Functional tests

Writing Functional Tests

Example test The file test/functional/example_test.py is a heavily commented example of a test case that uses both the RPC and P2P interfaces. If you are writing your first test, copy that file and modify to fit your needs.

Coverage Running `test/functional/test_runner.py` with the `--coverage` argument tracks which RPCs are called by the tests and prints a report of uncovered RPCs in the summary. This can be used (along with the `--extended` argument) to find out which RPCs we don't have test cases for.

Style guidelines

- Where possible, try to adhere to PEP-8 guidelines
- Use a python linter like flake8 before submitting PRs to catch common style nits (eg trailing whitespace, unused imports, etc)
- The oldest supported Python version is specified in doc/dependencies.md. Consider using pyenv, which checks .python-version, to prevent accidentally introducing modern syntax from an unsupported Python version. The CI linter job also checks this, but possibly not in all cases.
- See the python lint script that checks for violations that could lead to bugs and issues in the test code.
- Use type hints in your code to improve code readability and to detect possible bugs earlier.
- Avoid wildcard imports
- Use a module-level docstring to describe what the test is testing, and how it is testing it.
- When subclassing the BitcoinTestFramework, place overrides for the `set_test_params()`, `add_options()` and `setup_xxxx()` methods at the top of the subclass, then locally-defined helper methods, then the `run_test()` method.
- Use `f'{x}'` for string formatting in preference to `'{}'.format(x)` or `'%s' % x`.

Naming guidelines

- Name the test `<area>_test.py`, where area can be one of the following:
  - `feature` for tests for full features that aren't wallet/mining/mempool, eg `feature_rbf.py`
  - `interface` for tests for other interfaces (REST, ZMQ, etc), eg `interface_rest.py`
  - `mempool` for tests for mempool behaviour, eg `mempool_reorg.py`
  - `mining` for tests for mining features, eg `mining_prioritisetransaction.py`
  - `p2p` for tests that explicitly test the p2p interface, eg `p2p_disconnect_ban.py`

    - `rpc` for tests for individual RPC methods or features, eg `rpc_listtransactions.py`
    - `tool` for tests for tools, eg `tool_wallet.py`
    - `wallet` for tests for wallet features, eg `wallet_keypool.py`
- Use an underscore to separate words
    - exception: for tests for specific RPCs or command line options which don't include underscores, name the test after the exact RPC or argument name, eg `rpc_decodescript.py`, not `rpc_decode_script.py`
- Don't use the redundant word `test` in the name, eg `interface_zmq.py`, not `interface_zmq_test.py`

General test-writing advice

- Instead of inline comments or no test documentation at all, log the comments to the test log, e.g. `self.log.info('Create enough transactions to fill a block')`. Logs make the test code easier to read and the test logic easier to debug.
- Set `self.num_nodes` to the minimum number of nodes necessary for the test. Having additional unrequired nodes adds to the execution time of the test as well as memory/CPU/disk requirements (which is important when running tests in parallel).
- Avoid stop-starting the nodes multiple times during the test if possible. A stop-start takes several seconds, so doing it several times blows up the runtime of the test.
- Set the `self.setup_clean_chain` variable in `set_test_params()` to `True` to initialize an empty blockchain and start from the Genesis block, rather than load a premined blockchain from cache with the default value of `False`. The cached data directories contain a 200-block pre-mined blockchain with the spendable mining rewards being split between four nodes. Each node has 25 mature block subsidies (25x50=1250 BTC) in its wallet. Using them is much more efficient than mining blocks in your test.
- When calling RPCs with lots of arguments, consider using named keyword arguments instead of positional arguments to make the intent of the call clear to readers.
- Many of the core test framework classes such as `CBlock` and `CTransaction` don't allow new attributes to be added to their objects at runtime like typical Python objects allow. This helps prevent unpredictable side effects from typographical errors or usage of the objects outside of their intended purpose.

RPC and P2P definitions Test writers may find it helpful to refer to the definitions for the RPC and P2P messages. These can be found in the following source files:

- `/src/rpc/*` for RPCs
- `/src/wallet/rpc*` for wallet RPCs

- `ProcessMessage()` in `/src/net_processing.cpp` for parsing P2P messages

Using the P2P interface

- `P2P`s can be used to test specific P2P protocol behavior. p2p.py contains test framework p2p objects and messages.py contains all the definitions for objects passed over the network (`CBlock`, `CTransaction`, etc, along with the network-level wrappers for them, `msg_block`, `msg_tx`, etc).
- P2P tests have two threads. One thread handles all network communication with the bitcoind(s) being tested in a callback-based event loop; the other implements the test logic.
- `P2PConnection` is the class used to connect to a bitcoind. `P2PInterface` contains the higher level logic for processing P2P payloads and connecting to the Bitcoin Core node application logic. For custom behaviour, subclass the P2PInterface object and override the callback methods.

`P2PConnection`s can be used as such:

```
p2p_conn = node.add_p2p_connection(P2PInterface())
p2p_conn.send_and_ping(msg)
```

They can also be referenced by indexing into a `TestNode`'s `p2ps` list, which contains the list of test framework `p2p` objects connected to itself (it does not include any `TestNode`s):

```
node.p2ps[0].sync_with_ping()
```

More examples can be found in p2p_unrequested_blocks.py, p2p_compactblocks.py.

Prototyping tests The `TestShell` class exposes the BitcoinTestFramework functionality to interactive Python3 environments and can be used to prototype tests. This may be especially useful in a REPL environment with session logging utilities, such as IPython. The logs of such interactive sessions can later be adapted into permanent test cases.

Test framework modules

The following are useful modules for test developers. They are located in test/functional/test_framework/.

authproxy.py Taken from the python-bitcoinrpc repository.

test_framework.py Base class for functional tests.

util.py Generally useful functions.

p2p.py Test objects for interacting with a bitcoind node over the p2p interface.

script.py Utilities for manipulating transaction scripts (originally from python-bitcoinlib)

key.py Test-only secp256k1 elliptic curve implementation

blocktools.py Helper functions for creating blocks and transactions.

Benchmarking with perf

An easy way to profile node performance during functional tests is provided for Linux platforms using `perf`.

Perf will sample the running node and will generate profile data in the node's datadir. The profile data can then be presented using `perf report` or a graphical tool like hotspot.

There are two ways of invoking perf: one is to use the `--perf` flag when running tests, which will profile each node during the entire test run: perf begins to profile when the node starts and ends when it shuts down. The other way is the use the `profile_with_perf` context manager, e.g.

```
with node.profile_with_perf("send-big-msgs"):
    # Perform activity on the node you're interested in profiling, e.g.:
    for _ in range(10000):
        node.p2ps[0].send_message(some_large_message)
```

To see useful textual output, run

```
perf report -i /path/to/datadir/send-big-msgs.perf.data.xxxx --stdio | c++filt | less
```

See also:

- Installing perf
- Perf examples
- Hotspot: a GUI for perf output analysis