress()
>>> test.self.generatetoaddress(nodes[0], 101, address)
['2b98dd0044aae6f1cca7f88a0acf366a4bfe053c7f7b00da3c0d115f03d67efb', ...
```

Since the two nodes are both initialized by default to establish an outbound connection to each other during `setup`, the second node's chain will include the mined blocks as soon as they propagate.

```
>>> test.nodes[1].getblockchaininfo()["blocks"]
101
```

The block rewards from the first block are now spendable by the wallet of the first node.

```
>>> test.nodes[0].getbalance()
Decimal('50.00000000')
```

We can also log custom events to the logger.

```
>>> test.nodes[0].log.info("Successfully mined regtest chain!")
20XX-XX-XXTXX:XX:XX.XXXXXXX TestFramework.node0 (INFO): Successfully mined regtest chain!
```

Note: Please also consider the functional test readme, which provides an overview of the test-framework. Modules such as key.py, script.py and messages.py are particularly useful in constructing objects which can be passed to the bitcoind nodes managed by a running `TestShell` object.

5. Shutting the `TestShell` down

Shutting down the `TestShell` will safely tear down all running bitcoind instances and remove all temporary data and logging directories.

```
>>> test.shutdown()
20XX-XX-XXTXX:XX:XX.XXXXXXX TestFramework (INFO): Stopping nodes
20XX-XX-XXTXX:XX:XX.XXXXXXX TestFramework (INFO): Cleaning up /path/to/bitcoin_func_test_XXX
20XX-XX-XXTXX:XX:XX.XXXXXXX TestFramework (INFO): Tests successful
```

To prevent the logs from being removed after a shutdown, simply set the `TestShell.options.nocleanup` member to `True`.

```
>>> test.options.nocleanup = True
>>> test.shutdown()
20XX-XX-XXTXX:XX:XX.XXXXXXX TestFramework (INFO): Stopping nodes
20XX-XX-XXTXX:XX:XX.XXXXXXX TestFramework (INFO): Not cleaning up dir /path/to/bitcoin_func
20XX-XX-XXTXX:XX:XX.XXXXXXX TestFramework (INFO): Tests successful
```

The following utility consolidates logs from the bitcoind nodes and the underlying `BitcoinTestFramework`:

- `/path/to/bitcoin/test/functional/combine_logs.py '/path/to/bitcoin_func_test_XXXXXXX'`

6. Custom `TestShell` parameters

The `TestShell` object initializes with the default settings inherited from the `BitcoinTestFramework` class. The user can override these in `TestShell.setup(key=value)`.

Note: `TestShell.reset()` will reset test parameters to default values and can be called after the TestShell is shut down.

Test parameter key	Default Value	Description
`bind_to_localhost_only`	`True`	Binds bitcoind RPC services to `127.0.0.1` if set to `True`.
`cachedir`	`"/path/to/bitcoin/test/cache"`	Sets the bitcoind datadir directory.

Test parameter key	Default Value	Description
`chain`	`"regtest"`	Sets the chain-type for the underlying test bitcoind processes.
`configfile`	`"/path/to/bitcoin/test/config.ini"`	Sets the location of the test framework config file.
`coveragedir`	`None`	Records bitcoind RPC test coverage into this directory if set.
`loglevel`	`INFO`	Logs events at this level and higher. Can be set to `DEBUG`, `INFO`, `WARNING`, `ERROR` or `CRITICAL`.
`nocleanup`	`False`	Cleans up temporary test directory if set to `True` during `shutdown`.
`noshutdown`	`False`	Does not stop bitcoind instances after `shutdown` if set to `True`.
`num_nodes`	`1`	Sets the number of initialized bitcoind processes.
`perf`	False	Profiles running nodes with `perf` for the duration of the test if set to `True`.
`rpc_timeout`	`60`	Sets the RPC server timeout for the underlying bitcoind processes.
`setup_clean_chain`	`False`	A 200-block-long chain is initialized from cache by default. Instead, `setup_clean_chain` initializes an empty blockchain if set to `True`.

Test parameter key	Default Value	Description
`randomseed`	Random Integer	`TestShell.options.randomseed` is a member of `TestShell` which can be accessed during a test to seed a random generator. User can override default with a constant value for reproducible test runs.
`supports_cli`	`False`	Whether the bitcoin-cli utility is compiled and available for the test.
`tmpdir`	`"/var/folders/.../"`	Sets directory for test logs. Will be deleted upon a successful test run unless `nocleanup` is set to `True`
`trace_rpc`	`False`	Logs all RPC calls if set to `True`.
`usecli`	`False`	Uses the bitcoin-cli interface for all bitcoind commands instead of directly calling the RPC server. Requires `supports_cli`.