Populus Documentation

Release 2.0.0-alpha.1

Piper Merriam

Dec 11, 2017

Contents

Populus is a smart contract development framework for the Ethereum blockchain.

CHAPTER 1

Contents

1.1 Quickstart

Welcom to Populus! Populus has (almost) everything you need for Ethereum blockchain development.

1.1.1 System Dependencies

Populus depends on the following system dependencies.

- The Solidity Compiler : Contracts are authored in the Solidity language, and then compiled to the bytecode of the Ethereum Virtual Machine (EVM).
- Geth: The official Go implementation of the Ethereum protocol. The Geth client runs a blockchain node, lets you interact with the blockchain, and also runs and deploys to the test blockchains during development.

In addition, populus needs some system dependencies to be able to install the PyEthereum library.

Debian, Ubuntu, Mint

```
sudo apt-get install libssl-dev
```

Fedora, CentOS, RedHat

```
sudo yum install openssl-devel
```

OSX

```
brew install pkg-config libffi autoconf automake libtool openssl
```

1.1.2 Install Populus

Populus can be installed using `pip` as follows.

```
$ pip install populus
```

If you are installing on Ubuntu, and working with python3 (recommended):

```
$ pip3 install populus
```

Note: With Ubuntu, use Ubuntu's pip:

`$sudo apt-get install python-pip`

or, for python 3:

`$sudo apt-get install python3-pip`

You may need to install populus with sudo: `$ sudo -H pip install populus`

Installation from source can be done from the root of the project with the following command.

```
$ python setup.py install
```

Verify your installation

```
$ populus

Usage: populus [OPTIONS] COMMAND [ARGS]...

  Populus

Options:
  -p, --project PATH  Specify a populus project directory
  -l, --logging TEXT  Specify the logging level.  Allowed values are
                      DEBUG/INFO or their numeric equivalents 10/20
  -h, --help          Show this message and exit.

Commands:
  chain    Manage and run ethereum blockchains.
  compile  Compile project contracts, storing their...
  config   Manage and run ethereum blockchains.
  deploy   Deploys the specified contracts to a chain.
  init     Generate project layout with an example...
```

Great. Let's have the first populus project.

1.1.3 A Word of Caution

Populus is a development environment. It was designed to make things easy and fun for the Python Ethereum developer. We use test blockchains, demo accounts, simple passwords, everything that is required to help you focus on the code.

But once the code is ready for work and deployment with real Eth, you should be careful. As there is a clear difference between running your iOS app in the Xcode simulator to the real actions of the app on the iPhone, or to take another example, between running a website localy on 127.0.0.1 vs. running it on a real server which is opened to the entire internet, there **is** a difference between blockchain development environment, and when you deploy and send real Eth.

The core issue, as a developer, is that once you unlock an account, there is a running process with access to your Eth. Any mistake or security breach can cause loosing this Eth. This is not an issue with test blockchains and test tokens, but with real Eth it is.

As a rule:

[1] When you unlock an account to use real Ether, the unlocked account balance should have only the minimum Ether required for gas and the money transfers you unlocked if for. Ethereum accounts are free, so it will cost you nothing to have a dedicated account for your development, where you will send only the Ether you need it for, from time to time.

[2] Never unlock a real Ether account on a remote node. You can use a remote node, but not for actions that require an unlocked account. When you unlock an account on a remote node, anybody with access to the node has access to your funds. In other words, the geth instance that unlocked your acount should run on your local protected machine.

[3] Geth allows you to provide a password file, which is handy (more on it later). The password file should be protected by permissions.

Luckily, there are simple and effective practices to keep your Eth safe. See Protecting yourself and your funds

1.1.4 Initializing a New Project

Populus can initialize your project using the `$ populus init` command.

```
$ populus init
Wrote default populus configuration to `./populus.json`.
Created Directory: ./contracts
Created Example Contract: ./contracts/Greeter.sol
Created Directory: ./tests
Created Example Tests: ./tests/test_greeter.py
```

Your project will now have a `./contracts` directory with a single Solidity source file in it named `Greeter.sol`, as well as a `./tests` directory with a single test file named `test_greeter.py`.

Alternatively, you can init a new project by a directory:

```
$ populus -p /path/to/my/project/ init
```

1.1.5 Compiling your contracts

Before you compile our project, lets take a look at the `Greeter` contract that is generated as part of the project initialization.

```
$ nano contracts/Greeter.sol
```

Note: Check your IDE for Solidity extention/package.

Here is the contract:

```
pragma solidity ^0.4.11;

contract Greeter {
    string public greeting;

    function Greeter() {
        greeting = "Hello";
    }

    function setGreeting(string _greeting) public {
        greeting = _greeting;
    }

    function greet() constant returns (string) {
        return greeting;
    }
}
```

`Greeter` is a simple contract:

- The `contract` keyword starts a contract definition
- The contract has one public "state" variable, named `greeting`.
- The contract constructor function, `function Greeter()`, which has the same name of the contract, initializes with a default greeting of the string `'Hello'`.
- The `greet` function is exposed, and returns whatever string is set as the greeting,
- The `setGreeting` function is available, and allows the greeting to be changed.

You can now compile the contract using `$ populus compile`

```
$ populus compile
============ Compiling ==============
> Loading source files from: ./contracts

> Found 1 contract source files
- contracts/Greeter.sol

> Compiled 1 contracts
- Greeter

> Wrote compiled assets to: ./build/contracts.json
```

For compiling outside the project directory use:

```
$ populus -p /path/to/my/project/ compile
```

The build/contracts.json file contains a lot of information that the Solidity compiler produced. This is required to deploy and work with the contract. Some important info is the application binary interface (ABI) of the contract, which will allow to call it's functions after it's compiled, and the bytecode required to deploy the contract, and the bytecode that will run once the contract sits on the blockchain.

1.1.6 Testing your contract

Now that you have a basic contract you'll want to test that it behaves as expected. The project should already have a test module named `test_greeter.py` located in the `./tests` directory that looks like the following.

```
def test_greeter(chain):
    greeter, _ = chain.provider.get_or_deploy_contract('Greeter')

    greeting = greeter.call().greet()
    assert greeting == 'Hello'

def test_custom_greeting(chain):
    greeter, _ = chain.provider.get_or_deploy_contract('Greeter')

    set_txn_hash = greeter.transact().setGreeting('Guten Tag')
    chain.wait.for_receipt(set_txn_hash)

    greeting = greeter.call().greet()
    assert greeting == 'Guten Tag'
```

You should see two tests, one that tests the default greeting, and one that tests that we can set a custom greeting.

Note that both test functions accept a `chain` argument. This "chain" is actually a py.test fixture , provided by the populus pytest plugin. The chain in the tests is a populus "chain" object that runs a temporary blockchain called "tester". The tester chain is ephemeral. All blockchain state is reset at the beginning of each test run and is only stored in memory, so obviously not usable for long term runnig contracts, but great for testing.

You can run tests using the `py.test` command line utility which was installed when you installed populus.

```
$ py.test tests/
collected 2 items

tests/test_greeter.py::test_greeter PASSED
tests/test_greeter.py::test_custom_greeting PASSED
```

You should see something akin to the output above with three passing tests.

Finally, similarly to the tests deployment, test the same deployment from the command line:

```
$ populus deploy --chain tester --no-wait-for-sync
> Found 1 contract source files
- contracts/Greeter.sol
> Compiled 1 contracts
- contracts/Greeter.sol:Greeter
Please select the desired contract:

    0: Greeter
```

Type 0 at the prompt, and enter.

```
Beginning contract deployment.  Deploying 1 total contracts (1 Specified, 0 because␣
↪of library dependencies).

Greeter
Deploying Greeter
Deploy Transaction Sent:␣
↪0x84d23fa8c38a09a3b29c4689364f71343058879639a617763ce675a336033bbe
Waiting for confirmation...

Transaction Mined
=================
Tx Hash      : 0x84d23fa8c38a09a3b29c4689364f71343058879639a617763ce675a336033bbe
Address      : 0xc305c901078781c232a2a521c2af7980f8385ee9
Gas Provided : 465729
Gas Used     : 365729


Verified contract bytecode @ 0xc305c901078781c232a2a521c2af7980f8385ee9
Deployment Successful.
```

Nice. Of course, since this is an ad-hoc "tester" chain, it quits immediately, and nothing is really saved. But the deployment works and should work on a permanent blockchain, like the mainnet or testnet.

Again, outside the project directory use:

```
$ populus -p /path/to/my/project/ deploy --chain tester --no-wait-for-sync
```

1.1.7 Setup for development and contribution

In order to configure the project locally and get the whole test suite passing, you'll need to make sure you're using the proper version of the `solc` compiler. Follow these steps to install all the dependencies:

Virtual environment

If you don't already have it, go ahead and install `virtualenv` with `pip install virtualenv`. You can then create and activate your Populus environment with the following commands:

```
$ cd populus
$ virtualenv populus
$ source populus/bin/activate
```

This allows you to install the specific versions of the Populus dependencies without conflicting with global installations you may already have on your machine.

Install dependencies

Now, run the following commands to install all the dependencies specified in the project except for `solc`:

```
$ pip install -r requirements-dev.txt
$ pip install -r requirements-docs.txt
$ pip install -e .
```

Install Solidity

You'll have to install solidity, recommended from release 0.4.11 or greater.

Installation scripts for binary:

https://github.com/ethereum/py-solc#installing-the-solc-binary

Installation scripts building it:

First, clone the repository and switch to the proper branch:

```
$ git clone --recursive https://github.com/ethereum/solidity.git
$ cd solidity
$ git checkout release_0.4.13
```

You can also download the tar or zip file at:

https://github.com/ethereum/solidity/releases

Note: Use the tar.gz file to build from source, and make sure, after extracting the file, that the "deps" directory is not empty and actually contains the dependencies.

If you're on a Mac, you may need to accept the Xcode license as well. Make sure you have the latest version installed, and if you run into errors, try the following:

```
$ sudo xcodebuild -license accept
```

If you're on Windows, make sure you have Git, CMake, and Visual Studio 2015.

Now, install all the external dependencies. For Mac:

```
$ ./scripts/install_deps.sh
```

Or, for Windows:

```
$ scripts\install_deps.bat
```

Finally, go ahead and build Solidity. For Mac:

```
$ mkdir build
$ cd build
$ cmake .. && make
```

Or, for Windows:

```
$ mkdir build
$ cd build
$ cmake -G "Visual Studio 14 2015 Win64" ..
```

The following command will also work for Windows:

```
$ cmake --build . --config RelWithDebInfo
```

Confirm

This should have installed everything you need, but let's be sure. First, try running:

```
$ which solc
```

If you didn't see any output, you'll need to move the `solc` executable file into the directory specified in your `PATH`, or add an accurate `PATH` in your `bash` profile. If you can't find the file, you may need to run:

```
$ npm install -g solc
```

This should install the executable wherever your Node packages live.

Once you see output from the `which solc` command (and you're in the Populus directory with the `virtualenv` activated), you're ready to run the tests.

```
$ py.test tests/
```

At this point, all your tests should pass. If they don't, you're probably missing a dependency somewhere. Just retrace your steps and you'll figure it out.

1.2 Overview

1.2.1 Introduction

The primary interface to populus is the command line command `$ populus`.

1.2.2 Command Line Options

```
$ populus
Usage: populus [OPTIONS] COMMAND [ARGS]...

  Populus

Options:
  -p, --project DIRECTORY  Specify a populus project directory to be used.
  -h, --help               Show this message and exit.

Commands:
  chain          Manage and run ethereum blockchains.
  compile        Compile project contracts, storing their...
  deploy         Deploys the specified contracts to a chain.
  init           Generate project layout with an example...
  makemigration  Generate an empty migration.
  migrate        Run project migrations
```

1.2.3 Project Layout

By default Populus expects a project to be laid out as follows:

```
- project root
    - populus.json
    - build (automatically created during compilation)
    |   - contracts.json
    - contracts
    |   - MyContract.sol
    |   - ....
    - tests
        - test_my_contract.py
        - test_some_other_tests.py
        - ....
        - ....
```

Initialize

```
$ populus init --help
Usage: populus init [OPTIONS]

  Generate project layout with an example contract.

Options:
  -h, --help  Show this message and exit.
```

Running `$ populus init` will initialize the current directory with the default project layout that populus uses. If `-p` argument is provided, populus will init to that directory

- `./contracts/`
- `./contracts/Greeter.sol`
- `./tests/test_greeter.py`

You can also init a project from another directory with:

```
$ populus -p /path/to/my/project/ init
```

1.3 Tutorial

Learn how to use populus by working your way through the following tutorials.

1.3.1 Contents

Part 1: Basic Testing

Introduction

The following tutorial picks up where the quickstart leaves off. You should have a single solidity contract named `Greeter` located in `./contracts/Greeter.sol` and a single test module `./tests/test_greeter.py` that contains two tests.

Modify our Greeter

Lets add way for the Greeter contract to greet someone by name. We'll do so by adding a new function `greet(bytes name)` which you can see below. Update your solidity source to match this updated version of the contract.

```
pragma solidity ^0.4.11;


contract Greeter {
    string public greeting;

    function Greeter() {
        greeting = "Hello";
    }

    function setGreeting(string _greeting) public {
        greeting = _greeting;
    }

    function greet() constant returns (string) {
        return greeting;
    }

    function greet(bytes name) constant returns (bytes) {
        // create a byte array sufficiently large to store our greeting.
        bytes memory namedGreeting = new bytes(
            name.length + 1 + bytes(greeting).length
```

```
        );

        // push the greeting onto our return value.
        // greeting.
        for (uint i=0; i < bytes(greeting).length; i++) {
            namedGreeting[i] = bytes(greeting)[i];
        }

        // add a space before pushing the name on.
        namedGreeting[bytes(greeting).length] = ' ';

        // loop over the name and push all of the characters onto the
        // greeting.
        for (i=0; i < name.length; i++) {
            namedGreeting[bytes(greeting).length + 1 + i] = name[i];
        }
        return namedGreeting;
    }
}
```

Testing our changes

Now we'll want to test our contract. Lets add another test to `./tests/test_greeter.py` so that the file looks as follows.

```
def test_greeter(chain):
    greeter, _ = chain.provider.get_or_deploy_contract('Greeter')

    greeting = greeter.call().greet()
    assert greeting == 'Hello'

def test_custom_greeting(chain):
    greeter, _ = chain.provider.get_or_deploy_contract('Greeter')

    set_txn_hash = greeter.transact().setGreeting('Guten Tag')
    chain.wait.for_receipt(set_txn_hash)

    greeting = greeter.call().greet()
    assert greeting == 'Guten Tag'

def test_named_greeting(chain):
    greeter, _ = chain.provider.get_or_deploy_contract('Greeter')

    greeting = greeter.call().greet('Piper')
    assert greeting == 'Hello Piper'
```

You can run tests using the `py.test` command line utility which was installed when you installed populus.

```
$ py.test tests/
collected 3 items

tests/test_greeter.py::test_greeter PASSED
tests/test_greeter.py::test_custom_greeting PASSED
tests/test_greeter.py::test_named_greeting PASSED
```

You should see something akin to the output above with three passing tests.

Behind the scenes, populus uses a pytest plugin that creates a populus project object, and then provide this object, (and it's derived objects), to the test functions via a pytest fixture.

Thus, tests run for a specific project.

If you run py.test from within the project directory, populus will assume that the current working directory is the project you want to test, and the fixtures will be based on this directory.

The same is true if you provide pytest one positional argument for testing, which is the project directory:

```
$ py.test /path/to/my/project/
```

Here, populus will provide the fixtures based on the project at `/path/to/my/project/`. Pytest will also find the tests in that directory.

If the tests are at `/path/to/tests/`, then you can set the tested *project* directory as follows:

1. As a command line argument: `$ py.test /path/to/tests/ --populus-project /path/to/my/project/`
2. In a pytest.ini file, with the following entry: `populus_project=/path/to/my/project/`
3. With an environment variable: `PYTEST_POPULUS_PROJECT`. E.g., `$ export PYTEST_POPULUS_PROJECT=/path/to/my/project/`

If you used method 2 or 3, that is with pytest.ini or an environment variable, then:

```
$ py.test /path/to/tests/
```

Will do, and populus will figure out the testing project from pytest.ini or the environment variable. The tests found at `/path/to/tests/` will be applied to this project.

Note: For pytest.ini files, make sure the file is in the right location, and that py.test actually picks it. See https://docs.pytest.org/en/latest/customize.html#initialization-determining-rootdir-and-inifile .

So by providing explicit project for testing, you can run tests from one project on another, or if all your projects provide a repeating functionality, you can use the same set of tests for all of them.

Part 2: Local Chains

Introduction

In part 1 of the tutorial we modified our `Greeter` contract and expanded the test suite to cover the new functionality.

In this portion of the tutorial, we will explore the ability of populus to deploy the contracts to the blockchain.

One of the nice things about Ethereum is the *protocol*, a protocol which specifies how to run a blockchain. Theoretically, you can use this exact protocol to run your own blockchain, which is private and totally seprated from the mainnet Ethereum blockchain and it's nodes.

Although the private Ether you will mint in this private blockchain are not worth a lot in the outside world (but who knows?), such private, local blockchain is a great tool for development. It simulates the real ethereum blockchain, precisely, with the same protocol, but without the overhead of syncing, running and waiting for the real testnet blockchain connections and data.

Once the contract is ready, working, and tested on your local chain, you can deploy it to the distributed ethereum blockchains: testnet, the testing network, and then to the real Ethereum blockchain, with real money and by paying real gas (in Eth) for the deployment.

In this tutorial we will create, and deploy to, a local chain we'll name "horton".

Create a *Local* chain

To create a new local chain in your project directory type:

```
$ populus chain new horton
```

Check your project directory, which should look as follows:

```
- chains
|   - horton
|       - chain_data
|       |   - keystore
|       |       - UTC--2017-10-10T11-36-58.745314398Z--
↪eb9ae85fa8f6d8853afe45799d966dca89edd9aa
|       - genesis.json
|       - init_chain.sh
|       - password
|       - run_chain.sh
- contracts
|   - Greeter.sol
- project.json
- tests
    - test_greeter.py
```

Let's unpack this.

First, we see the project's known structure: the project config file `project.json`, and the `contracts` and `tests` directories.

The `chain new` command added a new directory: *chains*, and within it, the *horton* chain.

The `chain_data` directory is where geth will store the blockchain data. It's empty, for now, since we didn't run this local chain at all.

Wallets

Then, the `keystore` directory, where the "wallets" are saved. A wallet file is a special file that stores an ethereum account. Here we see one wallet file, `UTC--2017-10-10T11-36-58.745314398Z--eb9ae85fa8f6d8853afe45799d966dca89edd9aa`. The first part of the wallet file name

is the time that the wallet was created, and the second part is the account address. You should see similar file name on your machine, but of course with a different time-stamp and a different address.

Here is the wallet:

```
 $ cat chains/horton/chain_data/keystore/UTC--2017-10-10T11-36-58.745314398Z--
↪eb9ae85fa8f6d8853afe45799d966dca89edd9aa

 {
 "address":"eb9ae85fa8f6d8853afe45799d966dca89edd9aa",
 "crypto":{
   "cipher":"aes-128-ctr","ciphertext":
↪"202bee426c48fd087e19a351d207f74903a437ea74cff5f7491ed0b82a591737",
   "cipherparams":{
     "iv":"02c18eab6f32875de56cb4452f7d4fa8"
     },
     "kdf":"scrypt",
     "kdfparams":{
       "dklen":32,"n":262144,"p":1,"r":8,"salt":
↪"747653d095958f26666dd90a91b26bf00d0d848b37f9df26ad68badd004ee88f"
     },
     "mac":"ac8d6afd19a4dbd55b67ef94d31bb323f037346f6973b60b4948d5ab6ba7f6de"
   },
   "id":"9542872c-6855-4dcc-b45d-8654009a89c3",
   "version":3
}
```

The wallet doesn't save any info regarding the account balance, transactions, etc - this info is saved on the blockchain. It does, however, allows you to unlock an account, send Ethereum, and run transactions with this account.

The wallet file is encrypted with a password. To unlock the account in the wallet, geth requires the password. Here, populus saved the password in a password file:

```
$ cat chains/horton/password
this-is-not-a-secure-password
```

The default password we used, tells. It's designated for development and testing, not when using real Eth.

Why to save the password in a file *at all*? Because you can provide this file path to geth with the `password` commnad line argument. Otherwise, you will have to manually enter the password each time geth starts. Moreover, sometimes it's hard to spot the password prompt with all the info that geth spits. So a password file is more convinient, but obviously should be fully secured, with the right permissions.

Accounts

Populus created the account for you, but you can create more accounts with `$ geth account new`. You can keep as many wallets as you want in the keystore. One wallet, which you can set, is the primary default account, called "etherbase" or "coinbase". You can use any wallet you save in the keystore, as long as you have the password to unlock it. See geth accounts managment .

Note: The terms "create an account", or "new account", may be missleading. Nobody "creates" an account, since all the possible alphanumeric combinations of a valid Ethereum account address are already "out there". But any combination is useless, if you don't have the private key for this particular combination. "Create" an account means to start with a private key, and then **find** the combination, the address, which is derived from this specific private key (actually from the public key, which itself is derived from the private key).

The Genesis Block

The next file is `genesis.json`. This is the definition of the first block of the chain, which is called the "genesis" block. Every blockchain starts with an initial genesis block, the #0 block. The real ethereum genesis block can be seen here.

Take a look at the horton genesis block:

```
$ cat chains/horton/genesis.json

{
  "parentHash": "0x0000000000000000000000000000000000000000000000000000000000000000",
  "coinbase": "0xeb9ae85fa8f6d8853afe45799d966dca89edd9aa",
  "extraData": "0x686f727365",
  "config": {
      "daoForkBlock": 0,
      "daoForSupport": true,
      "homesteadBlock": 0
  },
  "timestamp": "0x0",
  "mixhash": "0x0000000000000000000000000000000000000000000000000000000000000000",
  "nonce": "0xdeadbeefdeadbeef",
  "alloc": {
      "0xeb9ae85fa8f6d8853afe45799d966dca89edd9aa":{
        "balance": "1000000000000000000000000000000"
    }
    },
  "gasLimit": "0x47d5cc",
  "difficulty": "0x01"
}
```

The genesis block parent hash is 0, since it's the first block.

The nice thing about having your very own money minting facility, is that you can mint money quite easily! So the genesis block allocates to the default account not less than one billion ether. Think of it as monopoly money: it looks like real money, it behaves like real money, but it will not get you much in the grocery store. However, this local chain Eth is very handy for development and testing.

Running the Local Blockchain

Great. Everything is in place to run your own local blockchain.

Before the first run, you need to initiate this blockchain. Go ahead and init the chain, with the script that populus created:

```
$ chains/horton/./init_chain.sh
```

Geth will init the blockchain:

```
INFO [10-10|07:17:48] Allocated cache and file handles         database=/home/marry/
↪projects/myproject/chains/horton/chain_data/geth/chaindata cache=16 handles=16
INFO [10-10|07:17:48] Writing custom genesis block
INFO [10-10|07:17:48] Successfully wrote genesis state         database=chaindata
↪                                                               hash=ab7daa...b26156
INFO [10-10|07:17:48] Allocated cache and file handles         database=/home/marry/
↪projects/myproject/chains/horton/chain_data/geth/lightchaindata cache=16 handles=16
INFO [10-10|07:17:48] Writing custom genesis block
```

```
INFO [10-10|07:17:48] Successfully wrote genesis state
↪database=lightchaindata
↪      hash=ab7daa...b26156
```

Note: You need to run the init script only once for each new chain

When geth created the blockchain, it added some files, where it stores the blockchain data:

```
chains/
- horton
    - chain_data
    |   - geth
    |   |   - chaindata
    |   |   |   - 000001.log
    |   |   |   - CURRENT
    |   |   |   - LOCK
    |   |   |   - LOG
    |   |   |   - MANIFEST-000000
    |   |   - lightchaindata
    |   |       - 000001.log
    |   |       - CURRENT
    |   |       - LOCK
    |   |       - LOG
    |   |       - MANIFEST-000000
    |   - keystore
    |       - UTC--2017-10-10T14-17-37.895269081Z--
↪62c4b5955c028ab16bfc5cc57e09af6370a270a1
    - genesis.json
    - init_chain.sh
    - password
    - run_chain.sh
```

Finally, you can run your own local blockchain!

```
$ chains/horton/./run_chain.sh
```

And you should see geth starting to actually run the blockchain:

```
INFO [10-10|07:20:45] Starting peer-to-peer node               instance=Geth/v1.6.7-
↪stable-ab5646c5/linux-amd64/go1.8.1
INFO [10-10|07:20:45] Allocated cache and file handles         database=/home/mary/
↪projects/myproject/chains/horton/chain_data/geth/chaindata cache=128 handles=1024
WARN [10-10|07:20:45] Upgrading chain database to use sequential keys
INFO [10-10|07:20:45] Initialised chain configuration          config="{ChainID: <nil>
↪ Homestead: 0 DAO: 0 DAOSupport: false EIP150: <nil> EIP155: <nil> EIP158: <nil>
↪Metropolis: <nil> Engine: unknown}"
INFO [10-10|07:20:45] Disk storage enabled for ethash caches   dir=/home/mary/
↪projects/myproject/chains/horton/chain_data/geth/ethash count=3
INFO [10-10|07:20:45] Disk storage enabled for ethash DAGs     dir=/home/mary/.ethash
↪                                                          count=2
WARN [10-10|07:20:45] Upgrading db log bloom bins
INFO [10-10|07:20:45] Bloom-bin upgrade completed              elapsed=163.975µs
INFO [10-10|07:20:45] Initialising Ethereum protocol           versions="[63 62]"
↪network=1234
INFO [10-10|07:20:45] Loaded most recent local header          number=0
↪hash=ab7daa...b26156 td=1
```

```
INFO [10-10|07:20:45] Loaded most recent local full block      number=0␣
↪hash=ab7daa...b26156 td=1
INFO [10-10|07:20:45] Loaded most recent local fast block      number=0␣
↪hash=ab7daa...b26156 td=1
INFO [10-10|07:20:45] Starting P2P networking
INFO [10-10|07:20:45] HTTP endpoint opened: http://127.0.0.1:8545
INFO [10-10|07:20:45] WebSocket endpoint opened: ws://127.0.0.1:8546
INFO [10-10|07:20:45] Database conversion successful
INFO [10-10|07:20:45] RLPx listener up                         self="enode://
↪dc6e3733c416843a35b829c4edf5452674fccf4d0e9e25d026ae6fe82a06ff600958d870c505eb4dd877e477ffb3831a10
↪discport=0"
INFO [10-10|07:20:45] IPC endpoint opened: /home/mary/projects/myproject/chains/
↪horton/chain_data/geth.ipc
INFO [10-10|07:20:46] Unlocked account                        ␣
↪address=0x62c4b5955c028ab16bfc5cc57e09af6370a270a1
INFO [10-10|07:20:46] Transaction pool price threshold updated price=1800000000
```

Note the IPC (in process communication) endpoint line: `IPC endpoint opened: /home/mary/projects/myproject/chains/horton/chain_data/geth.ipc`. The actual path on your machine should match the project path.

IPC allows connection from the same machine, which is safer.

Where the Blockchain is Actually Running?

The blockchain that runs now does not relate to populus. Populus just created some files, but the chain is an independent geth process which runs on your machine.

You can verify it, using the web3 javascript console. In another terminal, open a console that attaches to this blockchain:

```
$ geth attach /home/mary/projects/myproject/chains/horton/chain_data/geth.ipc
```

Use the actual IPC endpoint path that runs on your machine. You can take a look at `run_chain.sh` to see this path.

The web3.js console looks like this :

```
Welcome to the Geth JavaScript console!

instance: Geth/v1.6.7-stable-ab5646c5/linux-amd64/go1.8.1
coinbase: 0x62c4b5955c028ab16bfc5cc57e09af6370a270a1
at block: 9 (Tue, 10 Oct 2017 07:30:00 PDT)
 datadir: /home/may/projects/myproject/chains/horton/chain_data
 modules: admin:1.0 debug:1.0 eth:1.0 miner:1.0 net:1.0 personal:1.0 rpc:1.0 txpool:1.
↪0 web3:1.0

>
```

Check your account balance:

```
> web3.fromWei(eth.getBalance(eth.coinbase))
1000000000160
>
```

Wow! you already have even more than the original allocation of one billion! These are the rewards for successful mining. Boy, the rich get richer.

Note: Wei is the unit that *getBalance* returns by default, and *fromWei* converts Wei to Ethereum. See the Ethereum units denominations

You can work in the geth console and try other web3.js commands. But as much as we love javascript, if you were missing those brackets and semicolons you would not be here, in the Populus docs, would you?

So the next step is to deploy the Greeter contract with Populus to the horton local chain.

Note: To stop geth, go the terminal where it's running, and type Ctrl+C. If it runs as a daemon, use `kill INT <pid>`, where pid is the geth process id.

Part 3: Deploy to a Local Chain

At part 2 of the tutorial you created a local chain named 'horton'. Now, we will deploy the Greeter contract to this chain.

Project Config

First, we have to add the chain to the project configuration file:

```
$ nano project.json
```

After edit, the project file should look like this:

```
{
  "version":"7",
  "compilation":{
    "contracts_source_dirs": ["./contracts"],
    "import_remappings": []
  },
  "chains": {
    "horton": {
      "chain": {
        "class": "populus.chain.ExternalChain"
      },
      "web3": {
        "provider": {
          "class": "web3.providers.ipc.IPCProvider",
        "settings": {
          "ipc_path":"/home/mary/projects/myproject/chains/horton/chain_data/geth.ipc"
        }
       }
      },
      "contracts": {
        "backends": {
```

```
          "JSONFile": {"$ref": "contracts.backends.JSONFile"},
          "ProjectContracts": {
            "$ref": "contracts.backends.ProjectContracts"
          }
        }
      }
    }
  }
}
```

The `ipc_path` should be the exact ipc_path on your machine. If you are not sure, copy-paste the path from the `run_chain.sh` script in the `chains/horton/` directory.

Note: Populus uses JSON schema configuration files, on purpose. We think that for blockchain development, it's safer to use static, external configuration, than a Python module. For more about JSON based configuration see JSON Schema.

Note the line `{"$ref": "contracts.backends.JSONFile"}`. There is a `$ref`, but the reference key does not exist in the file. This is beacuase the `project.json` config file is *complementary* to the main populus user-scope config file, at `~/.populus/config.json`. The user config holds for all your populus projects, and you can use the `project.json` just for the configuration changes that you need for a specific project. Thus, you can `$ref` the user-config keys in any project configuration file.

Running the Chain

If the horton chain is not running (see part 2), run it. From the project directory, in another terminal, use the script that Populus created:

```
$ chains/horton/./run_chain.sh
```

You should see a ton of things that geth outputs.

Deploy

Finally, deploy the contract:

```
$ deploy --chain horton Greeter --no-wait-for-sync

> Found 1 contract source files
  - contracts/Greeter.sol
> Compiled 1 contracts
  - contracts/Greeter.sol:Greeter
Beginning contract deployment.  Deploying 1 total contracts (1 Specified, 0 because␣
↪of library dependencies).

Greeter

Deploy Transaction Sent:␣
↪0x364d8d4b7e40992bed2ea5f92af833d58ef1b9f3b4171c1f64f8843c2527437d
Waiting for confirmation...
```

After a few seconds the transaction is mined in your local chain:

```
Transaction Mined
=================
Tx Hash      : 0x364d8d4b7e40992bed2ea5f92af833d58ef1b9f3b4171c1f64f8843c2527437d
Address      : 0x1c51ff8a84345f0a5940601b3bd372d8105f71aa
Gas Provided : 465580
Gas Used     : 365579


Verified contract bytecode @ 0x1c51ff8a84345f0a5940601b3bd372d8105f71aa
Deployment Successful.
```

Well done!

Note: We used here `--no-wait-for-sync`, since the account has (a lot of) Eth from the get go, allocated in the genesis block. However, if you work with testnet or mainnet, you must sync at least until the block with the transactions that sent some Eth to the account you are deploying from. Otherwise, your local geth will not know that there is Eth in the account to pay for the gas. Once the chain is synced, you can deploy immidiatly.

1.4 Compiling

Running `$ populus compile` will compile all of the project contracts found in the `./contracts/` directory. The compiled assets are then written to `./build/contracts.json`.

Note: Populus currently only supports compilation of Solidity contracts.

1.4.1 Basic Compilation

Basic usage to compile all of the contracts and libraries in your project can be done as follows.

```
$ populus compile
============ Compiling ==============
> Loading source files from: ./contracts

> Found 1 contract source files
- contracts/Greeter.sol

> Compiled 1 contracts
- Greeter

> Wrote compiled assets to: ./build/contracts.json
```

Outside the project directory use

```
$ populus -p /path/to/my/project/ compile
```

1.4.2 Watching

This command can be used with the flag `--watch/-w` which will automatically recompile your contracts when the source code changes.

```
$ populus compile --watch
============ Compiling ==============
> Loading source files from: ./contracts

> Found 1 contract source files
- contracts/Greeter.sol

> Compiled 1 contracts
- Greeter

> Wrote compiled assets to: ./build/contracts.json
Change detected in: contracts/Greeter.sol
============ Compiling ==============
> Loading source files from: ./contracts

> Found 1 contract source files
- contracts/Greeter.sol

> Compiled 1 contracts
- Greeter

> Wrote compiled assets to: ./build/contracts.json
```

1.4.3 Build Output

Output is serialized as `JSON` and written to `build/contracts.json` relative to the root of your project. It will be a mapping of your contract names to the compiled assets for that contract.


```
{
    "Greeter": {
        "abi": [
            {
                "constant": false,
                "inputs": [
                    {
                        "name": "_greeting",
                        "type": "string"
                    }
                ],
                "name": "setGreeting",
                "outputs": [],
                "payable": false,
                "type": "function"
            },
            {
                "constant": true,
                "inputs": [],
                "name": "greet",
                "outputs": [
                    {
                        "name": "",
                        "type": "string"
                    }
                ],
                "payable": false,
                "type": "function"
```

```
            },
            {
                "constant": true,
                "inputs": [],
                "name": "greeting",
                "outputs": [
                    {
                        "name": "",
                        "type": "string"
                    }
                ],
                "payable": false,
                "type": "function"
            },
            {
                "inputs": [],
                "payable": false,
                "type": "constructor"
            }
        ],
        "bytecode": "0x6060604052....",
        "bytecode_runtime": "0x6060604052....",
        "metadata": {
            "compiler": {
                "version": "0.4.8+commit.60cc1668.Darwin.appleclang"
            },
            "language": "Solidity",
            "output": {
                "abi": [
                    {
                        "constant": false,
                        "inputs": [
                            {
                                "name": "_greeting",
                                "type": "string"
                            }
                        ],
                        "name": "setGreeting",
                        "outputs": [],
                        "payable": false,
                        "type": "function"
                    },
                    {
                        "constant": true,
                        "inputs": [],
                        "name": "greet",
                        "outputs": [
                            {
                                "name": "",
                                "type": "string"
                            }
                        ],
                        "payable": false,
                        "type": "function"
                    },
                    {
                        "constant": true,
                        "inputs": [],
```

```
                        "name": "greeting",
                        "outputs": [
                            {
                                "name": "",
                                "type": "string"
                            }
                        ],
                        "payable": false,
                        "type": "function"
                    },
                    {
                        "inputs": [],
                        "payable": false,
                        "type": "constructor"
                    }
                ],
                "devdoc": {
                    "methods": {}
                },
                "userdoc": {
                    "methods": {}
                }
            },
            "settings": {
                "compilationTarget": {
                    "contracts/Greeter.sol": "Greeter"
                },
                "libraries": {},
                "optimizer": {
                    "enabled": true,
                    "runs": 200
                },
                "remappings": []
            },
            "sources": {
                "contracts/Greeter.sol": {
                    "keccak256":
↪"0xe7900e8d25304f64a90939d1d9f90bb21268c4755140dc396b8b4b5bdd21755a",
                    "urls": [
                        "bzzr://
↪7d6c0ce214a43b81f423edff8b18e18ad7154b7f364316bbd3801930308c1984"
                    ]
                }
            },
            "version": 1
        }
    }
}
```


1.4.4 Configuration

The following configuration options can be set to control aspects of how Populus compiles your project contracts.

- `compilation.contracts_source_dirs`

  Defaults to `[./contracts]`. This sets the paths where populus will search for contract source files.

- `compilation.settings.optimize`

Defaults to `True`. Determines if the optimizer will be enabled during compilation.

1.5 Testing

1.5.1 Introduction

The Populus framework provides some powerful utilities for testing your contracts. Testing in Populus is powered by the python testing framework `py.test`.

By default tests are run against an in-memory ethereum blockchain.

The convention for tests is to place them in the `./tests/` directory in the root of your project. In order for `py.test` to find your tests modules their module name must start with `test_`.

1.5.2 Test Contracts

Populus supports writing contracts that are specifically for testing. These contract filenames should match the glob pattern `Test*.sol` and be located anywhere under your project tests directory `./tests/`.

Running Tests With Pytest

To run the full test suite of your project:

```
$ py.test tests/
```

Or to run a specific test

```
$ py.test tests/test_greeter.py
```

1.5.3 Pytest Fixtures

The test fixtures provided by populus are what make testing easy. In order to use a fixture in your tests all you have to do add an argument with the same name to the signature of your test function.

Project

- `project`

The Project object for your project.

This project object is initialised first, and the rest of the fixtures are derived from it.

```
def test_project_things(project):
    # directory things
    assert project.project_dir == '/path/to/my/project'

    # raw compiled contract access
    assert 'MyContract' in project.compiled_contract_data
```

How populus finds the project of the project fixture

If no other argument is provided, populus defaults to using the current tests directory. This is true if you run py.test from within the project's directory, or with a positional argument to this project's directory, e.g. `$ py.test /path/to/my/project/`.

If the tests are in a different directory, e.g. `$ py.tests /path/to/tests/`, you will have to provide the tested project:

1. With command line argument: `$ py.test /path/to/tests/ --populus-project /path/to/my/project/`
2. Or, in a pytest.ini file, with the following entry: `populus_project=/path/to/my/project/`
3. Or with an environment variable: `PYTEST_POPULUS_PROJECT`. E.g., `PYTEST_POPULUS_PROJECT=/path/to/my/project/ py.test /path/to/tests/`

The precedence order for these different methods of setting the project directory is:

1. command line
2. pytest.ini
3. environment variable

Chain

- `chain`

A running `'tester'` test chain.

```
def test_greeter(chain):
    greeter, _ = chain.provider.get_or_deploy_contract('Greeter')

    assert greeter.call().greet() == "Hello"

def test_deploying_greeter(chain):
    GreeterFactory = chain.provider.get_contract_factory('Greeter')
    deploy_txn_hash = GreeterFactory.deploy()
    ...
```

Registrar

- `registrar`

Convenience fixture for the `chain.registrar` property.

Provider

- `provider`

Convenience fixture for the `chain.provider` property.

Web3

- `web3`

Convenience fixture for the `chain.provider` property. A Web3.py instance configured to connect to `chain` fixture.

```
def test_account_balance(web3, chain):
    initial_balance = web3.eth.getBalance(web3.eth.coinbase)
    wallet = chain.get_contract('Wallet')

    withdraw_txn_hash = wallet.transact().withdraw(12345)
    withdraw_txn_receipt = chain.wait.for_receipt(withdraw_txn_hash)
    after_balance = web3.eth.getBalance(web3.eth.coinbase)

    assert after_balance - initial_balance == 1234
```

Base Contract Factories

- `base_contract_factories`

The contract factory classes for your project. These will all be associated with the Web3 instance from the `web3` fixture.

```
def test_wallet_deployment(web3, base_contract_factories):
    WalletFactory = base_contract_factories.Wallet

    deploy_txn_hash = WalletFactory.deploy()
```

Note: For contracts that have library dependencies, you should use the `Chain.get_contract_factory(...)` api. The contract factories from the `base_contract_factories` fixture will not be returned with linked bytecode. The ones from `Chain.get_contract_factory()` are returned fully linked.

Accounts

- `accounts`

The `web3.eth.accounts` property off of the `web3` fixture

```
def test_accounts(web3, accounts):
    assert web3.eth.coinbase == accounts[0]
```

1.5.4 Custom Fixtures

The built in fixtures for accessing contracts are useful for simple contracts, but this is often not sufficient for more complex contracts. In these cases you can create you own fixtures to build on top of the ones provided by Populus.

One common case is a contract that needs to be given constructor arguments. Lets make a fixture for a token contract that requires a constructor argument to set the initial supply.

```
import pytest

@pytest.fixture()
```

```
def token_contract(chain):
    TokenFactory = chain.get_contract_factory('Token')
    deploy_txn_hash = TokenFactory.deploy(arguments=[
        1e18,  # initial token supply
    )
    contract_address = chain.wait.for_contract_address(deploy_txn_hash)
    return TokenFactory(address=contract_address)
```

Now, you can use this fixture in your tests the same way you use the built-in populus fixtures.

```
def test_initial_supply(token_contract):
    assert token_contract.call().totalSupply() == 1e18
```

1.6 Deploy

1.6.1 Introduction

Populus provides a command line interface for contract deployments which is suitable for simple contract deployments which do not involve constructor arguments as well as APIs for performing more complex deployments using python.

1.6.2 Deploying A Contract with the Command Line

Deployment is handled through the `$ populus deploy` command. All of the following are handled automatically.

1. Selection of which chain should be deployed to.
2. Running the given chain.
3. Compilation of project contracts.
4. Derivation of library dependencies.
5. Library linking.
6. Individual contract deployment.

Lets deploy a simple Wallet contract. First we'll need a contract in our project `./contracts` directory.

```
// ./contracts/Wallet.sol
contract Wallet {
        mapping (address => uint) public balanceOf;

        function deposit() {
                balanceOf[msg.sender] += 1;
        }

        function withdraw(uint value) {
```

```
                if (balanceOf[msg.sender] < value) throw;
                balanceOf[msg.sender] -= value;
                if (!msg.sender.call.value(value)()) throw;
        }
}
```

We can deploy this contract to a local test chain like this:

```
    $ populus deploy Wallet -c local_a
    Beginning contract deployment.  Deploying 1 total contracts (1 Specified, 0␣
↪because of library dependencies).

    Wallet
    Deploying Wallet
    Deploy Transaction Sent:␣
↪0x29e90f07314db495989f03ca931088e1feb7fb0fc13286c1724f11b2d6b239e7
    Waiting for confirmation...

    Transaction Mined
    =================
    Tx Hash      : 0x29e90f07314db495989f03ca931088e1feb7fb0fc13286c1724f11b2d6b239e7
    Address      : 0xb6fac5cb309da4d984bb6145078104355ece96ca
    Gas Provided : 267699
    Gas Used     : 167699


    Verifying deployed bytecode...
    Verified contract bytecode @ 0xb6fac5cb309da4d984bb6145078104355ece96ca matches␣
↪expected runtime bytecode
Deployment Successful.
```

Above you can see the output for a basic deployment.

If your are outside the project directory, use:

```
$ populus -p /path/to/my/project deploy Wallet -c local_a
```

1.6.3 Programmatically deploy a contract

You can also deploy contracts using a Python script. This is a suitable method if your contracts take constructor arguments or need more complex initialization calls.

Example (`deploy_testnet.py`):

```
"""Deploy Edgeless token and smart contract in testnet.

A simple Python script to deploy contracts and then do a smoke test for them.
"""
from populus import Project
from populus.utils.wait import wait_for_transaction_receipt
from web3 import Web3


def check_succesful_tx(web3: Web3, txid: str, timeout=180) -> dict:
    """See if transaction went through (Solidity code did not throw).

    :return: Transaction receipt
```

```
    """

    # http://ethereum.stackexchange.com/q/6007/620
    receipt = wait_for_transaction_receipt(web3, txid, timeout=timeout)
    txinfo = web3.eth.getTransaction(txid)

    # EVM has only one error mode and it's consume all gas
    assert txinfo["gas"] != receipt["gasUsed"]
    return receipt


def main():

    project = Project()

    # This is configured in populus.json
    # We are working on a testnet
    chain_name = "ropsten"
    print("Make sure {} chain is running, you can connect to it, or you'll get timeout
↪".format(chain_name))

    with project.get_chain(chain_name) as chain:

        # Load Populus contract proxy classes
        Crowdsale = chain.get_contract_factory('Crowdsale')
        Token = chain.get_contract_factory('EdgelessToken')

        web3 = chain.web3
        print("Web3 provider is", web3.currentProvider)

        # The address who will be the owner of the contracts
        beneficiary = web3.eth.coinbase
        assert beneficiary, "Make sure your node has coinbase account created"

        # Random address on Ropsten testnet
        multisig_address = "0x83917f644df1319a6ae792bb244333332e65fff8"

        # Deploy crowdsale, open since 1970
        txhash = Crowdsale.deploy(transaction={"from": beneficiary},␣
↪args=[beneficiary, multisig_address, 1])
        print("Deploying crowdsale, tx hash is", txhash)
        receipt = check_succesful_tx(web3, txhash)
        crowdsale_address = receipt["contractAddress"]
        print("Crowdsale contract address is", crowdsale_address)

        # Deploy token
        txhash = Token.deploy(transaction={"from": beneficiary}, args=[beneficiary])
        print("Deploying token, tx hash is", txhash)
        receipt = check_succesful_tx(web3, txhash)
        token_address = receipt["contractAddress"]
        print("Token contract address is", token_address)

        # Make contracts aware of each other
        print("Initializing contracts")
        crowdsale = Crowdsale(address=crowdsale_address)
        token = Token(address=token_address)
        txhash = crowdsale.transact({"from": beneficiary}).setToken(token_address)
        check_succesful_tx(web3, txhash)
```

```
        # Do some contract reads to see everything looks ok
        print("Token total supply is", token.call().totalSupply())
        print("Crowdsale max goal is", crowdsale.call().maxGoal())

        print("All done! Enjoy your decentralized future.")


if __name__ == "__main__":
    main()
```

See full source code repository example.

1.7 Project

1.7.1 Introduction

class `populus.project.`**`BaseChain`**

The *`Project`* class is the primary entry point to all aspects of your populus project.

1.7.2 Basic Usage

- `Project(config_file_path=None)`

When instantaited with no arguments, the project will look for a `populus.json` file found in the current working directory and load that if found.

```
from populus.project import Project
# loads local `populus.json` file (if present)
project = Project()

# loads the specified config file
other_project = Project('/path/to/other/populus.json')
```

The project object is the entry point for almost everything that populus can do.

```
>>> project.project_dir
'/path/to/your-project`
>>> project.contracts_dir
'./contracts'
>>> project.config
{....}  # Your project configuration.
>>> project.compiled_contract_data
```

```
{
  'Greeter': {
    'code': '0x...',
    'code_runtime': '0x...',
    'abi': [...],
    ...
  },
  ...
}
>>> with p.get_chain('temp') as chain:
...     print(chain.web3.eth.coinbase)
...
0x4949dce962e182bc148448efa93e73c6ba163f03
```

1.7.3 Configuration

`Project.`**`config`**
Returns the current project configuration.

`Project.`**`load_config`**`()`
Loads the project configuration from disk, populating *`Project.config`*.

`Project.`**`write_config`**`()`
Writes the current project configuration from *`Project.config`* to disk.

1.7.4 Chains

`Project.get_chain(chain_name, chain_config=None)``
Returns a `populus.chain.Chain` instance. You may provide `chain_config` in which case the chain will be configured using the provided configuration rather than the declared configuration for this chain from your configuration file.

The returned `Chain` instance can be used as a context manager.

1.8 Configuration

1.8.1 Introduction

Populus is designed to be highly configurable through the configuration files.

By default, populus will load the configuration from two files: the user-scope main config file at `~/.populus/config.json`, and the project-scope config file, at the project directory, `project.json`.

Both files share the same JSON schema. You should use the `project.json` file for local changes that apply to specific project, and the user-scope file for the environment configs, which apply to all your projects.

When a configuration key exists in both the user-config and the project-config, the project-config overrides the user-config. However, progrmatically you have access to both configs and can decide in runtime to choose otherwise.

The `$ populus init` command writes a minimal `project.json` default file to the project directory.

Note: The `project.json` file is required, and all the populus commands require a directory with a project config file.

A Note for Django users

If you are used to django's `settings.py` file, populus is quite different. The configuration is saved in JSON files, on purpuse. While saving the configuration in a Python module is convinient, and often looks nicer, there is a caveat: a

python module is after all a programmble, running code. With an Ethereum development platform, that deals directly with money, we think it's safer to put the configurations in static, non programmble, and external files.

The option to change the configuration dynamically is still available in run time, using the `project.config` propery. But otherwise, Populus configuration comes from static JSON files. What you see is what you get, no surprises.

What You Can Configure

This config file controls many aspects of populus that are configurable. Currently the config file controls the following things.

- Project root directory
- Contract source file location
- Compiler settings
- Available chains and how web3 connects to them.

Compiler Configuration

Each complication backend takes settings that are passed down to Solidity compiler Input Description (JSON) and command line.

Here is an example for the compilation backend settings when using contract source files in folders outside of the Populus default `./contracts` folder.

```
{
  "compilation": {
    "contracts_source_dirs": [
      "./contracts",
      "/path-to-your-external-solidity-files",
    ],
    "backend": {
      "class": "populus.compilation.backends.SolcCombinedJSONBackend",
      "settings": {
          "stdin": {
            "optimizer": {
              "enabled": true,
              "runs": 500
            },
            "outputSelection": {
              "*": {
                "*": [
                  "abi",
                  "metadata",
                  "evm.bytecode",
                  "evm.deployedBytecode"
                ]
              }
            }
          },
          "command_line_options": {
            "allow_paths": "/path-to-your-external-solidity-files"
          }
        }
      }
    }
```

```
  }
}
```

`backend.settings` has two keys

- `stdin` is Solidity Input Description as JSON
- `command_line_options` are passed to Solidity compiler command line, as given keyword arguments to *py-solc* package's *solc.wrapper.solc_wrapper <https://github.com/ethereum/py-solc/blob/3a6de359dc31375df46418e6ffd7f45ab9567287/solc/wrapper.py#L20>_*

Contract Source Directory

The directory that project source files can be found in.

- key: `compilation.contracts_source_dirs`
- value: List of filesystem paths
- default: `['./contracts']`

Compiler Backend

Set which compiler backend should be used

- key: `compilation.backend.class`
- value: Dot separated python path
- default: `populus.compilation.backends.SolcStandardJSONBackend`

Settings for the compiler backend

- key: `compilation.backend.settings`
- value: Object of configuration parameters for the compiler backend.
- default: `{"optimize": true, "output_values": ["abi", "bin", "bin-runtime", "devdoc", "metadata", "userdoc"]}`

Configuring compiler for extra

Set solc import path remappings. This is especially useful if you want to use libraries like OpenZeppelin with your project. Then you can directly import Zeppelin contracts like `import "zeppelin/contracts/token/TransferableToken.sol";`.

- key: `compilation.import_remappings`
- value: Array of strings
- default: `[]`
- example: `["zeppelin=zeppelin"]` assuming that the root directory for the Zeppelin contracts is `./zeppelin` in the root of your project.

Chains

The `chains` key within the configuration file declares what chains populus has access to, and how to connect to them. Populus comes pre-configured with the following chains.

- `'mainnet'`: Connects to the public ethereum mainnet via `geth`.
- `'ropsten'`: Connects to the public ethereum ropsten testnet via `geth`.
- `'tester'`: Uses an ephemeral in-memory chain backed by pyethereum.
- `'testrpc'`: Uses an ephemeral in-memory chain backed by pyethereum.
- `'temp'`: Local private chain whos data directory is removed when the chain is shutdown. Runs via `geth`.

```
{
  "chains": {
    "my-chain": {
      ... // The chain settings.
    }
  }
}
```

Individual Chain Settings

Each key and value in the `chains` portion of the configuration corresponds to the name of the chain and the settings for that chain. Each chain has two primary sections, `web3` and `chain` configuration settings.

```
{
  "chains": {
    "my-chain": {
      "chain": {
        "class": "populus.chain.LocalGethChain"
      },
      "web3": {
        "provider": {
          "class": "web3.providers.ipc.IPCProvider"
        }
      }
    }
  }
}
```

The above chain configuration sets up a new local private chain within your project. The chain above would set it's data directory to `<project-dir>/chains/my-chain/`.

To simplify configuration of chains you can use the `ChainConfig` object.

```
>>> from populus.config import ChainConfig
>>> chain_config = ChainConfig()
>>> chain_config.set_chain_class('local')
>>> chain_config['web3'] = web3_config  # see below for the Web3Config object
>>> project.config['chains.my-chain'] = chain_config
```

The `set_chain_class()` method can take any of the following values.

- **These strings**
  - `chain_config.set_chain_class('local') => 'populus.chain.LocalGethChain'`

  - `chain_config.set_chain_class('external') => 'populus.chain.ExternalChain'`
  - `chain_config.set_chain_class('tester') => 'populus.chain.TesterChain'`
  - `chain_config.set_chain_class('testrpc') => 'populus.chain.TestRPCChain'`
  - `chain_config.set_chain_class('temp') => 'populus.chain.TemporaryGethChain'`
  - `chain_config.set_chain_class('mainnet') => 'populus.chain.MainnetChain'`
  - `chain_config.set_chain_class('testnet') => 'populus.chain.TestnetChain'`
  - `chain_config.set_chain_class('ropsten') => 'populus.chain.TestnetChain'`

- **Full python paths to the desired chain class.**
  - `chain_config.set_chain_class('populus.chain.LocalGethChain') => 'populus.chain.LocalGethChain'`
  - `chain_config.set_chain_class('populus.chain.ExternalChain') => 'populus.chain.ExternalChain'`
  - `...`
- **The actual chain class.**
  - `chain_config.set_chain_class(LocalGethChain) => 'populus.chain.LocalGethChain'`
  - `chain_config.set_chain_class(ExternalChain) => 'populus.chain.ExternalChain'`
  - `...`

Chain Class Settings

Determines which chain class will be used for the chain.

- key: `chains.<chain-name>.chain.class`
- value: Dot separated python path to the chain class that should be used.
- required: Yes

Available options are:

- `populus.chain.ExternalChain`

  A chain that populus does not manage or run. This is the correct class to use when connecting to a node that is already running.

- `populus.chain.TestRPCChain`

  An ephemeral chain that uses the python `eth-testrpc` package to run an in-memory ethereum blockchain. This chain will spin up an HTTP based RPC server.

- `populus.chain.TesterChain`

An ephemeral chain that uses the python `eth-testrpc` package to run an in-memory ethereum blockchain. This chain **must** be used in conjunction with a web configuration using the provider `EthereumTesterProvider`.

- `populus.chain.LocalGethChain`

  A geth backed chain which will setup it's own data directory in the `./chains` directory in the root of your project.

- `populus.chain.TemporaryGethChain`

  An ephemeral chain backed by `geth` which uses a temporary directory as the data directory which is removed when the chain is shutdown.

- `populus.chain.TestnetChain`

  A `geth` backed chain which connects to the public Ropsten test network.

- `populus.chain.MainnetChain`

  A `geth` backed chain which connects to the main public network.

Web3

Configuration for the Web3 instance that will be used with this chain. See *Web3 Configuration* for more details.

- key: `chains.<chain-name>.web3`
- value: Web3 Configuration
- required: Yes

1.8.2 Custom Chains Using the ExternalChain Class

You can define your own custom chains. Custom chains should use the `ExternalChain` class, which lets you access a Web3 provider. Web3 is the actual layer that connects to the running geth process, and let Populus interact with the blockchain.

Note: If you are familiar with web development, then you can think of Web3 as the underlying WSGI. In web development WSGI hooks to Apache or Nginx, here we have Web3 that hooks to geth.

The minimum configuration that Web3 requires are *either*:

- `IPCProvider`: connects to geth via IPC, by the configured `ipc_path`
- `HTTPProvider`: connects via http or https to geth's rpc, by the configured `endpoint_uri`

Here are two examples of a custom `ExternalChain` configuration.

IPC

```
"chains": {
  "horton": {
    "chain": {
      "class": "populus.chain.ExternalChain"
    },
    "web3": {
      "provider": {
        "class": "web3.providers.ipc.IPCProvider",
```

```
        "settings": {
          "ipc_path": "./chains/horton/geth.ipc"
        }
      }
    },
    ...
  }
}
```

HTTP

```
"chains": {
  "local_chain": {
    "chain": {
      "class": "populus.chain.ExternalChain"
    },
    "web3": {
      "provider": {
        "class": "web3.providers.rpc.HTTPProvider",
        "settings": {
          "endpoint_uri": "https://127.0.0.1:8545"
        }
      }
    },
    ...
  }
}
```

The important thing to remeber is that Populus will **not** run geth for you. You will have to run geth, and then Populus will use the chain configuration to connect to this **already running** process via Web3. If you created a local chain with the `$ populus chain new` command, Populus will create an executable that you can use to run the chain, see *Running the Local Blockchain*

In the next Populus version, all the chains will be configured as `ExternalChain`

For more details on Web3, see the Web3 documentation .

1.8.3 Web3 Configuration

Configuration for setting up a Web3 instance.

```
{
  "provider": {
    "class": "web3.providers.ipc.IPCProvider",
    "settings": {
      "ipc_path": "/path/to/geth.ipc"
    }
  }
  "eth": {
    "default_account": "0xd3cda913deb6f67967b99d67acdfa1712c293601",
  }
}
```

In order to simplify configuring Web3 instances you can use the `Web3Config` class.

```
>>> from populus.config import Web3Config
>>> web3_config = Web3Config()
```

```
>>> web3_config.set_provider('ipc')
>>> web3_config.provider_kwargs['ipc_path'] = '/path/to/geth.ipc'
>>> web3.config.default_account = '0x0000000000000000000000000000000000000001'
>>> project.config['chains.my-chain.web3'] = web3_config
>>> project.write_config()  # optionally persist the configuration to disk
```

Provider Class

Specifies the import path for the provider class that should be used.

- key: `provider.class`
- value: Dot separated python path
- required: Yes

Provider Settings

Specifies the `**kwargs` that should be used when instantiating the provider.

- key: `provider.settings`
- value: Key/Value mapping

Default Account

If present the `web3.eth.defaultAccount` will be populated with this address.

- key: `eth.default_account`
- value: Ethereum Address

1.8.4 Configuration API

The project configuration can be accessed as a property on the `Project` object via `project.config`. This object is a dictionary-like object with some added convenience APIs.

Project configuration is represented as a nested key/value mapping.

Getting and Setting

The `project.config` object exposes the following API for getting and setting configuration values. Supposing that the project configuration file contained the following data.

```
{
  'a': {
    'b': {
      'c': 'd',
      'e': 'f'
    }
  },
  'g': {
    'h': {
      'i': 'j',
```

```
      'k': 'l'
    }
  }
}
```

The config object supports retrieval of values in much the same manner as a dictionary. For convenience, you can also access *deep* nested values using a single key which is dot-separated combination of all keys.

```
>>> project.config.get('a')
{
  'b': {
    'c': 'd',
    'e': 'f'
  }
}
>>> project.config['a']
{
  'b': {
    'c': 'd',
    'e': 'f'
  }
}
>>> project.config.get('a.b')
{
  'c': 'd',
  'e': 'f'
}
>>> project.config['a.b']
{
  'c': 'd',
  'e': 'f'
}
>>> project.config.get('a.b.c')
'd'
>>> project.config['a.b.c']
'd'
>>> project.config.get('a.b.x')
None
>>> project.config['a.b.x']
KeyError: 'x'
>>> project.config.get('a.b.x', 'some-default')
'some-default'
```

The config object also supports setting of values in the same manner.

```
>>> project.config['m'] = 'n'
>>> project.config
{
  'a': {
    'b': {
      'c': 'd',
      'e': 'f'
    }
  },
  'g': {
    'h': {
      'i': 'j',
      'k': 'l'
```

```
    }
  },
  'm': 'n'
}
>>> project.config['o.p'] = 'q'
>>> project.config
{
  'a': {
    'b': {
      'c': 'd',
      'e': 'f'
    }
  },
  'g': {
    'h': {
      'i': 'j',
      'k': 'l'
    }
  },
  'm': 'n'
  'o': {
    'p': 'q'
  }
}
```

Config objects support existence queries as well.

```
>>> 'a' in project.config
True
>>> 'a.b' in project.config
True
>>> 'a.b.c' in project.config
True
>>> 'a.b.x' in project.config
False
```

Config References

Sometimes it is useful to be able to re-use some configuration in multiple locations in your configuration file. This is where references can be useful. To reference another part of your configuration use an object with a single key of `$ref`. The value should be the full key path that should be used in place of the reference object.

```
{
  'a': {
    '$ref': 'b.c'
  }
  'b': {
    'c': 'd'
  }
}
```

In the above, the key `a` is a reference to the value found under key `b.c`

```
>>> project.config['a']
['d']
```

```
>>> project.config.get('a')
['d']
```

1.8.5 Defaults

Populus ships with many defaults which can be overridden as you see fit.

Built-in defaults

Populus ships with the following *default* configuration.

```
{
  "version":"7",
  "compilation":{
    "contracts_source_dirs": ["./contracts"],
    "import_remappings": []
  }
}
```

It is recommended to use the `$ populus init` command to populate this file as it contains useful defaults.

Pre-Configured Web3 Connections

The following pre-configured configurations are available. To use one of the configurations on a chain it should be referenced like this:

```
{
  "chains": {
    "my-custom-chain": {
        "web3": {"$ref": "web3.GethIPC"}
    }
  }
}
```

GethIPC

Web3 connection which will connect to geth using an IPC socket.

- key: `web3.GethIPC`

InfuraMainnet

Web3 connection which will connect to the mainnet ethereum network via Infura.

- key: `web3.InfuraMainnet`

InfuraRopsten

Web3 connection which will connect to the ropsten ethereum network via Infura.

- key: `web3.InfuraRopsten`

TestRPC

Web3 connection which will use the `TestRPCProvider`.

- key: `web3.TestRPC`

Tester

Web3 connection which will use the `EthereumTesterProvider`.

- key: `web3.Tester`

1.8.6 Command Line Interface

You can manage your configuration using the command line with the `$ populus config` command.

```
$ populus config
Usage: populus config [OPTIONS] COMMAND [ARGS]...

  Manage and run ethereum blockchains.

Options:
  -h, --help  Show this message and exit.

Commands:
  delete  Deletes the provided key/value pairs from the...
  get     Gets the provided key/value pairs from the...
  list    Prints the project configuration out to the...
  set     Sets the provided key/value pairs in the...
```

To interact with nested keys simply separate them with a `.`.

```
$ populus config list
some.nested.key_a: the_value_a
some.nested.key_b: the_value_b
$ populus config set some.nested.key_c:the_value_c
$ populus config list
some.nested.key_a: the_value_a
some.nested.key_b: the_value_b
some.nested.key_c: the_value_c
$ populus config get some.nested.key_a
some.nested.key_a: the_value_a
$ populus config delete some.nested.key_a
some.nested.key_a: the_value_a (deleted)
```

1.9 Chains

Chains are how populus interacts with the Ethereum blockchain.

1.9.1 Introduction to Chains

Introduction

Populus has the ability to run and/or connect to a variety of blockchains for you, both programatically and from the command line.

Transient Chains

Populus can run two types of transient chains.

- `tester`

  A test EVM backed blockchain.

- `testrpc`

  Runs the `eth-testrpc` chain which implements the full JSON-RPC interface backed by a test EVM.

- `temp`

  Runs a blockchain backed by the go-ethereum `geth` client. This chain will use a temporary directory for it's chain data which will be cleaned up and removed when the chain shuts down.

Local Chains

Local chains can be setup within your `populus.json` file. Each local chain stores its chain data in the `populus.Project.blockchains_dir` and persists it's data between runs.

Local chains are backed by the go-ethereum `geth` client.

Public Chains

Populus can run both the main and ropsten public chains.

- `mainnet`

  With `$ populus chain run mainnet` populus will run the the go-ethereum client for you connected to the main public ethereum network.

- `ropsten`

  With `$ populus chain run ropsten` populus will run the the go-ethereum client for you connected to the ropsten testnet public ethereum network.

Running from the command line

The `$ populus chain` command handles running chains from the command line.

```
$ populus chain
Usage: populus chain [OPTIONS] COMMAND [ARGS]...

  Manage and run ethereum blockchains.

Options:
  -h, --help  Show this message and exit.

Commands:
  reset  Reset a chain removing all chain data and...
  run    Run the named chain.
```

Running programatically from code

The `populus.Project.get_chain(chain_name, chain_config=None)` method returns a `populus.chain.Chain` instance that can be used within your code to run any populus chain. Also read up on the Web3.py library, which offers additional functions to communicate with an Ethereum blockchain.

Lets look at a basic example of using the `temp` chain.

```
>>> from populus import Project
>>> project = Project()
>>> with project.get_chain('temp') as chain:
...     print('coinbase:', chain.web3.eth.coinbase)
...
...
coinbase: 0x16e11a86ca5cc6e3e819efee610aa77d78d6e075
>>>
>>> with project.get_chain('temp') as chain:
...     print('coinbase:', chain.web3.eth.coinbase)
...
...
coinbase: 0x64e49c86c5ad1dd047614736a290315d415ef28e
```

You can see that each time a `temp` chain is instantiated it creates a new data directory and generates new keys.

The `testrpc` chain operates in a similar manner in that each time you run the chain the EVM data is fully reset. The benefit of the `testrpc` server is that it starts quicker, and has mechanisms for manually resetting the chain.

Here is an example of running the `tester` blockchain.

```
>>> from populus import Project
>>> project = Project()
>>> with project.get_chain('tester') as chain:
...     print('coinbase:', chain.web3.eth.coinbase)
...     print('blockNumber:', chain.web3.eth.blockNumber)
...     chain.mine()
...     print('blockNumber:', chain.web3.eth.blockNumber)
...     snapshot_id = chain.snapshot()
...     print('Snapshot:', snapshot_id)
...     chain.mine()
...     chain.mine()
...     print('blockNumber:', chain.web3.eth.blockNumber)
...     chain.revert(snapshot_id)
```

```
...     print('blockNumber:', chain.web3.eth.blockNumber)
...
coinbase: 0x82a978b3f5962a5b0957d9ee9eef472ee55b42f1
blockNumber: 1
blockNumber: 2
Snapshot: 0
blockNumber: 4
blockNumber: 2
```

Note: The `testrpc` chain can be run in the same manner.

class `populus.chain.base.`**`BaseChain`**

1.9.2 Accessing Contracts

The *`BaseChain`* object is the entry point for the Provider and Registrar APIs which collectively give access to your project contracts and related information.

- The Provider API gives access to both the raw compiler output, the contract factories and the deployed instances of your contracts. This api can be accessed from the *`BaseChain.provider`* property.
- The Registrar API records the addresses of deployed contract instances for later retrieval. This api can be accessed from the *`BaseChain.registrar`* property.

Getting the raw compiled data

To retrieve the contract data for a specific contract you will use the `BaseChain.provider.get_base_contract_factory()` method. Supposing that your project contained a contract named "Math" you could retrieve the contract data using the following code.

```
>>> chain.provider.get_base_contract_factory('Math')
{
  'abi': [...],
  'bytecode': '0x...',
  'bytecode_runtime': '0x...',
  'metadata': {...},
}
```

You may also want to retrieve all of the contract data for your project. This can be done with the `BaseChain.provider.get_all_contract_data()` method.

```
>>> chain.provider.get_all_contract_data()
{
  'Math': {'abi': [...], ...},
  'MyOtherContract': {'abi': [...], ...},
}
```

Getting contract factories

The `BaseChain.provider.get_contract_factory()` method gives you access to the contract factory classes for your contracts.

```
>>> Math = chain.provider.get_contract_factory('Math')
>>> Math.abi
[...]
>>> Math.bytecode
"0x..."
>>> Math.bytecode_runtime
"0x..."
```

Contract factories returned by this method will be returned with their underlying bytecode linked against the appropriate library addresses. In the event that one of the underlying dependencies is not available a `NoKnownAddress` exception will be raised.

In some cases you may want the contract factory class without worrying about whether the underlying bytecode linking. Such contract factories are referred to as *"base"* contract factories and can be retrieved using the `BaseChain.provider.get_base_contract_factory()` method.

```
>>> Math = chain.provider.get_base_contract_factory('Math')
>>> Math.abi
[...]
>>> Math.bytecode
"0x..."  # <-- may contain unlinked bytecode.
>>> Math.bytecode_runtime
"0x..."  # <-- may contain unlinked bytecode.
```

Registering contract addresses

When you deploy an instance of a contract populus stores the contract address using the registry API. This is an API that you should rarely need to interact with directly as populus does the registration of new addresses automatically. To set the address for a contract manually you would use the `BaseChain.registrar.set_contract_address()` method.

```
>>> chain.registrar.set_contract_address('Math', '0x...')
```

Retrieving contract addresses

You can use the `BaseChain.registrar.get_contract_addresses()` method to retrieve all known addresses for a given contract. This method will return an interable of addresses or throw a *~populus.contracts.exceptions.NoKnownAddress* exception.

```
>>> chain.registrar.get_contract_address('Math')
['0x123abc....']
```

Retrieving contracts

Populus provides the following APIs for retrieving instances of your deployed contracts.

- `BaseChain.provider.get_contract()`
- `BaseChain.provider.deploy_contract()`
- `BaseChain.provider.get_or_deploy_contract()`

The `BaseChain.provider.get_contract()` function returns an instance of the requested contract.

```
>>> math = chain.provider.get_contract('Math')
>>> math.address
'0x123abc....'
```

The `BaseChain.provider.deploy_contract()` function will deploy a new instance of the requested contract and return a two-tuple of the contract instance and the transaction hash that it was deployed with.

```
>>> math, deploy_txn_hash = chain.provider.deploy_contract('Math')
>>> math.address
'0x123abc....'  # 20 byte hex encoded address
>>> deploy_txn_hash
'0xabcdef...'  # 32 byte hex encoded transaction hash
```

The `BaseChain.provider.get_or_deploy_contract()` function is primarily for testing purposes. If the contract is already available this method will return a two tuple of the contract instance and `None`. If the contract is not available it will be deployed using the provided deploy transaction and arguments, returning a two tuple of the contract instance and the deploy transaction hash.

```
>>> math, deploy_txn_hash = chain.provider.get_or_deploy_contract('Math')
>>> math.address
'0x123abc....'  # 20 byte hex encoded address
>>> deploy_txn_hash
'0xabcdef...'  # 32 byte hex encoded transaction hash
>>> chain.provider.get_or_deploy_contract('Math')
(<Math at 0x123abc>, None)
```

Checking availability of contracts

Sometimes it may be useful to query whether a certain contract or its dependencies are available. This can be done with the following APIs.

- `BaseChain.provider.are_contract_dependencies_available()`
- `BaseChain.provider.is_contract_available()`

The `BaseChain.provider.are_contract_dependencies_available()` method returns `True` if all of the necessary dependencies for the provided contract are avialable. This check includes checks that the bytecode for all dependencies matched the expected compiled bytecode.

The `BaseChain.provider.is_contract_available()` method returns `True` if all dependencies for the requested contract are available **and** there is a known address for the contract **and** the bytecode at the address matches the expected bytecode for the contract.

1.9.3 Wait API

class `populus.wait.`**`Wait`**(*web3*, *timeout=empty*, *poll_interval=empty*)

Each chain object exposes the following API through a property *Chain.wait*.

- The `timeout` parameter sets the default number of seconds that each method will block before raising a `Timeout` exception.
- The `poll_interval` determines how long it should wait between polling. If `poll_interval == None` then a random value between 0 and 1 second will be used for the polling interval.

`Wait.`**`for_contract_address`**(*txn_hash*, *timeout=120*, *poll_interval=None*)
Blocks for up to `timeout` seconds returning the contract address from the transaction receipt for the given `txn_hash`.

`Wait.`**`for_receipt`**(*txn_hash*, *timeout=120*, *poll_interval=None*)
Blocks for up to `timeout` seconds returning the transaction receipt for the given `txn_hash`.

`Wait.`**`for_block`**(*block_number=1*, *timeout=120*, *poll_interval=None*)
Blocks for up to `timeout` seconds waiting until the highest block on the current chain is at least `block_number`.

`Wait.`**`for_unlock`**(*account=web3.eth.coinbase*, *timeout=120*, *poll_interval=None*)
Blocks for up to `timeout` seconds waiting until the account specified by `account` is unlocked. If `account` is not provided, `web3.eth.coinbase` will be used.

`Wait.`**`for_peers`**(*peer_count=1*, *timeout=120*, *poll_interval=None*)
Blocks for up to `timeout` seconds waiting for the client to have at least `peer_count` peer connections.

`Wait.`**`for_syncing`**(*timeout=120*, *poll_interval=None*)
Blocks for up to `timeout` seconds waiting the chain to begin syncing.

1.9.4 Chain API

class `populus.chain.base.`**`BaseChain`**
All chain objects inherit from the *`populus.chain.base.BaseChain`* base class and expose the following API.

`BaseChain.`**`web3`**
Accessor for the `Web3` instance that this chain is configured to use.

`BaseChain.`**`wait`**
Accessor for the *Wait API*.

`BaseChain.`**`registrar`**
Accessor for Registrar API

`BaseChain.`**`provider`**
Accessor for the Provider API

1.10 Development Cycle

Full contract development cycle in Python, with Populus and Web3.py.

1.10.1 Contents

Introduction

Background

The purpose of this tutorial is to go one step beyond the common "Hello World" Greeter example, to the entire developement cycle of an Ethereum smart contract, using Python. We will try to unpack the confusing issues, clear up the mystery, and even have some fun when you dive into blockchain and contracts developemnt.

The tools we will use are Populus and Web3.py

Note: Web3 in general is the client side API that let you intreacts with the blockchain. Populus is a development framework, that is built on top of Web3. If you are into javascript, you can use the Truffle javascript framework, and Web3.js which ships with the `geth` client. If you prefer Python, then Populus and Web3.py are your friends.

We assume that your read 1-2 intros about Ethereum and the blockchain, and know Python.

You don't need the complex math of the elliptic curves, but to get a grasp of the basic concepts, and the basic idea: A system that prevents bad behaviour not by moral rules, but by incentives. Incentinve that make honest behaviour *more* profitable (let this bold concept sink in for a moment).

Development Steps

We will take a walk through an entire contract development cycle, with Python, Populus and Web3.py.

Typical iteration will include:

- Writing the contract
- Testing, fixing bugs
- Deployment to a local chain, make sure everything works
- Deployment to testnet,
- Finally deployment to mainnet which will cost real gas
- Interaction with the contract on the blockchain.

Glossary

Just a succinct reference, as a reminder if you need it during the tutorial (in a "chronological" order)

Private Key: A long combination of alphanumeric characters. There is almost zero chance that the algorithm that creates this combination will create the same combination twice.

Public Key: A combination of alphanumeric characters that is derived from the private key. It's easy to derive the public key from the private key, but the opposite is impossible.

Address: A combination of alphanumeric characters that is derived from the public key.

Ethereum Account: An address that is used on the blockchain. There are inifinite potential combinations of alphanumeric characters, but only when someone has the private key that the address was derived from, this address can be used as an *account*.

Transaction: A message that one account sends to another. The message can contain Ether (the digital currency), and data. The data is used to run a contract if the account has one.

Pending Transaction: A transaction that was sent to the network, but still waiting for a miner to include it in a block, and to the network to accept this block.

Why the private key and the public key? The keys mechanism can confirm that the transaction was indeed authorised by the account owner, that claims to sent it.

Block: A group of transactions

Mining: Bundling a group of transactions into a block

Why mining is hard? Because the miner needs to bundle the transactions with an additional input that requires significant computational effort to find. Without this addtional input, the block is not valid.

Rewards: The Ether reward that a miner gets when it finds a valid block

Blockchain: Well, it's a chain of blocks. Each block has a parent, and each time a new block is found it is added to the blockchain on top of the current last block. When a block is added to the blockchain, all the transactions in this blocks are accepted and carried out by all the nodes.

Node: A running instance of the blockchain. The nodes sync to one another. When there are conflicts, e.g. if two nodes suggest two different block for the next block, the nodes gets a decision by consensus.

Consensus: Miners get rewards when they find a valid block, and a valid block is valid only if it's built on a valid parent block, and *accepted by the majority of nodes on the blockchain*. So miners are incentivised to reject false blocks and false transactions. They know that if they work on a false transaction (say a cheat), then there is high probability that other nodes will reject it, and their work effort will be lost without rewards. They prefer to find valid blocks with valid transacions, and send them as fast as possible to the blockchain.

Uncles: Miners get rewards when they find valid blocks, even if those blocks are *not* part of the direct line of the blockchain. If the blockchain is `block1` >> `block2` >> `block3` >> `block4`, and a miner found another valid block on top of `block3`, say `block4a`, but wasn't fast enough to introduce it to the chain, it will still get *partial* rewards. The faster miner, of `block4`, will get the *full* rewards. `block4` is included in the direct sequence of the blockchain, and `block4a` is not used in the sequence but included as an "*uncle*". The idea is to spread compenstatations among miners and avoid "the winner takes it all" monopoly.

Contract: The word ""contract" is used for three different (albeit related) concepts: (1) A compiled runnable bytecode that sits on the blockchain (2) A Solidity source code contract definition (3) A Web3 contract object

EVM: The Ethereum Virtual Machine, the (quite complex) piece of code that runs the Ethereum protocol. It accepts an Assembler like instructions, and can run contracts after compilation to this Assembler bytecode.

Solidity: A programming language, similar to javascript, designed for contract authors.

Solc: Compiler of Solidity source code to the EVM bytecode

ABI: Application Binary Interface. A JSON file that describes a contract interface: the functions that the contract exposes, and their arguments. Since the contracts on the blockchain are a compiled bytecode, the EVM needs the ABI in order to know how to call the bytecode.

Web3: Client side API that lets you interact with the blockchain. Web3.js is the javascript version, Web3.py is the Python one.

geth: The official implemntation of an Ethereum blockchain node, written in Go

gas: The price that users pay to run computational actions on the blockchain (deploying a new contract, send money, run a contract function, storage, memory)

mainnet: The Ethereum blockchain

testnet: An Ethereum blockchain for testing. It behaves exactly as mainnet, but you don't use real Ether to send money and pay for the gas

Local chain: A blockchain that runs localy, has it's own blocks, and does not sync to any other blockchain. Useful for development and testing

Part 1: Solidity Contract

Start a New Project

Create a directory for the project:

```
$ mkdir projects projects/donations
$ cd projects/donations
$ populus init

Wrote default populus configuration to `./project.json`.
Created Directory: ./contracts
Created Example Contract: ./contracts/Greeter.sol
Created Directory: ./tests
Created Example Tests: ./tests/test_greeter.py
```

You just created a new populus project. Populus created an example contract called `Greeter` and some tests. To learn about the greeter example see the *Quickstart*.

We will need a local private blockchain. This local blockchain is an excellent tool for development: it runs on your machine, does not take the time to sync the real blockchain, does not costs real gas and fast to respond. Yet the local chain works with the same Ethereum protocol. If a contract runs on the local chain, it should run on mainnet as well.

Note: If you are familiar to web development, then running a local blockchain is similar to running a local website on 127.0.0.1, before publishing it to the internet.

We will create a a local chain we'll name "horton":

```
$ populus chain new horton
$ chains/horton/./init_chain.sh

INFO [10-14|12:31:31] Allocated cache and file handles         database=/home/mary/
↪projects/donations/chains/horton/chain_data/geth/chaindata cache=16 handles=16
INFO [10-14|12:31:31] Writing custom genesis block
INFO [10-14|12:31:31] Successfully wrote genesis state         database=chaindata
↪                                                                              ␣
↪hash=faa498...370bf1
INFO [10-14|12:31:31] Allocated cache and file handles         database=/home/mary/
↪projects/donations/chains/horton/chain_data/geth/lightchaindata cache=16 handles=16
INFO [10-14|12:31:31] Writing custom genesis block
INFO [10-14|12:31:31] Successfully wrote genesis state         ␣
↪database=lightchaindata                                                       ␣
↪           hash=faa498...370bf1
```

Add this local chain to the project config.

```
$ nano project.json
```

The file should look as follows. Update `ipc_path` to the actual path on your machine (if you are not sure about the path, take a look at `chains/horton/run_chain.sh`).

```
{
  "version":"7",
  "compilation":{
    "contracts_source_dirs": ["./contracts"],
    "import_remappings": []
  },
  "chains": {
    "horton": {
      "chain": {
        "class": "populus.chain.ExternalChain"
      },
      "web3": {
        "provider": {
          "class": "web3.providers.ipc.IPCProvider",
        "settings": {
          "ipc_path":"/home/mary/projects/donations/chains/horton/chain_data/geth.ipc"
        }
       }
      },
      "contracts": {
        "backends": {
          "JSONFile": {"$ref": "contracts.backends.JSONFile"},
          "ProjectContracts": {
            "$ref": "contracts.backends.ProjectContracts"
          }
        }
      }
    }
  }
}
```

For more on the horton local chain see *Running the Local Blockchain* .

Everything is ready.

Add a Contract

Ok, time to add a new contract.

```
$ nano contracts/Donator.sol
```

Note: You can work with your favourite IDE. Check for Solidity extention/package. Atom.io has some nice Solidity packages.

In this example we will work with a very simple contract that accepts donations for later use. The contract will also handle the donations value in USD.

Since the ETH/USD exchange rate fluctates, typically upward, we want to track not only how much ETH the contract collected, but also the accumulating USD value of the donations *at the time of the donation*. If the ETH rate is rising,

then we will probably see smaller donations in terms of Ether, but similar donations in terms of USD.

In other words, two donations of say $30 will have different amounts in ETH if the exchange rate changed between the donations. As a simple solution, we will ask donators to provide the effective ETH/USD exchange rate when they send their (hopefully generous) donations.

Here is the new contract code:

```
pragma solidity ^0.4.11;

/// TUTORIAL CONTRACT DO NOT USE IN PRODUCTION
/// @title Donations collecting contract


contract Donator {

    uint public donationsTotal;
    uint public donationsUsd;
    uint public donationsCount;
    uint public defaultUsdRate;

    function Donator() {
        defaultUsdRate = 350;
    }

    // fallback function
    function () payable {
        donate(defaultUsdRate);
    }

    modifier nonZeroValue() { if (!msg.value > 0) throw; _; }

    function donate(uint usd_rate) public payable nonZeroValue {
        donationsTotal += msg.value;
        donationsCount += 1;
        defaultUsdRate = usd_rate;
        uin inUsd = msg.value * usd_rate;
        donationsUsd += inUsd;
    }
}
```

Save the code to `contracts/Donator.sol`.

Quick Solidity Overview

Pragma: Every Solidity source should provide the compiler compatability: *pragma solidity ^0.4.11;*

Contract definition: The `contract` keyword starts a new contract definition, named `Donator`.

Note: Contracts names should follow class naming rules (like MyWallet, GoodLuck or WhyNot).

State variables: The contract has 4 state variables: `donationsTotal`, `donationsUsd`, `donationsCount` and `defaultUsdRate`. A state variable is defined in the *contract scope*. State variables are saved in the contract's persisten *storage*, kept after the transaction run ends, and synced to every node on the blockchain.

Visibility: The `public` decleration ensures that all state variables and the `donate` function will be available for the callers of the contrat, in the contract's interface.

Note: For the public state variables, the compiler actually creates an accessor function which if you had to type manually could look like: `function total() public returns (uint) {return donationsTotal;}`

Data types: Since we are dealing with numbers, the only data type we use here is `uint`, unsigned integer. The `int` and `uint` are declated in steps of 8 bits, `unint8`, `uint16` etc. When the bits indicator is omitted, like `int` or `uint`, the compiler will assumes `uint256`.

Note: If you know in advance the the maximum size of a variable, better to limit the type and save the gas of extra memory or storage.

As of version 0.4.17 Solidity does *not* support decimal point types. If you need decimal point, you will have to manauly handle the fixed point calculations with integers. For the sake of simplicty, the example uses only ints.

Constructor: The function `function Donator()` is a constructor. A constructor function's name is always identical to the contract's name. It runs once, when the contract is created, and can't be called again. Here we set the `defaultUsdRate`, to be used when the donator didn't provide the effective exchange rate. Providing a constructor function is optional.

Functions: The `donate` function accepts one argument: `usd_rate`. Then the function updates the total donated, both of Ether and USD value. It also updates the default USD rate and the donations counter.

Magic Variables: In every contract you get three magic variables in the global scope: `msg`, `block` and `tx`. You can use these variable without prior decleration or assignment. To find out how much Ether was sent, use `msg.value`.

Modifiers: `modifier nonZeroValue() { if (!msg.value > 0) throw; _; }`. The term "modifier" is a bit confusing. A modifier of a function is *another* function that injects, or modifies, code, typically to verify some pre-existing condition. Since the donate function uses the modifier `function donate(uint usd_rate) public payable nonZeroValue {...}`, then `nonZeroValue` will run *before* `donate`. The code in `donate` will run only if `msg.value > 0`, and make sure that the `donationsCount` does not increase by a zero donation.

Note: The modifier syntax uses `_;` to tell solidity where to insert the *modified* function. We can of course check the include the modifier condition the original function, but a declared modifier is handy when you want to use the same pre-condition validation in more than one function.

Fallback: The weired function without a name, `function () payable {...}`, is the "fallback". It calls `donate`, so when somebody just send Ether to the contract address without explicitly call `donate`, we can still accept the Ether. A fallback function is what the contract runs when called *without an explicit function name*. This happens (a) when you call a contract with `address.call`, and (b) when just send just Ether, in a transaction that don't call anything.

Note: If a contract has a fallback function, any transaction or sending of ether to an address with code will result in it's code being invoked.

Payable: `function donate(uint usd_rate) public payable nonZeroValue {...}` and `function () payable {...}` use the *payable* builtin modifier, in order to accept Ether. Otherwise, without this modifier, a transaction that sends Ether will fail. If none of the contract functions has a `payable` modifier, the contract can't accept Ether.

Initial Values: Note that `donationsTotal += msg.value;` was used before any assignment to `donationsTotal`. The variables are auto initiated with default values.

Side Note

This Donator example is fairly simple.

If you are following the Ethereum world for a while, you probably noticed that many Ethereum projects are much more complex. People and companies try to use contracts to manage distributed activity among very large groups, assuming you need special, usually complex, code and strategies that defend against bad actores. Some noticeable initiatives are the decentrelized autonomous organizations (DAO), getting groups decisions where the voting rights are proportional to the Ether the voter sent to the contract, or crowd funding with Ether, initial coin offerings (ICO), feeds that send the contract up-to-date data from the "outside world", etc.

Don't let these projects intimidate you.

If you have a simple Ethereum based idea that is useful, even for you personally, or to family and friends, go ahead and implement it. A small group of people that already know each other and **trust** each other don't need the complex overhead. Just make sure the contract code is correct. You can do really nice things, some are not possible without Ethereum.

We would be delighted to hear how it worked!

Interim Summary

So far you have:

- Initiated a project
- Initiated a local blockchain
- Added a new contract

Great. Next step is compiling and first deployment.

Part 2: Compilation and Tester Deployment

Compile

Solidity compiles the source to EVM bytecode, the operations codes that will actually run on the blockchain.

To compile, first check that solidity works on your machine. Try to get the latest version.

```
$ solc --version
solc, the solidity compiler commandline interface
Version: 0.4.13+commit.0fb4cb1a.Linux.g++
```

In the project directory:

```
$ populus compile
```

If you copy-pasted the `Donator` contract example, you will get:

```
Traceback (most recent call last):
  File "/usr/local/bin/populus", line 11, in <module>
    sys.exit(main())
  File "/usr/local/lib/python3.5/dist-packages/click/core.py", line 722, in __call__
    return self.main(*args, **kwargs)
  File "/usr/local/lib/python3.5/dist-packages/click/core.py", line 697, in main
    rv = self.invoke(ctx)
  File "/usr/local/lib/python3.5/dist-packages/click/core.py", line 1066, in invoke
    return _process_result(sub_ctx.command.invoke(sub_ctx))
  File "/usr/local/lib/python3.5/dist-packages/click/core.py", line 895, in invoke
    return ctx.invoke(self.callback, **ctx.params)
  File "/usr/local/lib/python3.5/dist-packages/click/core.py", line 535, in invoke
    return callback(*args, **kwargs)
  File "/usr/local/lib/python3.5/dist-packages/click/decorators.py", line 17, in new_
↪func
    return f(get_current_context(), *args, **kwargs)
  File "/usr/local/lib/python3.5/dist-packages/populus/cli/compile_cmd.py", line 29,␣
↪in compile_cmd
    compile_project(project, watch)
  File "/usr/local/lib/python3.5/dist-packages/populus/api/compile_contracts.py",␣
↪line 18, in compile_project
    _, compiled_contracts = compile_project_contracts(project)
  File "/usr/local/lib/python3.5/dist-packages/populus/compilation/__init__.py", line␣
↪54, in compile_project_contracts
    import_remappings=project.config.get('compilation.import_remappings'),
  File "/usr/local/lib/python3.5/dist-packages/populus/compilation/backends/solc_auto.
↪py", line 52, in get_compiled_contracts
    return self.proxy_backend.get_compiled_contracts(*args, **kwargs)
  File "/usr/local/lib/python3.5/dist-packages/populus/compilation/backends/solc_
↪standard_json.py", line 131, in get_compiled_contracts
    compilation_result = compile_standard(std_input, **command_line_options)
  File "/usr/local/lib/python3.5/dist-packages/solc/main.py", line 184, in compile_
↪standard
    message=error_message,
solc.exceptions.SolcError: contracts/Donator.sol:21:5: DeclarationError: Identifier␣
↪not found or not unique.
uin inUsd = msg.value * usd_rate;
^-^

    > command: `solc --standard-json`
    > return code: `0`
    > stderr:
    {"contracts":{},"errors":[{"component":"general","formattedMessage":"contracts/
↪Donator.sol:21:5: DeclarationError: Identifier not found or not unique.\n    uin␣
↪inUsd = msg.value * usd_rate;\n    ^-^\n","message":"Identifier not found or not␣
↪unique.","severity":"error","type":"DeclarationError"}],"sources":{}}

    > stdout:
```

What's that? actually it's not that bad. You can ignore the Python traceback, which is just the Populus call stack until the actual call to the compiler.

To undersatnd what went wrong, just look at the compiler's output. The error message is quite clear:

```
solc.exceptions.SolcError: contracts/Donator.sol:21:5: DeclarationError: Identifier␣
↪not found or not unique.
uin inUsd = msg.value * usd_rate;
```

```
^_^
```

Oh. Ok. `uin inUsd` should be `uint inUsd`. Edit and fix it:

```
$ nano contracts/Donator.sol
```

The fixed line should be:

```
uint inUsd = msg.value * usd_rate;
```

Note: Try the online IDE , which has great interactive compiler and web-form like interface to call the contract and it's funcitons.

Try to compile again:

```
populus compile
> Found 2 contract source files
  - contracts/Donator.sol
  - contracts/Greeter.sol
> Compiled 2 contracts
  - contracts/Donator.sol:Donator
  - contracts/Greeter.sol:Greeter
> Wrote compiled assets to: build/contracts.json
```

Nice. The two contracts are now compiled. Take a look at the file that Populus just added, `build/contracts.json`. The file saves some of the compilers output, which will be useful later.

Note: Compilation creates `bytecode` and `bytecode_runtime`. The `bytecode` contains the `bytecode_runtime`, as well as additional code. The additional code is required to deploy the runtime, but once deployed the runtime *is* the contract on the blockchain.

Tester Deployment

You now have two compiled contracts, ready for deployment.

The first deployment step is to verify that it works on the `tester` chain. This is an ephemeral blockchain. It runs localy, and resets each time is starts. The state of the chain when it runs is kept only in memory, and cleared when done. It's a great tool for a testing.

Deploy to the `tester` chain:

```
$ populus deploy --chain tester Donator

> Found 2 contract source files
  - contracts/Donator.sol
  - contracts/Greeter.sol
> Compiled 2 contracts
  - contracts/Donator.sol:Donator
  - contracts/Greeter.sol:Greeter

Beginning contract deployment.  Deploying 1 total contracts (1 Specified, 0 because␣
↪of library dependencies).
Donator
```

```
Deploying Donator
Deploy Transaction Sent:␣
↪0xd6de5b96feb23ce2550434a46ae9c95a9ab9c76c6274cc2b1f80e0b5a6870d11
Waiting for confirmation...

Transaction Mined
=================
Tx Hash      : 0xd6de5b96feb23ce2550434a46ae9c95a9ab9c76c6274cc2b1f80e0b5a6870d11
Address      : 0xc305c901078781c232a2a521c2af7980f8385ee9
Gas Provided : 294313
Gas Used     : 194313

Verified contract bytecode @ 0xc305c901078781c232a2a521c2af7980f8385ee9
Deployment Successful.
```

When you deploy a contract Populus re-compiles *all* the contracts, but deploys only those you asked for.

Well, deployment works. Since the `tester` chain is *not* persistent, everything was deleted, but the deployment should work on persistent chains: it's the same Ethereum protocol. Check for yourself and run the deploy again, it will re-dploy exactly the same,since each starts from a reset state.

Interim Summary

So far you have:

- Compiled the project contracts
- Verified that deployment works, using the tester chain

In the next step we will add some tests.

Part 3: First Test

Tests and Chains

For testing, we will use the `tester` chain again. It's very convinient blockchain for tests, because it reset on each run, and the state is saved only in memory and cleared after each run. In a way, this is a similar idea to running tests against a DB, where you create an ad-hoc temporary DB for the tests.

You will run Populus tests with `py.test`, which was installed when you installed Populus.

Add a test file:

```
$ nano tests/test_donator.py
```

Note: py.test collects all the tests that follow it's naming conventions

We don't need the Greeter example for this project, so delete it:

```
$ rm contracts/Greeter
4 rm tests/test_greeter.py
```

Now, before we start writing tests, pause for a moment: What are we actually testing? Obviously a contract, but what *is* a contract?

What is a Contract?

The simplest definition of a contract is a compiled program that runs on the Ethereum blockchain. It's a bytecode of instructions, saved on the blockchain, that a blockchain node can run. The node gives this program a sandbox, a closed environment, to execute.

In a way, if the blockchain is the OS, then a contract is an executable that runs by the OS. Everything else: syncing nodes, consensus, mining, is OS stuff. Yes, it's a smart OS that magically syncs a lot of nodes, but the contract doesn't care - it just runs, and is allowed to use the API that the OS gives it.

However, the term *contract* can be confusing, since it's used for several different things:

1. A bytecode program that is saved on the blockchain
2. A `contract` code in a Solidity source file
3. A Web3 `contract` object

Luckily, it can be even more confusing. You may have a Solidity source file of the `Donator` contract. If you deployed it to `testnet`, then we have another "contract", the code that sits on `testnet`. When you deploy to `mainnet`, which is another blockchain, we now have three contracts: the source file, the bytecode on `testnet`, and the bytecode on `mainnet`.

But wait, there is more! To interact with an *existing* contract on say `mainnet`, we need a Web3 "contract" *object*. This object does not need the solidity source, since the bytcode is alreadt compiled and deployed. It does need the ABI: the ABI is the detailed specification of the functions and arguments structure of the *bytecode* contract's interface, and the address of this *bytecode* contract on the mainnet. Again, a contract might be the bytecode on the blockchain, a Solidty source, or a web3.py contract object.

Testing a Contract

Get the contract object

The answer to the question "what are we actually testing" is: We test a bytecode program that runs on a blockchain. We test a contract *instance*.

Ideally, for testing, we would need to take the Solidity source file, compile it, deploy it to a blockchain, create a Web3 contract object that points to this instance, and handover this object to the test function so we can test it.

And here is where you will start to appreciate Populus, **which does all that for you in one line of code.**

Add the first test:

```
$ nano tests/test_donatory.py
```

The test file should look as follows:

```
def test_defaultUsdRate(chain):
    donator, deploy_tx_hash _ = chain.provider.get_or_deploy_contract('Donator')
    defaultUsdRate = donator.call().defaultUsdRate()
    assert defaultUsdRate == 350
```

The magic happens with `get_or_deploy_contract`. This function gets an existing contract if it exists on the blockchain, and if it doesn't, it compiles the Solidity source, deploys it to the blockchain, creates a `Contract` object, exposes the deployed contract as a *python object with python functions*, and returns this object to the test function.

From this point onward, you have a *Python* object, with *Python* methods, that correspond to the original deployed contract bytecode on the blockchain. Cool, isn't it?

Note: For the contract name you use the Solidity contract name, `Donator`, and *not* the file name, `Donator.sol`. A Solidity source file can include more than one contract definition (as a Python file can include more than one class definition).

Get the blockchain

Another bonus is the `chain` object, provided as an argument at `def test_defaultUsdRate(chain)`. It gives the test function a Python object that corresponds to a running blockchain, the `tester` blockchain. Reminder: The `tester` chain is ephemeral, saved only in memory, and will reset on every test run.

The `chain` argument is a py.test *fixture*: in py.test world it's a special argument that the test function can accept. You don't have to declare or assign it, it's just ready and available for your test.

The Populus testing fixtures comes from the Populus py.test plug-in, which prepares for you several useful fixtures: `project`, `chain`, `provider`, `registrar` and `web3`. All these fixtures are part of the Populus API. See *Testing*

Note: The `tester` also chain creates and unlocks new accounts in each run, so you don't have to supply a private key or a wallet.

Run the First Test: Public State Variable

Ready for the first test: we have a test function that runs the `tester` chain. Using `get_or_deploy_contract('Donator')` it compiles `Donator.sol`, deploys it to the `tester` chain, gets a Python contract object that wraps the actual contract's bytecode on the chain, and assigns this object to a variable, `donator`.

Once we have the `donator` contract as a Python object, we can call any function of this contract. You get the *contract's* interface with `call()`. Reminder: `call` behaves exactly as a transaction, but does not alter state. It's like a "dry-run". It's also useful to query the current state, without changing it.

The first test important line is:

```
defaultUsdRate = donator.call().defaultUsdRate()
```

In the Solidity source code we had:

```
...
uint public defaultUsdRate;
...
function Donator() {
  defaultUsdRate = 350;
}
...
```

To recap, `defaultUsdRate` is a `public` variable, hence the compiler automatically created an accessor function, a "get", that returns this variable. The test just used this function.

What is the expected return value? It's 350. We assigned to it 350 in the *constructor*, the function that runs once, when the contract is created. The test function should deploy `Donator` on the `tester` chain, but nothing else is called afterwards, so the initial value should not be changed.

Run the test:

```
$ py.test --disable-pytest-warnings

platform linux -- Python 3.5.2, pytest-3.1.3, py-1.4.34, pluggy-0.4.0
rootdir: /home/mary/projects/donations, inifile:
plugins: populus-1.8.0, hypothesis-3.14.0
collected 1 item s

tests/test_donator.py .

================================================== 1 passed, 5 warnings in 0.29␣
↪seconds ======
```

Note: Usually you don't want to use `--disable-pytest-warnings`, because the warnings provide important infromation. We use it here to make the output less confusing, for the tutorial only.

Interim Summary

Congrats. Your first project test just passed.

Continue to a few more.

Part 4: Transaction Tests

Test a Contract Function

Edit the tests file and add another test:

```
$ nano contracts/test_donator.py
```

After the edit, the file should look as follows:

```
def test_defaultUsdRate(chain):
    donator, deploy_tx_hash = chain.provider.get_or_deploy_contract('Donator')
    defaultUsdRate = donator.call().defaultUsdRate()
    assert defaultUsdRate == 350


def test_donate(chain):
    donator, deploy_tx_hash = chain.provider.get_or_deploy_contract('Donator')
    donator.transact({'value':500}).donate(37)
    donator.transact({'value':650}).donate(38)
    donationsCount = donator.call().donationsCount()
    donationsTotal = donator.call().donationsTotal()
    defaultUsdRate = donator.call().defaultUsdRate()

    assert donationsTotal == 1150
    assert donationsCount == 2
    assert defaultUsdRate == 380
```

You added another test, `test_donations`. The second test is similar to the first one:

[1] Get the chain: The test function accepts the `chain` argument, the auto-generated Python object that corresponds to a `tester` chain. Reminder: the `tester` chain is ephimeral, in memory, and reset on each test function.

[2] Get the contract: With the magic function `get_or_deploy_contract` Populus compiles the *Donator* contract, deploys it to the `chain`, creates a Web3 `contract` object, and returns it to the function as a Python object with Python methods. This object is stored in the `donator` variable.

[3] The "transact" function:

```
donator.transact({'value':500}).donate(37)
```

Reminder: we have two options to interact with a contract on the blockchain, *transactions* and *calls*. With Populus, you initiate a transaction with `transact`, and a call with `call`:

- *Transactions*: Send a transaction, run the contract code, transfer funds, and *change* the state of the contract and it's balance. This change will be permenant, and synced to the entire blockchain.
- *Call*: Behaves exactly as a transaction, but once done, everything is revert and no state is changed. A call is kinda "dry-run", and an efficient way to query the current state without expensive gas costs.

[4] Test transactions: The test commits two transactions, and send funds in both. In the first the `value` of the funds is 500, and in the second the `value` is 650. The `value` is provided as a `transact` argument, in a dictionary, where you can add more kwargs of an Ethereum transaction.

Note: Since these are *transactions*, they will change state, and in the case of the `tester` chain this state will persist until the test function quits.

---

[5] Providing arguments: The donate function in the contract accepts one argument

```
function donate(uint usd_rate) public payable nonZeroValue {...}
```

This argument is provided in the test as *Python* donate function:

```
donator.transact({'value':650}).donate(38).
```

Populus gives you a *Python* interface to a bytecode contract. Nice, no?

[6] Asserts: We expect the `donationsTotal` to be `500 + 650 = 1150`, the `donationsCount` is 2, and the `defaultUsdRate` to match the last update, 380.

The test gets the varaibles with `call`, and should update instanrly because it's a local `tester` chain. On a distributed blockchain it will take sometime until the transactions are mined and actually change the state.

Run the test:

```
$ py.test --disable-pytest-warnings

platform linux -- Python 3.5.2, pytest-3.1.3, py-1.4.34, pluggy-0.4.0
rootdir: /home/mary/projects/donations, inifile:
plugins: populus-1.8.0, hypothesis-3.14.0
collected 2 items

tests/test_donator.py ..

====================== 2 passed, 10 warnings in 0.58 seconds =============
```

Voila. The two tests pass.

Test Calculations

The next one will test the ETH/USD calculations:

```
$ nano tests/test_donator.py
```

Add the following test to the bottom of the file:

```
def test_usd_calculation(chain):

    ONE_ETH_IN_WEI = 10**18  # 1 ETH == 1,000,000,000,000,000,000 Wei

    donator, deploy_tx_hash = chain.provider.get_or_deploy_contract('Donator')
    donator.transact({'value':ONE_ETH_IN_WEI}).donate(4)
    donator.transact({'value':(2 * ONE_ETH_IN_WEI)}).donate(5)
    donationsUsd = donator.call().donationsUsd()

    # donated 1 ETH in  $4 per ETH = $4
    # donated 2 ETH in $5 per ETH = 2 * $5 = $10
    # total $ value donated = $4 + $10 = $14
    assert donationsUsd == 14
```

The test sends donations worth of 3 Ether. Reminder: by default, all contract functions and contract interactions are handled in *Wei*.

In 1 Ether we have 10^18 Wei (see the Ether units denominations)

The test runs two transactions: note the `transact` function, which will change the contract state and balance on the blockchain. We use the `tester` chain, so the state is reset on each test run.

First transaction

```
donator.transact({'value':ONE_ETH_IN_WEI}).donate(4)
```

Donate Wei worth of 1 Ether, where the effective ETH/USD rate is $4. That is, $4 per Ether, and a total *USD* value of $4

Second transaction

```
donator.transact({'value':(2 * ONE_ETH_IN_WEI)}).donate(5)
```

Donate Wei worth of *2* Ether, where the effective ETH/USD rate is $5 (no markets sepculations on the tutorial) It's $5 per Ether, and total *USD* value of 2 * $5 = $10

Hence we excpect the total *USD* value of these two donations to be $4 + $10 = $14

```
donationsUsd = donator.call().donationsUsd()
assert donationsUsd == 14
```

OK, that wan't too complicated. Run the test:

```
$ py.test --disable-pytest-warnings
```

And the py.test results:

```
platform linux -- Python 3.5.2, pytest-3.1.3, py-1.4.34, pluggy-0.4.0
rootdir: /home/mary/projects/donations, inifile:
plugins: populus-1.8.0, hypothesis-3.14.0
collected 3 items

tests/test_donator.py ..F

================================ FAILURES␣
↪=============================================================
_____________________________ test_usd_calculation ____________________________________
↪____________

chain = <populus.chain.tester.TesterChain object at 0x7f2736d1c630>

    def test_usd_calculation(chain):

        ONE_ETH_IN_WEI = 10**18  # 1 ETH == 1,000,000,000,000,000,000 Wei

        donator, deploy_tx_hash = chain.provider.get_or_deploy_contract('Donator')
        donator.transact({'value':ONE_ETH_IN_WEI}).donate(4)
        donator.transact({'value':(2 * ONE_ETH_IN_WEI)}).donate(5)
        donationsUsd = donator.call().donationsUsd()

        # donated 1 ETH at $4 per ETH = $4
        # donated 2 ETH at $5 per ETH = 2 * $5 = $10
        # total $ value donated = $4 + $10 = $14
>       assert donationsUsd == 14
E       assert 14000000000000000000 == 14

tests/test_donator.py:32: AssertionError
====================================== 1 failed, 2 passed, 15 warnings in 0.95␣
↪seconds =========
```

Ooops. Something went wrong. But this is what tests are all about.

Py.test tells us that the assert failed. Instead of 14, the `donationsUsd` is 14000000000000000000. And you know the saying: a billion here, a billion there, and pretty soon you're talking about real money.

Where is the bug? you maybe guessed it already, but let's take a look at the contract's `donate` function:

```
function donate(uint usd_rate) public payable nonZeroValue {
    donationsTotal += msg.value;
    donationsCount += 1;
    defaultUsdRate = usd_rate;
    uint inUsd = msg.value * usd_rate;
    donationsUsd += inUsd;
    }
```

Now it's clear:

```
uint inUsd = msg.value * usd_rate;
```

This line multiplies `msg.value`, which is in Wei, by `usd_rate`, which is the exchange rate per *Ether*.

Reminder: as of 0.4.17 Solidity does not have a workable decimal point calculation, and you have to handle fixed-point with integers. For the sake of simplicity, we will stay with ints.

Edit the contract:

```
$ nano contracts/Donator.sol
```

We could fix the line into:

```
uint inUsd = msg.value * usd_rate / 10**18;
```

But Solidity can do the math for you, and Ether units are reserved words. So fix to:

```
uint inUsd = msg.value * usd_rate / 1 ether;
```

Run the tests again:

```
$ py.test --disable-pytest-warnings

=================================== test session starts ===================
platform linux -- Python 3.5.2, pytest-3.1.3, py-1.4.34, pluggy-0.4.0
rootdir: /home/mary/projects/donations, inifile:
plugins: populus-1.8.0, hypothesis-3.14.0
collected 3 items

tests/test_donator.py ...

============================== 3 passed, 15 warnings in 0.93 seconds =======
```

Easy.

Warning: Note that if this contract was running on `mainent`, you could not fix it, and probably had to deploy a new one and loose the current contract and the money paid for it. This is why testing *beforehand* is so important with smart contracts.

Interim Summary

- Three tests pass

- Transactions tests pass
- Exchange rate calculations pass
- You fixed a bug in the contract source code.

The contract seems Ok, but to be on the safe side, we will run next a few tests for the edge cases.

Part 5: Edge Cases Tests

Test a Modifier "throw" Exception

In the contract we used a modifier, to enforce a pre-condition on the `donate` function: A donation should not be of value 0, otherwise the modifier will `throw`. We wanted this modifier to make sure that the donations counter will not increment for zero donations:

```
modifier nonZeroValue() { if (!(msg.value > 0)) throw; _; }
function donate(uint usd_rate) public payable nonZeroValue {...}
```

Edit the tests file:

```
$ nano tests/test_donator.py
```

And add the following test to the bottom of the file:

```
import pytest
from ethereum.tester import TransactionFailed


def test_modifier(chain):

    donator, deploy_tx_hash = chain.provider.get_or_deploy_contract('Donator')
    with pytest.raises(TransactionFailed):
        donator.transact({'value':0}).donate(4)

    defaultUsdRate = donator.call().defaultUsdRate()
    assert defaultUsdRate == 350
```

Simple test. Note the py.test syntax for *expected* exceptions: `with pytest.raises(...)`.

The test transaction is of 0 value:

```
donator.transact({'value':0}).donate(4)
```

So the modifier should `throw`.

Since the transaction should fail, the `defaultUsdRate` should remain the same, with the original initialisation of the constructor

```
function Donator() {
defaultUsdRate = 350;
}
```

And ignore the test transaction with `.donate(4)`.

Run the tests:

```
$ py.test --disable-pytest-warnings

====================================== test session starts ===============
platform linux -- Python 3.5.2, pytest-3.1.3, py-1.4.34, pluggy-0.4.0
rootdir: /home/mary/projects/donations, inifile:
plugins: populus-1.8.0, hypothesis-3.14.0
collected 4 items

tests/test_donator.py ....

======================== 4 passed, 20 warnings in 1.07 seconds =========
```

Works, all 4 tests passed.

Fallback Function

At the moment, Populus does not have a builtin option to call fallback function. To run it, you can send a simple transaction with Web3, or even better: write another function that you can test, and let the `fallback` only call this function

Interim Summary

- Working Contract
- All tests pass

The next step is to deploy the contract to a persisetent chain.

Part 6: Contract Instance on a Local Chain

Deploy to a Local Chain

So far we worked with the `tester` chain, which is ephimeral: it runs only on memory, reset in each test, and nothing is saved after it's done.

The easiest *persisten* chain to start with, is a private local chain. It runs on your machine, saved to hard drive, for persistency, and fast to respond. Yet it keeps the same Ethereum protocol, so everything that works locally should work on `testnet` and `mainnet`. See *Running the Local Blockchain*

You already have `horton`, a local chain, which you set when you started the project.

Run this chain:

```
$ chains/horton/./run_chain.sh
```

And you will see that geth starts to do it's thing:

```
INFO [10-18|19:11:30] Starting peer-to-peer node               instance=Geth/v1.6.7-
↪stable-ab5646c5/linux-amd64/go1.8.1
INFO [10-18|19:11:30] Allocated cache and file handles         database=/home/mary/
↪projects/donations/chains/horton/chain_data/geth/chaindata cache=128 handles=1024
INFO [10-18|19:11:30] Initialised chain configuration          config="{ChainID: <nil>
↪ Homestead: 0 DAO: 0 DAOSupport: false EIP150: <nil> EIP155: <nil> EIP158: <nil>␣
↪Metropolis: <nil> Engine: unknown}"
INFO [10-18|19:11:30] Disk storage enabled for ethash caches   dir=/home/mary/
↪projects/donations/chains/horton/chain_data/geth/ethash count=3
INFO [10-18|19:11:30] Disk storage enabled for ethash DAGs     dir=/home/mary/.ethash
```

The chain runs as an independent geth process, that is not related to Populus (Populus just created the setup files). Let the chain run. Open another terminal, and deploy the contract to `horton`:

```
$ populus deploy --chain horton Donator --no-wait-for-sync

> Found 1 contract source files
- contracts/Donator.sol
> Compiled 1 contracts
- contracts/Donator.sol:Donator
Beginning contract deployment.  Deploying 1 total contracts (1 Specified, 0 because␣
↪of library dependencies).
Donator

Deploy Transaction Sent:␣
↪0xc2d2bf95b7de4f63eb5712d51b8d6ebe200823e0c0aed524e60a411dac379dbc
Waiting for confirmation...
```

And after a few seconds the transaction is mined:

```
Transaction Mined
=================
Tx Hash      : 0xc2d2bf95b7de4f63eb5712d51b8d6ebe200823e0c0aed524e60a411dac379dbc
Address      : 0xb8d9d2afbe18fd6ac43042164ece9691eb9288ed
Gas Provided : 301632
Gas Used     : 201631


Verified contract bytecode @ 0xb8d9d2afbe18fd6ac43042164ece9691eb9288ed
Deployment Successful.
```

If you looked at the other terminal window, where the chain is running, you would also see the contract creation transaction:

```
INFO [10-18|19:28:58] Commit new mining work                   number=62 txs=0␣
↪uncles=0 elapsed=139.867µs
INFO [10-18|19:29:02] Submitted contract creation              ␣
↪fullhash=0xc2d2bf95b7de4f63eb5712d51b8d6ebe200823e0c0aed524e60a411dac379dbc␣
↪contract=0xb8d9d2afbe18fd6ac43042164ece9691eb9288ed
```

```
INFO [10-18|19:29:03] Successfully sealed new block            number=62␣
↪hash=183d75...b0ce05
INFO [10-18|19:29:03]  block reached canonical chain           number=57␣
↪hash=1f9cc1...b2ebe3
INFO [10-18|19:29:03]  mined potential block                   number=62␣
↪hash=183d75...b0ce05
```

Note that when Populus created `horton`, it also created a wallet file, a password, and added the `unlock` and `password` arguments to the `geth` command in the run script, `run_chain.sh`.

The same account also gets an allocation of (dummy) Ether in the first block of the `horton` local chain, and this is why we can use `--no-wait-for-sync`. Otherwise, if your account get money from a transaction in a far (far away) block, that was not synced yet localy, geth thinks that you don't have the funds for gas, and refuses to deploy until you sync.

Note: When you work with `mainnet` and `testnet` you will need to create your own wallet, password, and get some Ether (dummy Ether in the case of testnet) for the gas. See *Part 3: Deploy to a Local Chain*

Persistence of the Contract Instance

Unlike the previous runs of the tests on the `tester` chain, this time the contract instance is persistent on the local `horton` blockchain.

Check for yourself. Add the following script to your project.

```
$ mkdir scripts
$ nano scripts/donator.py
```

The script should look as follows:

```
from populus.project import Project

p = Project(project_dir="/home/mary/projects/donations/")
with p.get_chain('horton') as chain:
    donator, deploy_tx_hash = chain.provider.get_or_deploy_contract('Donator')

print("Donator address on horton is {address}".format(address=donator.address))
if deploy_tx_hash is None:
    print("The contract is already deployed on the chain")
else:
    print("Deploy Transaction {tx}".format(tx=deploy_tx_hash))
```

It starts by initiating a Populus `Project`. The Project is the enrty point to the Populus API, where you can get all the relevant resources programatically.

Note: we used an *absolute* path, so this script can be saved and run from anywhere on your machine.

The next line gets the `horton` chain object:

```
with p.get_chain('horton') as chain
```

Using `get_chain`, the Populus Project object has access to any chain that is defined in the project's configuration file, `project.json`, and the user-scope configuration file, `~/.popuplus/config.json`. Go ahead and take

a look at the `chains` key in those files. Populus' config files are in JSON: not so pretty to the Pythonic developer habbits, but for blockchain development it safer to use non programmble, static, external files (and hey, you got to admit that Populus saves you form quite a lot javascript).

The `chain` is wrapped in a *context manager*, because it needs to run initialisation code when it starts, and cleanup when done. The code inside the `with` clause runs after initialisation, and when it finishes, python runs the exit code for you.

The next line should be familiar to you by now:

```
donator, deploy_tx_hash = chain.provider.get_or_deploy_contract('Donator')
```

Populus does it's magic:

New: If the contract was *never* deployed to a blockchain, compile the source, deploy to the chain, create a Web3 contract Python object instance, which points to the blockchain bytecode, and returns this Python object.

Existing: If the contract *was already deployed*, that is the contract's bytecode *already* sits on the blockchain and has an address, populus just create the Python object instance for this bytecode.

Time to check it, just make sure that the `horton` chain runs (`chains/horton/./run_chain.sh`).

Run the script:

```
$ python scripts/donator.py

Donator address on horton is 0xb8d9d2afbe18fd6ac43042164ece9691eb9288ed
The contract is already deployed on the chain
```

Ok, Populus found the contract on the chain, at exactly the same address.

Note: You may need to run the script with `$ python3`

To make sure it's persistent, stop the chain, then run it again. Type Ctrl+C in the running chain window:

```
INFO [10-19|05:28:09] WebSocket endpoint closed: ws://127.0.0.1:8546
INFO [10-19|05:28:09] HTTP endpoint closed: http://127.0.0.1:8545
INFO [10-19|05:28:09] IPC endpoint closed: /home/mary/projects/donations/chains/
↪horton/chain_data/geth.ipc
INFO [10-19|05:28:09] Blockchain manager stopped
INFO [10-19|05:28:09] Stopping Ethereum protocol
INFO [10-19|05:28:09] Ethereum protocol stopped
INFO [10-19|05:28:09] Transaction pool stopped
INFO [10-19|05:28:09] Database closed     database=/home/mary/projects/donations/
↪chains/horton/chain_data/geth/chaindata
```

Geth stopped. Re-run it:

```
$ chains/horton/./run_chain.sh

INFO [10-19|05:34:23] Starting peer-to-peer node               instance=Geth/v1.6.7-
↪stable-ab5646c5/linux-amd64/go1.8.1
INFO [10-19|05:34:23] Allocated cache and file handles         database=/home/mary/
↪projects/donations/chains/horton/chain_data/geth/chaindata cache=128 handles=1024
INFO [10-19|05:34:23] Initialised chain configuration          config="{ChainID: <nil>
↪ Homestead: 0 DAO: 0 DAOSupport: false EIP150: <nil> EIP155: <nil> EIP158: <nil>␣
↪Metropolis: <nil> Engine: unknown}"
```

Then, in another terminal window run the script again:

```
$ python scripts/donator.py

Donator address on horton is 0xb8d9d2afbe18fd6ac43042164ece9691eb9288ed
The contract is already deployed on the chain
```

Same contract, *same* address. The contract is persistent on the blockchain. It is *not* re-deployed on each run, like the `tester` in-memory ephemeral chain that we used in the tests, and was reset for each test.

Note: Persistence for a local chain is simply it's data directory on your local hard-drive. It's a one-peer chain. On `mainnet` and `testnet` this persistency is synced between many nodes on the blockchain. However the concept is the same: a persistent contract.

Registrar

When Populus deploys a contract to a blockchain, is saves the deployment details in `registrar. json` file. This is how you project directory should look:

```
- build
|   - contracts.json
- chains
|   - horton
|       - chain_data
|       |     |
|       |     - ...
|       - nodekey
|       |   - keystore
|       |       - UTC--...
|       - genesis.json
|       - init_chain.sh
|       - password
|       - run_chain.sh
- contracts
|   - Donator.sol
- project.json
- registrar.json
- scripts
|   - donator.py
- tests
    - test_donator.py
```

The *registrar* is loaded with `get_or_deploy_contract`, and if Populus finds an entry for a contract, it knows that the contract already deployed, and it's address on this chain.

Take a look at the registrar:

```
  $ cat registrar.json

  {
"deployments": {
  "blockchain://c77836f10cb9691c430638647b95701568ace603d0876ff41c6f0b61218254b4/
↪block/667aa2e5f0dea4087b645a9287efa181cf6dad4ed96516b63aefb7ef5c4b1dff": {
    "Donator": "0xb8d9d2afbe18fd6ac43042164ece9691eb9288ed"
```

```
  }
}
```

The registrar saves a deployment reference with unique "signature" of the blockchain that the contract was deployed to. The signature is the first block hash, which is obviously unique. It appears after the `blockchain://` part. Then the hash of the latest block at the time of deployment after, `block`. The registrar uses a special URI structure designed for blockchains, which is built from a resource name (blockchain, block, etc) and it's hash. See BIP122 URI

To have *another* contract deployed to the *same* chain, we will greet our good ol' friend, the Greeter. Yes, you probably missed it too.

```
$ nano contracts/Greete.sol
```

Edit the contract file:

```
pragma solidity ^0.4.0;

contract Greeter {
    string public greeting;

    // TODO: Populus seems to get no bytecode if `internal`
    function Greeter() public {
        greeting = 'Hello';
    }

    function setGreeting(string _greeting) public {
        greeting = _greeting;
    }

    function greet() public constant returns (string) {
        return greeting;
    }
}
```

Deploy to `horton`, after you make sure the chain runs:

```
$ populus deploy --chain horton Greeter --no-wait-for-sync
```

You should see the usuall deployment log, and in a few seconds the contract creation transaction is picked and mined:

```
Transaction Mined
=================
Tx Hash      : 0x5df249ed014b396655724bd572b4e44cbc173ab1b5ba5fdc61d541a39daa6d59
Address      : 0xc5697df77a7f35dd1eb643fc2826c79d95b0bd76
Gas Provided : 465580
Gas Used     : 365579


Verified contract bytecode @ 0xc5697df77a7f35dd1eb643fc2826c79d95b0bd76
Deployment Successful.
```

Now we have two deployments to `horton`:

```
  $ cat registrar.json

  {
"deployments": {
  "blockchain://c77836f10cb9691c430638647b95701568ace603d0876ff41c6f0b61218254b4/
↪block/34f52122cf90aa2ad90bbab34e7ff23bb8619d4abb2d8e66c52806ec9b992986": {
```

```
    "Greeter": "0xc5697df77a7f35dd1eb643fc2826c79d95b0bd76"
  },
  "blockchain://c77836f10cb9691c430638647b95701568ace603d0876ff41c6f0b61218254b4/
↪block/667aa2e5f0dea4087b645a9287efa181cf6dad4ed96516b63aefb7ef5c4b1dff": {
    "Donator": "0xb8d9d2afbe18fd6ac43042164ece9691eb9288ed"
  }
}
```

The blockchain id for the two deployments is the same, but the latest block at the time of deployment is obviously different.

Note: Don't edit the registrar yourself unless you know what you are doing. The edge case that justifies such edit is when you have a a project with contract that is already deployed on the `mainnet`, and you want to include it in another project. Re-deployment will waste gas. Otherwise, you can just re-deploy.

Contract Instances on More than One Chain

We will create another instance of `Donator`, but on another chain we'll name `morty`.

Create and init the chain:

```
$ populus chain new morty
$ chains/morty/./init_chain.sh
```

Edit the project config file to include the new `morty` chain:

```
$ nano project.json
```

The file should look as follows:

```
{
  "version":"7",
  "compilation":{
    "contracts_source_dirs": ["./contracts"],
    "import_remappings": []
  },
  "chains": {
    "horton": {
      "chain": {
        "class": "populus.chain.ExternalChain"
      },
      "web3": {
        "provider": {
          "class": "web3.providers.ipc.IPCProvider",
        "settings": {
          "ipc_path":"/home/mary/projects/donations/chains/horton/chain_data/geth.ipc"
        }
       }
      },
      "contracts": {
        "backends": {
          "JSONFile": {"$ref": "contracts.backends.JSONFile"},
          "ProjectContracts": {
            "$ref": "contracts.backends.ProjectContracts"
          }
```

```
        }
      }
    },
    "morty": {
      "chain": {
        "class": "populus.chain.ExternalChain"
      },
      "web3": {
        "provider": {
          "class": "web3.providers.ipc.IPCProvider",
        "settings": {
          "ipc_path":"/home/mary/projects/donations/chains/morty/chain_data/geth.ipc"
        }
       }
      },
      "contracts": {
        "backends": {
          "JSONFile": {"$ref": "contracts.backends.JSONFile"},
          "ProjectContracts": {
            "$ref": "contracts.backends.ProjectContracts"
          }
        }
      }
    }
  }
}
```

Fix the `ipc_path` to the actual ipc_path on your machine, you can see it in the run file at `chains/morty/run_chain.sh`.

Run the **horton** chain:

```
$ chains/horton/./run_chain.sh
```

And try to deploy again `Donator` to `horton`, although we know it's already deployed to this chain:

```
   $ populus deploy --chain horton Donator --no-wait-for-sync

    Found 2 contract source files
 - contracts/Donator.sol
 - contracts/Greeter.sol
 > Compiled 2 contracts
 - contracts/Donator.sol:Donator
 - contracts/Greeter.sol:Greeter
 Beginning contract deployment.  Deploying 1 total contracts (1 Specified, 0 because␣
↪of library dependencies).

Donator
Found existing version of Donator in registrar. Would you like to use
the previously deployed contract @ 0xb8d9d2afbe18fd6ac43042164ece9691eb9288ed? [True]:
```

Populus found a matching entry for `Donator` deployment on the `horton` chain, and suggest to use it. For now, accept and prompt `True`:

```
Deployment Successful.
```

Success message, but without a new transaction and new deployment, just use the already deployed instance on `horton`.

Good. Stop the `horton` chain with Ctrl+C, and start **morty**:

```
$ chains/morty/./run_chain.sh

INFO [10-19|09:41:28] Starting peer-to-peer node instance=Geth/v1.6.7-stable-ab5646c5/
↪linux-amd64/go1.8.1
```

And deploy to `morty`:

```
$ populus deploy --chain morty Donator --no-wait-for-sync
```

This time a new contract is deployed to the `morty` chain:

```
> Found 2 contract source files
- contracts/Donator.sol
- contracts/Greeter.sol
> Compiled 2 contracts
- contracts/Donator.sol:Donator
- contracts/Greeter.sol:Greeter
Beginning contract deployment.  Deploying 1 total contracts (1 Specified, 0 because␣
↪of library dependencies).

  Donator
  Deploy Transaction Sent:␣
↪0x842272f0f2b1f026c6ef003769b1f6acc1b1e43eac0d053541f218e795615142
  Waiting for confirmation...

  Transaction Mined
  =================
  Tx Hash      : 0x842272f0f2b1f026c6ef003769b1f6acc1b1e43eac0d053541f218e795615142
  Address      : 0xcffb2715ead1e0278995cdd6d1736a60ff50c6a5
  Gas Provided : 301632
  Gas Used     : 201631
```

Registrar:

```
$ cat registrar.json
```

Now with the new deployment:

```
    {
  "deployments": {
    "blockchain://927b61e39ed1e14a6e8e8b3d166044737babbadda3fa704b8ca860376fe3e90b/
↪block/2e9002f82cc4c834369039b87916be541feb4e2ff49036cafa95a23b45ecce73": {
      "Donator": "0xcffb2715ead1e0278995cdd6d1736a60ff50c6a5"
    },
    "blockchain://c77836f10cb9691c430638647b95701568ace603d0876ff41c6f0b61218254b4/
↪block/34f52122cf90aa2ad90bbab34e7ff23bb8619d4abb2d8e66c52806ec9b992986": {
      "Greeter": "0xc5697df77a7f35dd1eb643fc2826c79d95b0bd76"
    },
    "blockchain://c77836f10cb9691c430638647b95701568ace603d0876ff41c6f0b61218254b4/
↪block/667aa2e5f0dea4087b645a9287efa181cf6dad4ed96516b63aefb7ef5c4b1dff": {
      "Donator": "0xb8d9d2afbe18fd6ac43042164ece9691eb9288ed"
    }
  }
}
```

To summarise, the project has two Solidity source files with two contracts. Both are deployed to the `horton` chain, `blockchain://c77836f1...`. The `Donator` contract has *another* contract instance on the `morty` chain,

`blockchain://927b61e3...`. So the project has 3 contract instances on 2 chains.

Note: It is very common to have more than on contract instance per source file. You can have one on a local chain, on `testnet`, and production on `mainent`

Interim Summary

- You deployed a persistent contract instance to a local chain
- You interacted with the `Project` object, which is the entry point to the Populus API
- You deployed the same Solidity source file on two seprated local chains, `horton` and `morty`
- Deployments are saved in the the `registrar`

Part 7: Interacting With a Contract Instance

Python Objects for Contract Instances

A contract instance is a bytecode on the blockchain at a specific address. Populus and Web3 give you a Python object with Python methods, which correspond to this bytecode. You interact with this local *Python* object, but behind the scenes these Python interactions are sent to a the bytecode on the blockchain.

To find and interact with a contract on the blockchain, this local contract object needs an *address* and the *ABI* (application binary interface).

Reminder: the contract instance is compiled, a bytecode, so the EVM (ethereum virtual machine) needs the ABI in order to call this bytecode. The ABI (application binary interface) is essentially a JSON file with detailed description of the functions and their arguments, which tells how to call them. It's part of the compiler output.

Populus does not ask you for the address and the ABI of the projects' contracts: it already has the address in the *registrar* file at `registrar.json`, and the ABI in `build/contracts.json`

However, if you want to interact with contract instances from other projects, or even deployed by others, you will need to create a Web3 contract yourself and manually provide the address and the ABI. See Web3 Contracts

Warning: When you call a contract that you didn't compile and didn't deploy yourself, you should be 100% sure that it's trusted, that the author is trusted, and only only after you got a version of the Solidity contract's source, and verified that the compilation of this source is *identical* to the bytecode on the blockchain.

Call an Instance Function

A `call` is a contract instance invocation that doesn't change state. Since the state is *not* changed, there is no need to create a transaction, to mine the transaction into a block, and to propagate it and sync to the entire blockchain. The `call` runs only on the one node you are connected to, and the node reverts everything when the `call` is finished - and saves you the expensive gas.

Calls are useful to query an *existing* contract state, without any changes, when a local synced node can just hand you this info. It's also useful as a "dry-run" for transactions: you run a ''call'', make sure everything is working, then send the real transaction.

To access a conract function with `call`, in the same way you have done with the tests, use `contract_obj.call().foo(arg1,arg2...)` where `foo` is the contract function. Then `call()` returns an object that exposed the contract instance functions in Python.

To see an example, edit the script:

```
$ nano scripts/donator.py
```

And add a few lines, as follows:

```
from populus.project import Project

p = Project(project_dir="/home/mary/projects/donations/")
with p.get_chain('horton') as chain:
    donator, deploy_tx_hash = chain.provider.get_or_deploy_contract('Donator')

print("Donator address on horton is {address}".format(address=donator.address))
if deploy_tx_hash is None:
    print("The contract is already deployed on the chain")
else:
    print("Deploy Transaction {tx}".format(tx=deploy_tx_hash))

# Get contract state with calls
donationsCount = donator.call().donationsCount()
donationsTotal = donator.call().donationsTotal()

# Client side
ONE_ETH_IN_WEI = 10**18  # 1 ETH == 1,000,000,000,000,000,000 Wei
total_ether = donationsTotal/ONE_ETH_IN_WEI
avg_donation = donationsTotal/donationsCount if donationsCount > 0 else 0
status_msg = (
    "Total of {:,.2f} Ether accepted in {:,} donations, "
    "an avergage of {:,.2f} Wei per donation."
    )

print (status_msg.format(total_ether, donationsCount, avg_donation))
```

Pretty much similar to what we did so far: The script starts with the `Project` object, the main entry point to the Populus API. The project object provides a `chain` object (as long as this chain is defined in the project-scope or user-scope configs), and once you have the `chain` you can get the contract *instance* on that chain.

Then we get the `donationsCount` and the `donationsTotal` with `call`. Populus, via Web3, calls the running geth node, and geth grabs and return these two state variables from the contract's storage. Even if we had used geth as a node to `mainnet`, a sync node can get this info localy.

These are the same public variables that you declared in the `Donator` Solidity source:

```
contract Donator {

    uint public donationsTotal;
    uint public donationsUsd;
    uint public donationsCount;
    uint public defaultUsdRate;

    ...
}
```

Finally, we can do some client side processing.

Run the script:

```
$ python scripts/donator.py

Donator address on horton is 0xb8d9d2afbe18fd6ac43042164ece9691eb9288ed
The contract is already deployed on the chain
Total of 0.00 Ether accepted in 0 donations, an avergage of 0.00 Wei per donation.
```

Note that we don't need an expensive state variable for "average", in the contract, nor a function to calculate average. The contract just keeps only what can't be done elsewhere, to save gas. Moreover, code on deployed contracts can't be changed, so offloading code to the client gives you a lot of flexibility (and, again, gas, if you need a fix and re-deploy).

Send a Transaction to an Instance Function

To change the *state* of the instance, ether balance and the state variables, you need to send a transaction.

Once the transaction is picked by a miner, included in a block and accepted by the blockchain, every node on the blockchain will run and update the state of your contract. This process obviously costs real money, the gas.

With Populus and Web3 you send transactions with the `transact` function. For every contract instance object, `transact()` exposes the contract's instance functions. Behind the scenes, Populus takes your Pythonic call and, via Web3, convert it to the transactions' `data` payload, then sends the transaction to geth.

When geth get the transaction, it sends it to the blockchain. Populus will return the transaction hash. and you will have to wait until it's mined and accepted in a block. Typically 1-2 seconds with a local chain, but will take more time on `testnet` and `mainnet` (you will watch new blocks with `filters` and `events`,later on that).

We will add a transaction to the script:

```
$ nano scripts/donator.py
```

Update the script:

```
import random
from populus.project import Project

p = Project(project_dir="/home/mary/projects/donations/")
with p.get_chain('horton') as chain:
    donator, deploy_tx_hash = chain.provider.get_or_deploy_contract('Donator')

print("Donator address on horton is {address}".format(address=donator.address))
if deploy_tx_hash is None:
    print("The contract is already deployed on the chain")
else:
    print("Deploy Transaction {tx}".format(tx=deploy_tx_hash))
```

```
# Get contract state with calls
donationsCount = donator.call().donationsCount()
donationsTotal = donator.call().donationsTotal()

# Client side
ONE_ETH_IN_WEI = 10**18  # 1 ETH == 1,000,000,000,000,000,000 Wei
total_ether = donationsTotal/ONE_ETH_IN_WEI
avg_donation = donationsTotal/donationsCount if donationsCount > 0 else 0
status_msg = (
    "Total of {:,.2f} Ether accepted in {:,} donations, "
    "an avergage of {:,.2f} Wei per donation."
    )

print (status_msg.format(total_ether, donationsCount, avg_donation))

# Donate
donation = ONE_ETH_IN_WEI * random.randint(1,10)
effective_eth_usd_rate = 5
transaction = {'value':donation, 'from':chain.web3.eth.coinbase}
tx_hash = donator.transact(transaction).donate(effective_eth_usd_rate)
print ("Thank you for the donation! Tx hash {tx}".format(tx=tx_hash))
```

The transaction is a simple Python dictionary:

```
transaction = {'value':donation, 'from':chain.web3.eth.coinbase}
```

The `value` is obviously the amount you send *in Wei*, and the `from` is the account that sends the transaction.

Note: You can include any of the ethereum allowed items in a transaction except `data` which is created auto by converting the Python call to an EVM call. Web3 also set 'gas' and 'gasPrice' for you based on estimates if you didn't provide any. The 'to' field, the instance address, is already known to Populus for project-deployed contracts. See transaction parameters

---

Coinbase Account

Until now you didn't provide any account, because in the tests the `tester` chain magically creates and unlocks ad-hoc accounts. With a *persistent* chain you have to explictly provide the account.

Luckily, when Populus created the local `horton` chain it also created a default wallet file, a password file that unlocks the wallet, and included the `--unlock` and `--password` arguments for geth in the run script, `run_chain.sh`. When you run `horton` with `chains/horton/./run_chain.sh` the account is already unlocked.

All you have to do is to say that you want this account as the transaction account:

```
'from':chain.web3.eth.coinbase
```

The `coinbase` (also called `etherbase`) is the default account that geth will use. You can have as many accounts as you want, and set *one* of them as a coinbase. If you didn't add an account for `horton`, then the chain has only one account, the one that Populus created, and it's automatically assigned as the coinbase.

Note: The wallet files are saved in the chain's `keystore` directory. For more see the tutorial on *Wallets* and *Accounts*. For a more in-depth discussion see geth accounts managment

---

Finally, the script sends the transaction with `transact`:

```
tx_hash = donator.transact(transaction).donate(effective_eth_usd_rate)
```

Ok. Run the script, after you make sure that `horton` is running:

```
$ python scripts/donator.py

Donator address on horton is 0xb8d9d2afbe18fd6ac43042164ece9691eb9288ed
The contract is already deployed on the chain
Total of 0.00 Ether accepted in 0 donations, an avergage of 0.00 Ether per donation.
Thank you for the donation! Tx hash␣
↪0xbe9d182a508ec3a7efc3ada8cfb134647b39feec4a7eb018ef91cc38e216ddbc
```

Worked. The transaction was sent, yet we still don't see it. Run again:

```
$ python scripts/donator.py

Donator address on horton is 0xb8d9d2afbe18fd6ac43042164ece9691eb9288ed
The contract is already deployed on the chain
Total of 3.00 Ether accepted in 1 donations, an avergage of 3,000,000,000,000,000,000.
↪00 Wei per donation.
Thank you for the donation! Tx hash␣
↪0xf6d40adfedf1882e7543c4ef96803bd790127afdc67e40a4c7d91d29884ad182
```

First donation accepted! Run again:

```
$ python scripts/donator.py

Donator address on horton is 0xb8d9d2afbe18fd6ac43042164ece9691eb9288ed
The contract is already deployed on the chain
Total of 4.00 Ether accepted in 2 donations, an avergage of 2,000,000,000,000,000,000.
↪00 Wei per donation.
Thank you for the donation! Tx hash␣
↪0x21bd87b9db76b54a48c5a12a4bf7930a0e45480f5af5d0745cb2e8b4a438c5af
```

And they just keep coming.

If you looked at your geth chain terminal windown, you could see how geth picks the transaction and mine it:

```
INFO [10-20|01:48:32]  mined potential block                  number=3918␣
↪hash=d36ecd...e724c1
INFO [10-20|01:48:32] Commit new mining work                   number=3919 txs=0␣
↪uncles=0 elapsed=1.084ms
INFO [10-20|01:48:40] Submitted transaction                    ␣
↪fullhash=0xbe9d182a508ec3a7efc3ada8cfb134647b39feec4a7eb018ef91cc38e216ddbc␣
↪recipient=0xb8d9d2afbe18fd6ac43042164ece9691eb9288ed
INFO [10-20|01:49:05] Successfully sealed new block            number=3919␣
↪hash=4e36eb...01e41f
INFO [10-20|01:49:05]  mined potential block                  number=3919␣
↪hash=4e36eb...01e41f
INFO [10-20|01:49:05] Commit new mining work                   number=3920 txs=1␣
↪uncles=0 elapsed=735.282µs
INFO [10-20|01:49:21] Successfully sealed new block
```

Check the persistancy of the instance again. Stop the `horton` chain, press Ctrl+C in it's terminal window, and then re-run it with `chains/horton/./run_chain.sh`.

Run the script again:

```
$ python scripts/donator.py

Donator address on horton is 0xb8d9d2afbe18fd6ac43042164ece9691eb9288ed
The contract is already deployed on the chain
Total of 7.00 Ether accepted in 3 donations, an avergage of 2,333,333,333,333,333,504.
↪00 Wei per donation.
Thank you for the donation! Tx hash␣
↪0x8a595949271f17a2a57a8b2f37f409fb1ee809c209bcbcf513706afdee922323
```

Oh, it's so easy to donate when a genesis block allocates you billion something.

The contract instance *is* persistent, and the state is saved. With `horton`, a local chain, it's saved to your hard-drive. On `mainent` and `testnet`, to the entire blockchain nodes network.

Note: You may have noticed that we didn't call the `fallback` function. Currently there is no builtin way to call the `fallback` from Populus. You can simply send a transaction to the contract instance's address, without any explicit function call. On transaction w/o a function call the EVM will call the `fallback`. Even better, write another named function that you can call and test from Populus, and let the `fallback` do one thing - call this function.

Programatically Access to a Contract Instance

The script is very simple, but it gives a glimpse how to use Populus as bridge between your Python application and the Ethereum Blockchain. As an excercise, update the script so it prompts for donation amount, or work with the `Donator` instance on the *morty* local chain.

This is another point that you'll appreciate Populus: not only it helps to manage, develop and test blockchain assets (Solidity sources, compiled data, deployments etc), but it also exposes your blockchain assets as Python objects that you can later use *natively* in any of your Python projects. For more see #TODO Populus API.

Interim Summary

- You interacted with an Ethereum persistent contract instance on a local chain
- You used `call` to invoke the instance (no state change)
- You sent transactions to the instance (state changed)
- You used the `Project` object as an entry point to Populus' API for a simple Python script
- And, boy, you just donated a very generous amount of Wei.

Part 8: Web3.py Console

Geth Console in Python

You have probably stumbled already with the term "geth console". Geth exposes a javascript console with the Web3 javascript API, which is handy when you need an interactive command-line access to geth and the running node. It's also a great way to tinker with the blockchain.

Good news: you have identical interactive console with Python. All you have to do is to initiate a Web3.py connection in a Python shell, and from that point forward you have the exact same Web3 interface in Python.

We will show here only a few examples to get you going. Web3 is a comprehensive and rich API to the Ethereum platform, and you probably want to familiarise yourself with it.

Almost any Web3.js javascript API has a *Pythonic* counterpart in Web3.py.

See the the Web3.py documentation and the full list of 'web3.js JavaScript API.

Web3.py Connection to Geth

Start with the `horton` local chain. Run the chain:

```
$ chains/horton/./run_chain.sh
```

Start a Python shell:

```
$ python
```

Note: You may need `$ python3`

Initiate a Web3 object:

```
>>> from web3 import Web3, IPCProvider
>>> w3 = Web3(IPCProvider(ipc_path="/home/mary/projects/donations/chains/horton/chain_
↪data/geth.ipc"))
>>> w3
>>> <web3.main.Web3 object at 0x7f29dcc7c048>
```

That's it. You now have a full Web3 API access in Python.

Note: Use the actual path to the `horton` ipc_path. If you are not sure, look at the run file argument at `chains/horton/run_chain.sh`.

Sidenote, this is a good example why geth uses IPC (inter-process communication). It allows other *local* processes to access the running node. Web3, as another process, hooks to this IPC endpoint.

Console Interaction with a Contract Instance

We can interact with a contract instance via the Python shell as well. We will do things in a bit convuluted manual way here, for the sake of demostration. During the regular development process, Populus does all that for you.

To get a handle to the `Donator` instance on `horton` we need (a) it's **address**, where the bytecode sits, and (b) the **ABI** ,application binary interface, the detailed description of the functions and arguments of the contract interface. With the ABI the EVM knows how to call the compiled bytecode.

> **Warning:** A reminder, even when you call a contract without an ABI, or just send it Ether, you may still invoke code execution. The EVM will call the *fallback function*, if it exists in the contract

First, the address. In another terminal window:

```
$ cat registrar.json
```

The *Registrar* is where Populus holds the deployment details:

```
    {
  "deployments": {
    "blockchain://927b61e39ed1e14a6e8e8b3d166044737babbadda3fa704b8ca860376fe3e90b/
↪block/2e9002f82cc4c834369039b87916be541feb4e2ff49036cafa95a23b45ecce73": {
      "Donator": "0xcffb2715ead1e0278995cdd6d1736a60ff50c6a5"
    },
    "blockchain://c77836f10cb9691c430638647b95701568ace603d0876ff41c6f0b61218254b4/
↪block/34f52122cf90aa2ad90bbab34e7ff23bb8619d4abb2d8e66c52806ec9b992986": {
      "Greeter": "0xc5697df77a7f35dd1eb643fc2826c79d95b0bd76"
    },
    "blockchain://c77836f10cb9691c430638647b95701568ace603d0876ff41c6f0b61218254b4/
↪block/667aa2e5f0dea4087b645a9287efa181cf6dad4ed96516b63aefb7ef5c4b1dff": {
      "Donator": "0xb8d9d2afbe18fd6ac43042164ece9691eb9288ed"
    }
  }
}
```

It's hard to tell which blockchain is `horton`. Populus encodes a blockchain signature by the *hash* of it's block 0. We only see that `Donator` is deployed on two blockchains.

But since `Greeter` was deployed only on `horton`, the blockchain with two deployments is `horton`, and the other one is `morty`. So we can tell that `Donator` address on `horton` is `"0xb8d9d2afbe18fd6ac43042164ece9691eb9288ed"`.

Copy the actual address from your registrar file. Back in the python shell terminal, paste it:

```
>>> address = "0xb8d9d2afbe18fd6ac43042164ece9691eb9288ed"
```

Now we need the ABI. Go to the other terminal window:

```
$ solc --abi contract/Donator.sol
======= contracts/Donator.sol:Donator =======
Contract JSON ABI
[{"constant":true,"inputs":[],"name":"donationsCount","outputs":[{"name":"","type":
↪"uint256"}],"payable":false,"type":"function"},{"constant":true,"inputs":[],"name":
↪"donationsUsd","outputs":[{"name":"","type":"uint256"}],"payable":false,"type":
↪"function"},{"constant":true,"inputs":[],"name":"defaultUsdRate","outputs":[{"name":
↪"","type":"uint256"}],"payable":false,"type":"function"},{"constant":true,"inputs
↪":[],"name":"donationsTotal","outputs":[{"name":"","type":"uint256"}],"payable
↪":false,"type":"function"},{"constant":false,"inputs":[{"name":"usd_rate","type":
↪"uint256"}],"name":"donate","outputs":[],"payable":true,"type":"function"},{"inputs
":[],"payable":false,"type":"constructor"},{"payable":true,"type":"fallback"}]
```

Copy only the long list `[{"constant":true,"inputs":[]....]`, get back to the python shell, and paste the abi inside single quotes, like `'[...]'` as follows:

```
>>> abi_js = '[{"constant":true,"inputs":[],"name":"donationsCount","outputs":[{"name
↪":"","type":"uint256"}],"payable":false,"type":"function"},{"constant":true,"inputs
↪":[],"name":"donationsUsd","outputs":[{"name":"","type":"uint256"}],"payable":false,
↪"type":"function"},{"constant":true,"inputs":[],"name":"defaultUsdRate","outputs":[{
↪"name":"","type":"uint256"}],"payable":false,"type":"function"},{"constant":true,
↪"inputs":[],"name":"donationsTotal","outputs":[{"name":"","type":"uint256"}],
↪"payable":false,"type":"function"},{"constant":false,"inputs":[{"name":"usd_rate",
↪"type":"uint256"}],"name":"donate","outputs":[],"payable":true,"type":"function"},{
↪"inputs":[],"payable":false,"type":"constructor"},{"payable":true,"type":"fallback"}
↪]'
```

Your python shell should look like this:

```
>>> from web3 import Web3, IPCProvider
>>> w3 = Web3(IPCProvider(ipc_path="/home/mary/projects/donations/chains/horton/chain_
↪data/geth.ipc"))
>>> w3
>>> <web3.main.Web3 object at 0x7f29dcc7c048>
>>> address = "0xb8d9d2afbe18fd6ac43042164ece9691eb9288ed"
>>> abi_js = '[{"constant":true,"inputs":[],"name":"donationsCount","outputs":[{"name
↪":"","type":"uint256"}],"payable":false,"type":"function"},{"constant":true,"inputs
↪":[],"name":"donationsUsd","outputs":[{"name":"","type":"uint256"}],"payable":false,
↪"type":"function"},{"constant":true,"inputs":[],"name":"defaultUsdRate","outputs":[{
↪"name":"","type":"uint256"}],"payable":false,"type":"function"},{"constant":true,
↪"inputs":[],"name":"donationsTotal","outputs":[{"name":"","type":"uint256"}],
↪"payable":false,"type":"function"},{"constant":false,"inputs":[{"name":"usd_rate",
↪"type":"uint256"}],"name":"donate","outputs":[],"payable":true,"type":"function"},{
↪"inputs":[],"payable":false,"type":"constructor"},{"payable":true,"type":"fallback"}
↪]'
```

From now on, we will stay in the python shell.

Solc produced the ABI in JSON. Convert it to Python:

```
>>> import json
>>> abi = json.loads(abi_js)
```

Ready to instanciate a contract *object*:

```
>>> donator = w3.eth.contract(address=address,abi=abi)
>>> donator
<web3.contract.Contract object at 0x7f3b285245f8>
```

You now have the familiar `donator` Python object, with Python methods, that corresponds to a deployed contract instance bytecode on a blockchain.

Let's verify it:

```
>>> donator.call().donationsCount()
4
>>> donator.call().donationsTotal()
8000000000000000000
```

Works.

Btw, everything you did so far you can do in Populus with *one* line of code:

```
donator, deploy_tx_hash = chain.provider.get_or_deploy_contract('Donator')
```

When you work with Populus, you don't have to mess with the ABI. Populus saves the ABI with other important compilation info at `build/contracts.json`, and grabs it when required, gets the Web3 handle to the chain, creates the contract object and returns it to you (and if the contract is not deployed, it will deploy it).

Warning: Worth to remind again. **Never** call a contract with just an address and an ABI. You never know what the code at that address does behind the ABI. The only safe way is either if you absolutly know and trust the author, or to check it yourself. Get the source code from the author, make sure the source is safe, then compile it yourself, and verify that the compiled bytecode on your side is **exactly the same** as the bytecode at the said address on the blockchain.

Note that `donationsTotal` is *not* the account balance as it is saved in the *blockchain* state. It's rather a *contract* state. If we made a calculation mistake with `donationsTotal` then it won't reflect the actual Ether balance of the contract.

So all these doantions are not in the balance? Let's see:

```
>>> w3.eth.getBalacne(donator.address)
8000000000000000000
```

Phew. It's OK. The `donationsTotal` is exactly the same as the "official" balance.

And in Ether:

```
>>> w3.fromWei(w3.eth.getBalance(donator.address),'ether')
Decimal('8')
```

How much is left in your coinbase account on `horton`?

```
>>> w3.fromWei(w3.eth.getBalance(w3.eth.coinbase),'ether')
Decimal('1000000026682')
```

Oh. This is one of those accounts where it's fun to check the balance, isn't it?

So much Ether! why not donate some? Prepare a transaction:

```
>>> transaction = {'value':5*(10**18),'from':w3.eth.coinbase}
```

Only 5 Ether. Maybe next time. Reminder: The default unit is always Wei, and 1 Ether == 10 ** 18 Wei.

Send the transaction, assume the effective ETH/USD exchange rate is $7 per Ether:

```
>>> donator.transact(transaction).donate(7)
'0x86826ad2df93ffc6d6a6ac94dc112a66be2fff0453c7945f26bcaf20915058f9'
```

The hash is the *transaction's* hash. On local chains transactions are picked and mined in seconds, so we can expect to see the changed state almost immidiately:

```
>>> donator.call().donationsTotal()
13000000000000000000
>>> donator.call().defaultUsdRate()
7
>>> w3.fromWei(w3.eth.getBalance(donator.address),'ether')
Decimal('13')
```

You can also send a transaction *directly* to the chain, instead of via the `donator` contract object. It's a good opportunity, too, for a little more generousity. Maybe you go through the roof and donate 100 Ether!

```
>>> transaction = {'value':100*(10**18),'from':w3.eth.coinbase,'to':donator.address}
>>> w3.eth.sendTransaction(transaction)
>>> '0x395f5fdda0be89c803ba836e57a81920b41c39689ffefaaaaf6a30f532901bf5'
```

Check the state:

```
>>> donator.call().donationsTotal()
113000000000000000000
>>> donator.call().defaultUsdRate()
7
>>> w3.fromWei(w3.eth.getBalance(donator.address),'ether')
Decimal('113')
```

Now pause for a moment. What just happened here? The transaction you just sent didn't call the `donate` function at all. How did the donations total and balance increased? Take a look at the transaction again:

```
>>> transaction = {'value':100*(10**18),'from':w3.eth.coinbase,'to':donator.address}
```

No mention of the `donate` function, yet the 100 Ether were transfered and donated. How?

If you answered *fallback* you would be correct. The contract has a fallback function:

```
// fallback function
 function () payable {
   donate(defaultUsdRate);
 }
```

A *fallback function* is the one un-named function you can optionally include in a contract. If it exists, the EVM will call it when you just send Ether, without a function call. In `Donator`, the fallback just calls `donate` with the current `defaultUsdRate`. This is why the balance *did* increase by 100 Ether, but the ETH/USD rate didn't change (there are also other options to invoke the fallback).

Unlike a transaction, `call` doesn't change state:

```
>>> transaction = {'value':50,'from':w3.eth.coinbase}
>>>> donator.call(transaction).donate(10)
[]
>>> w3.fromWei(w3.eth.getBalance(donator.address),'ether')
>>> Decimal('113')
>>> donator.call().defaultUsdRate()
>>> 7
```

Console Interaction With Accounts

List of accounts:

```
>>> w3.eth.accounts
['0x66c91389b47bcc0bc6206ef345b889db05ca6ef2']
```

Go to another terminal, in the shell:

```
$ ls chains/horton/chain_data/keystore
UTC--2017-10-19T14-43-31.487534744Z--66c91389b47bcc0bc6206ef345b889db05ca6ef2
```

The account hash in the file name should be the *same* as the one you have on the Python shell. Web3 got the account from geth, and geth saves the accounts as wallet files in the ''keystore'' directory.

Note: The first part of the wallet file is a timestamp. See *Wallets*

Create a new account:

```
>>> w3.personal.newAccount()
Warning: Password input may be echoed.
Passphrase:demopassword
Warning: Password input may be echoed.
Repeat passphrase:demopassword

'0x7ddb35e66679cb9bdf5380bfa4a7f87684c418d0'
```

A new account was created. In another terminal, in a shell:

```
$ ls chains/horton/chain_data/keystore

UTC--2017-10-19T14-43-31.487534744Z--66c91389b47bcc0bc6206ef345b889db05ca6ef2
UTC--2017-10-21T14-08-01.257964745Z--7ddb35e66679cb9bdf5380bfa4a7f87684c418d0
```

The new wallet file was added to the chain keystore.

In the python shell:

```
>>> w3.eth.accounts
['0x66c91389b47bcc0bc6206ef345b889db05ca6ef2',
 ↪'0x7ddb35e66679cb9bdf5380bfa4a7f87684c418d0']
```

Unlock this new account:

```
>>> w3.personal.unlockAccount(account="0x7ddb35e66679cb9bdf5380bfa4a7f87684c418d0",
 ↪passphrase="demopassword")
True
```

Warning: Tinkering with accounts freely is great for development and testing. **Not** with real Ether. You should be extremely careful when you unlock an account with real Ether. Create new accounts with geth directly, so passwords don't appear in history. Use strong passwords, and the correct permissions. See *A Word of Caution*

Getting Info from the Blockchain

The Web3 API has many usefull calls to query and get info from the blockchain. All this information is publicly available, and there are many websites that present it with a GUI, like etherscan.io. The same info is available programmaticaly with Web3.

Quit the `horton` chain and start a new Python shell.

Mainnet with Infura.io

As an endpoint we will use `infura.io`. It's a publicly avaialble blockchain node, by Consensys, which is great for read-only queries.

Infura is a remote node, so you will use the `HTTPProvider`.

Note: Reminder: IPC, by design, allows only *local* processes to hook to the endpoint. Proccesses that run on the same machine. IPC is safer if you have to unlock an account, but for *read-only* queries remote HTTP is perfectly OK. Did we asked you already to look at *A Word of Caution*? We thought so.

Start a Web3 connection.

```
>>> from web3 import Web3,HTTPProvider
>>> mainent = Web3(HTTPProvider("https://mainnet.infura.io"))
>>> mainent
<web3.main.Web3 object at 0x7f7d3fb71fd0>
```

Nice. You have an access to the entire blockchain info at yor fingertips.

Get the last block:

```
>>> mainent.eth.blockNumber
4402735
```

Get the block itself:

```
>>> mainent.eth.getBlock(4402735)
AttributeDict({'mixHash': '0x83a49ac6843 ... })
```

The number of transactions that were included in this block:

```
>>> mainent.eth.getBlockTransactionCount(4402735)
58
```

The hash of the first transaction in this block:

```
>>> mainent.eth.getBlock(4402735)['transactions'][0]
'0x03d0012ed82a6f9beff945c9189908f732c2c01a71cef5c453a1c22da7f884e4'
```

This transaction details:

```
>>> mainent.eth.getTransaction(
↪'0x03d0012ed82a6f9beff945c9189908f732c2c01a71cef5c453a1c22da7f884e4')
AttributeDict({'transactionIndex': 0, 'to': '0x34f9f3a0e64ba ... })
```

Note: If you will use block number 4402735, you should get **exactly** the same output as shown above. This is the `mainent`, which is synced accross all the nodes, and every node will return the same info. The local chains `horton` or `morty` run a private instance, so every machine produces it's own blocks and hashes. Not so on the global, real blockchain, where all the nodes are synced (which is the crux of the whole blockchain idea).

Testnet with Infura.io

Web3 exposes the API to the testnet, just by using a different url:

```
>>> from web3 import Web3,HTTPProvider
>>> testnet = Web3(HTTPProvider("https://ropsten.infura.io"))
>>> testnet
```

```
<web3.main.Web3 object at 0x7ff597407d68>
>>> testnet.eth.blockNumber
1916242
```

Mainnet and Testnet with a Local Node

In order to use Web3.py to access the `mainnet` simply run geth on your machine:

```
$ geth
```

When geth starts, it provides *a lot* of info. Look for the line `IPC endpoint opened:`, and use this IPC endpoint path for the Web3 IPCProvider.

In a similar way, when you run:

```
$ geth --testnet
```

You will see another ipc path, and hooking to it will open a Web3 instance to the `testnet`.

Note: When you run geth for the first time, syncing can take time. The best way is to just to let geth run without interuption until it synced.

Note: Geth keeps the accounts in the `keystore` directory for each chain. If you want to use the same account, you will have to copy of import the wallet file. See Managing Accounts in geth

---

Interim Summary

- Interactive Web3 API in a Python shell
- Interacting with a contract instance in the Python shell
- Managing accounts in the Python shell
- Query the blockchain info on `mainnet` and `testnet` with HTTP from a remote node
- Query the blockchain info on local geth nodes.

Although Populus does a lot of work behind the scenes, it's recommended to have a good grasp of Web3. See the the Web3.py documentation and the full list of 'web3.js JavaScript API. Most of the javascript API have a Python equivalent.

Part 9: Withdraw Money from a Contract

Well, time to get out some of the donations, for a good cause!

Available Balance

First, we will check the balanc. The balance of an Ethereum account is saved as a blockchain status for an *address*, wether that address has a contract or not.

In addition to the "official" balance, the contract manages a `total_donations` state variable that should be the same.

We will query both. Add the following script:

```
$ nano scripts/donator_balance.py
```

The script should look as follows:

```
from populus.project import Project

p = Project(project_dir="/home/mary/projects/donations/")
with p.get_chain('horton') as chain:
    donator, _ = chain.provider.get_or_deploy_contract('Donator')

# state variable
donationsTotal = donator.call().donationsTotal()

# access to Web3 with the chain web3 property
w3 = chain.web3

# the account balance as saved by the blockchain
donator_balance = w3.fromWei(w3.eth.getBalance(donator.address),'ether')

print (donationsTotal)
print (donator_balance)
```

The script is very similar to what you already done so far. It starts with a `Project` object which is the main entry point to the Populus API. From the project we get the chain, and a contract object. Then the script grabs the `donationsTotal` with a `call`, no need for a transaction. Finally, with the Web3 API, we get the contract's balance as it is saved on the chain.

Both `donationsTotal` and the balance are in Wei. The web3 API `fromWei` converts it to Ether.

Two things to notice. A minor style change:

```
donator, _ = chain.provider.get_or_deploy_contract('Donator')
```

Instead of:

```
donator, deploy_tx_hash = chain.provider.get_or_deploy_contract('Donator')
```

We know that the contract is already deployed, so the return tuple from `get_or_deploy_contract` for an *already deployed* contract is `(contract_obj, None)`.

Second thing to notice, is that the Populus API gives you access the entire Web3 API, using the `chain.web3` property.

Run the script:

```
$ python scripts/donator_balance.py
113000000000000000000
113
```

Withdraw Funds from a Contract

Can you guess the correct way to *withdraw* Ether from `Donator`? You can dig a little into what you have done so far.

(we are waiting, it's OK)

Have an idea? Any suggestion will do.

(still waiting, np)

You don't have a clue how to withdraw the donations from the contract, do you?

It's OK. Neither do we.

The contract has **no** method to *withdraw* the Ether. If you, as the contract author, don't implement a way to withdraw funds or send them to another account, there is **no built in way to release the money**. The Ether is stucked on the contract balance forever. As far as the blockchain is concerned, those 113 Ether will remain in the balance of the `Donator` address, and you will not be able to use them.

Can you fix the code and redeploy the contract? Yes. But it will not release those 113 Ether. The new fixed contract will be deployed to a **new address**, an address with zero balance. The 113 lost Ether are tied to the **old address**. On the Ethereum blockchain, the smart contract's bytecode is tied to a *specific* address, and the funds that the contract holds are tied to the *same address*.

Unlike common software code, the smart contract is *stateful*. The code is saved with a state. And this state is synced to the entire network. The state can't be changed without a proper transaction, that is valid, mined, included in a block, and accepted by the network. Without a way to accept a transaction that releases funds, the `Donator` will just continue to hold these 113 Ether. In other words, they are lost.

Note: The blockchain "state" is not a physical property of nature. The state is a consensus among the majority of the nodes on the blockchain. If, theoreticaly, all the nodes decide to wipe out an account balance, they can do it. A single node can't, but the entire network can. It's unlikely to happen, but it's a theoretical possiblility you should be aware of. It happend once, after the DAO hack, where all the nodes agreed on a *hard fork*, a forced update of the blockchain state, which reverted the hack. See a good discussion of the issue on Quartz.

Withdraw Funds from a Contract, Take 2

Don't sweat those lost Ether. After all, what are 113 dummy Ethers out of a billion something Ether in your local `horton` chain. With the `horton` chain, you can absolutly afford it. And if it will prevent you from loosing real Ether on `mainent` in the future, then the cost/utility ratio of this lesson is excellent. Wish we could pay for more lessons with dummy Ether, if we were asked (but nobody is asking).

Anyway. Let's move on to a fixed contract with an option to withdraw the funds.

Create a new contract:

```
$ nano contracts/Donator2.sol
```

The new contract should look as follows:

```
pragma solidity ^0.4.11;

/// TUTORIAL CONTRACT DO NOT USE IN PRODUCTION
/// @title Donations collecting contract


contract Donator2 {
   uint public donationsTotal;
   uint public donationsUsd;
   uint public donationsCount;
   uint public defaultUsdRate;

    function Donator2() {
      defaultUsdRate = 350;
    }

    // fallback function
    function () payable {
        donate(defaultUsdRate);
    }

    modifier nonZeroValue() { if (!(msg.value > 0)) throw; _; }

    function donate(uint usd_rate) public payable nonZeroValue {
        donationsTotal += msg.value;
        donationsCount += 1;
        defaultUsdRate = usd_rate;
        uint inUsd = msg.value * usd_rate / 1 ether;
        donationsUsd += inUsd;
    }

    //demo only allows ANYONE to withdraw
    function withdrawAll() external {
        require(msg.sender.send(this.balance));
    }
}
```

Withdraw is handled in one simple function:

```
//demo only allows ANYONE to withdraw
function withdrawAll() external {
    require(msg.sender.send(this.balance));
      }
```

Anyone that calls this function will get the entire Ether in the contract to his or her own account. The contract sends it's remaining balance, `this.balance`, to the account address that sent the transaction, `msg.sender`.

The send is enclosed in a `require` clause, so if something failed everything is reverted.

Warning: This is a very naive way to handle money, only for the sake of demonstration. In the next chapter we will limit the withdrwal only to the contract owner. Usually contracts keep track of beneficiaries and the money they are allowed to withdraw.

Re-entry attack

When `Donator2` will run `send(this.balance)`, the beneficiary contract gets an opportunity to run it's `fallback` and get the execution control. In the fallback, it can call `Donator2` again before the `send` was line was completed, but the money already *sent*. This is a *re-entry* attack. To avoid it, any state changes should occur *before* the send.

```
//demo only allows ANYONE to withdraw
function withdrawAll() external {
    // update things here, before msg.sender gets control
    // if it re-enters, things already updated
    require(msg.sender.send(this.balance));
    // if you update things here, msg.sender get the money from the send
    // then call you, but things were not updated yet!
    // your contract state will not know that it's a re-entry
    // and the money was already sent
      }
```

To summarise, if you need to update state variables about sending money, do it *before* the send.

Deploy Donator2

Ok. Ready for deployment (probably much less mysterious by now):

```
$ chains/horton/./run_chain.sh

INFO [10-22|01:00:58] Starting peer-to-peer node
```

In another terminal:

```
$ populus compile
> Found 3 contract source files
  - contracts/Donator.sol
  - contracts/Donator2.sol
  - contracts/Greeter.sol
> Compiled 3 contracts
  - contracts/Donator.sol:Donator
  - contracts/Donator2.sol:Donator2
  - contracts/Greeter.sol:Greeter
> Wrote compiled assets to: build/contracts.json
```

Compilation passed. Deploy:

```
$ populus depoly --chain horton Donator2 --no-wait-for-sync

Donator2
Deploy Transaction Sent:␣
↪0xc34173d97bc6f4b34a630db578fb382020f092cc9e7fda20cf10e897faea3c7b
Waiting for confirmation...


Transaction Mined
=================
Tx Hash      : 0xc34173d97bc6f4b34a630db578fb382020f092cc9e7fda20cf10e897faea3c7b
Address      : 0xcb85ba30c0635872774e74159e6e7abff0227ac2
```

```
Gas Provided : 319968
Gas Used     : 219967
```

Deployed to `horton` at `0xcb85ba30c0635872774e74159e6e7abff0227ac2`.

Add a simple script that queries the `Donator2` instance on `horton`:

```
$ nano contracts/donator2_state.py
```

The script should look as follows:

```
from populus.project import Project

p = Project(project_dir="/home/mary/projects/donations/")
with p.get_chain('horton') as chain:
    donator2, _ = chain.provider.get_or_deploy_contract('Donator2')

donationsCount = donator2.call().donationsCount()
donationsTotal = donator2.call().donationsTotal()
donationsUsd = donator2.call().donationsTotal()
w3 = chain.web3
balance = w3.fromWei(w3.eth.getBalance(donator2.address),'ether')

print("donationsCount {:d}".format(donationsCount))
print("donationsTotal {:d}".format(donationsTotal))
print("donationsUsd {:d}".format(donationsUsd))
print("balance {:f}".format(balance))
```

Again, we use the Populus API to get a handle to the `Project`, and with a project object we can get the chain, the contract object, and the web3 connection.

Run the script:

```
$ python scripts/donator2_state.py

donationsCount 0
donationsTotal 0
donationsUsd 0
balance 0.000000
```

Nice new blank slate contract, with zero donations. Told you: those 113 Ether in `Donator` are lost

Add another script that donates 42 Ether to `Donator2`. To be precise, to the `Donator2` instance on `horton`:

```
$ nano scripts/donator2_send_42eth.py
```

And you could probably write the script yourself by now:

```
from populus.project import Project

p = Project(project_dir="/home/mary/projects/donations/")
with p.get_chain('horton') as chain:
    donator2, _ = chain.provider.get_or_deploy_contract('Donator2')

ONE_ETH_IN_WEI = 10**18
effective_eth_usd_rate = 5
transaction = {'value':42 * ONE_ETH_IN_WEI, 'from':chain.web3.eth.coinbase}
tx_hash = donator2.transact(transaction).donate(effective_eth_usd_rate)
print (tx_hash)
```

Save the script and run it 3 times:

```
$ python scripts/donator2_send_42eth.py
0xd3bbbd774bcb1cd72fb4b5823c71c5fe0b2efa84c5eeba4144464d95d810a353
$ python scripts/donator2_send_42eth.py
0xbc20f92b2940bdecb9aac7c181480647682218b552a7c96c4e72cf93b237160c
$ python scripts/donator2_send_42eth.py
0x43b99aa89af1f5596e5fa963d81a57bfe0c9da0100c9f4108540a67c57be0c93
```

Check state:

```
$ python scripts/donator2_state.py
donationsCount 0
donationsTotal 0
donationsUsd 0
balance 0.000000
```

Still nothing. Wait a few seconds, then try again:

```
$ python scripts/donator2_state.py
donationsCount 3
donationsTotal 126000000000000000000
donationsUsd 630
balance 126
```

Ok. All the three transactions where picked and mined by the chain.

A Transaction to Withdraw Ether to a New Account

Open a Python shell and create a new account:

```
>>> from populus.project import Project
>>> p = Project(project_dir="/home/mary/projects/donations/")
>>> with p.get_chain('horton') as chain:
...     donator2, _ = chain.provider.get_or_deploy_contract('Donator2c')
>>> w3 = chain.web3
>>> w3.personal.newAccount()
Warning: Password input may be echoed.
Passphrase:demopassword

Warning: Password input may be echoed.
Repeat passphrase:demopassword

'0xe4b83879df1194fede2a95555576bbd33142c244'
>>> new_account = '0xe4b83879df1194fede2a95555576bbd33142c244'
```

To withdraw money, the withdrawing account must send a transaction. If successful, this transaction will change the state of the blockchain: the contract's account sends Ether, another account recieves it.

The `'from'` key of the transaction will be this *new_account*, the withdrawer. Type:

```
>>> tx_withdraw = {'from':new_account}
```

Reminder. The following Solidity line in the contract will pick the sender, and tell the EVM to send the balance to the account that sent the transaction:

```
require(msg.sender.send(this.balance));
```

Send the transaction:

```
>>> donator2.transact(tx_withdraw).withdrawAll()

...
raise ValueError(response["error"])
builtins.ValueError: {'message': 'insufficient funds for gas * price + value', 'code
↪': -32000}
```

Right. The new account is obviously empty and doesn't have money for the gas:

```
>>> w3.eth.getBalance(new_account)
0
```

Transfer one Ether from your billion something `coinbase` account to the new account:

```
>>> w3.eth.sendTransaction({'from':w3.eth.coinbase,'to':new_account,'value':10**18})
'0x491f45c225e7ce22e8cf8289da392c4b34952101582b3b9c020d9ad5b6c61504'
>>> w3.eth.getBalance(new_account)
1000000000000000000
```

Great. Has more than enough Wei to pay for the gas.

Note: This is exactly why you used `--no-wait-for-sync` on deployments. When the account has funds to pay for the gas, you don't have to sync. But when you work with `mainnet` and your local node is not synced, it may think that the account is empty, although some transactions in further blocks did send the account money. Once the local node is synced to this block, geth can use it to pay for gas.

Send the withdraw transaction again:

```
>>> donator2.transact(tx_withdraw).withdrawAll()

...
raise ValueError(response["error"])
builtins.ValueError: {'message': 'authentication needed: password or unlock', 'code':␣
↪-32000}
```

Oops. Who said that withdrawing money is easy.

You created a new account but *didn't unlock* it. Geth can send transactions only with an *unlocked* account. It needs the unlocked account to sign the transaction with the account's private key, otherwise the miners can't ensure that the transaction was actually sent by the account that claims to send it.

Unlock the account:

```
>>> w3.personal.unlockAccount(new_account,passphrase="demopassword")
```

Warning: Again, extremely naive and unsafe way to unlock and use passwords. Use only for development and testing, with dummy Ether

The new account should be ready, it's unlocked, and has the funds for the gas.

Send the withdraw transaction yet *again*:

```
>>> donator2.transact(tx_withdraw).withdrawAll()
'0x27781b2b3a644b7a53681459081b998c42cfcf02d87d82c78dbb7d6119110521'
>>> w3.eth.getBalance(new_account)
126999489322000000000
```

Works. The geek shell inherit the earth.

Quit the Python shell, and check the contract's balance, or more precisely, the balance of the *address* of this contract *instance*:

```
$ python scripts/donator2_state.py
donationsCount 3
donationsTotal 126000000000000000000
donationsUsd 630
balance 0.000000
```

Correct. The balance is 0, yet `donationsTotal` that saves a running total of the *accepted* donations, shows all the 3 accepted donations of 42 Ether each.

Note: `donationsTotal` is a state variable that is saved in the *contract's* storage. The `balance` is the balance in Wei of the *address* of the contract, which is saved as part of the *blockchain's* status.

As an exercise, add some tests to test the `withdrawAll` functionality on `Donator2`.

Interim Summary

- If an author of a contract didn't implement a way to withdraw Ether, there is no builtin way to do it, and any money that was sent to this contract is lost forever
- Fixing a contract source code and re-deploying it saves the new bytecode to a *new* address, and does *not* and can *not* fix an existing contract instance on a previously deployed address
- You just created a new account, unlocked it, and withdrew money to it, with a transaction to a contract instance on a local chain
- You used the Web3 API via the Populus API

Part 10: Dependencies

So far we have worked with, and deployed, one contract at a time. However, Solidity allows you to inherit from other contracts, which is especially useful when you a more generic functionality in a basic core contract, a functionality you want to inherit and re-use in another, more specific use case.

A Contract that is Controlled by it's Owner

The `Donator2` contract is better than `Donator`, because it allows to withdraw the donations that the contract accepts. But not much better: at any given moment *anyone* can withdraw the entire donations balance, no questioned asked.

If you recall the withdraw function from the contract:

```
//demo only allows ANYONE to withdraw
function withdrawAll() external {
    require(msg.sender.send(this.balance));
      }
```

The `withdrawAll` function just sends the entire balane, `this.balance`, to whoever request it. Send everything to `msg.sender`, without any further verification.

A better implemntation would be to allow **only the owner of the contract to withdraw the funds**. An *owne*, in the Ethereum world, is an *account*. An *address*. The owner is the account that the contract creation transaction was sent from.

Restricting persmission to run some actions only to the owner is a very common design pattern. There are many open-source implemntations of this pattern, and we will work here with the OpenZeppelin library.

Note: When you adapt an open sourced Solidity contract from github, or elsewhere, be careful. You should trust only respected, known authors, and always review the code yourself. Smart contracts can induce some smart bugs, which can lead directly to loosing money. Yes, the Ethereum platform has many innovations, but loosing money from software bugs is not one of them. However, it's not fun to loose Ether like this. Be careful from intentional or unintentional bugs. See some really clever examples in Underhanded Solidity Coding Contest

---

This is the OpenZeppelin Ownable.sol Contract:

```
    pragma solidity ^0.4.11;


  /**
   * @title Ownable
   * @dev The Ownable contract has an owner address, and provides basic authorization␣
↪control
   * functions, this simplifies the implementation of "user permissions".
   */
  contract Ownable {
    address public owner;


    event OwnershipTransferred(address indexed previousOwner, address indexed␣
↪newOwner);


    /**
     * @dev The Ownable constructor sets the original `owner` of the contract to the␣
↪sender
     * account.
     */
    function Ownable() {
      owner = msg.sender;
    }
```

```
    /**
     * @dev Throws if called by any account other than the owner.
     */
    modifier onlyOwner() {
      require(msg.sender == owner);
      _;
    }


    /**
     * @dev Allows the current owner to transfer control of the contract to a␣
↪newOwner.
     * @param newOwner The address to transfer ownership to.
     */
    function transferOwnership(address newOwner) onlyOwner public {
      require(newOwner != address(0));
      OwnershipTransferred(owner, newOwner);
      owner = newOwner;
    }

  }
```

Save it to your conatracts directory:

```
$ nano contracts Ownable.sol
```

Inheritence & Imports

Create the new improved `Donator3`:

```
// TUTORIAL CONTRACT DO NOT USE IN PRODUCTION
/// @title Donations collecting contract
import "./Ownable.sol";


contract Donator3 is Ownable {
    uint public donationsTotal;
    uint public donationsUsd;
    uint public donationsCount;
    uint public defaultUsdRate;

    function Donator3() {
        defaultUsdRate = 350;
    }

    // fallback function
    function () payable {
        donate(defaultUsdRate);
    }

    modifier nonZeroValue() { if (!(msg.value > 0)) throw; _; }

    function donate(uint usd_rate) public payable nonZeroValue {
        donationsTotal += msg.value;
        donationsCount += 1;
```

```
        defaultUsdRate = usd_rate;
        uint inUsd = msg.value * usd_rate / 1 ether;
        donationsUsd += inUsd;
    }

    // only allows the owner to withdraw
    function withdrawAll() external onlyOwner {
        require(msg.sender.send(this.balance));
    }
}
```

Almost the same code of `Donator2`, with 3 important additions:

[1] An import statement: `import "./Ownable.sol" ``: Used to when you use constructs from other source files, a common practice in almost any programming language. The format of local path with ``./Filename.sol` is used for an import of a file in the same directory.

Note: Solidity supports quite comprehensive import options. See the Solidity documentation of Importing other Source Files

---

[2] Subclassing: `contract Donator3 is Ownable {...}`

[3] Use a parent member in the subclass:

```
function withdrawAll() external onlyOwner {
      require(msg.sender.send(this.balance));
        }
```

The `onlyOwner` modifier was *not* defined in `Donator3`, but it is inhereted, and thus can be used in the subclass.

Your contracts directory should look as follows:

```
$ ls contracts
Donator2.sol  Donator3.sol  Donator.sol  Greeter.sol  Ownable.sol
```

Compile the project:

```
$ populus compile
> Found 5 contract source files
- contracts/Donator.sol
- contracts/Donator2.sol
- contracts/Donator3.sol
- contracts/Greeter.sol
- contracts/Ownable.sol
> Compiled 5 contracts
- contracts/Donator.sol:Donator
- contracts/Donator2.sol:Donator2
- contracts/Donator3.sol:Donator3
- contracts/Greeter.sol:Greeter
- contracts/Ownable.sol:Ownable
```

Compilation works, `solc` successfuly found `Ownable.sol` and imported it.

Testing the Subclass Contract

The test is similar to a regular contract test. All the parents' memebers are inherited and available for testing (if a parent member was overidden, use `super` to access the parent member)

Add a test:

```
$ nano tests/test_donator3.py
```

The test should look as follows:

```
import pytest
from ethereum.tester import TransactionFailed

ONE_ETH_IN_WEI = 10**18

def test_ownership(chain):
    donator3, deploy_tx_hash = chain.provider.get_or_deploy_contract('Donator3')
    w3 = chain.web3
    owner = w3.eth.coinbase # alias

    # prep: set a second test account, unlocked, with Wei for gas
    password = "demopassword"
    non_owner =  w3.personal.newAccount(password=password)
    w3.personal.unlockAccount(non_owner,passphrase=password)
    w3.eth.sendTransaction({'value':ONE_ETH_IN_WEI,'to':non_owner,'from':w3.eth.
↪coinbase})

    # prep: initial contract balance
    donation = 42 * ONE_ETH_IN_WEI
    effective_usd_rate = 5
    transaction = {'value': donation, 'from':w3.eth.coinbase}
    donator3.transact(transaction).donate(effective_usd_rate)
    assert w3.eth.getBalance(donator3.address) == donation

    # test: non owner withdraw, should fail
    with pytest.raises(TransactionFailed):
        donator3.transact({'from':non_owner}).withdrawAll()
    assert w3.eth.getBalance(donator3.address) == donation

    # test: owner withdraw, ok
    donator3.transact({'from':owner}).withdrawAll()
    assert w3.eth.getBalance(donator3.address) == 0
```

The test is similar to the previous tests. Py.test and the Populus plugin provide a `chain` fixture, as an argument to the test function which is a handle to the `tester` ephemeral chain. `Donator3` is deployed, and the test function gets a contract object to `doantor3`.

Then the test creates a second account, `non_owner`, unlocks it, and send to this account 1 Ether to pay for the gas. Next, send 42 Ether to the contract.

Note: The test was deployed with the default account, the `coinbase`. So `coinbase`, or the alias `owner`, is the owner of the contract.

When the test tries to withdraw with `non_owner`, which is *not* the owner, the transaction fails:

```
# test: non owner withdraw, should fail
with pytest.raises(TransactionFailed):
    donator3.transact({'from':non_owner}).withdrawAll()
assert w3.eth.getBalance(donator3.address) == donation
```

When the owner tries to withdraw it works, and the balance is back to 0:

```
# test: owner withdraw, ok
donator3.transact({'from':owner}).withdrawAll()
assert w3.eth.getBalance(donator3.address) == 0
```

Run the test:

```
$ py.test --disable-pytest-warnings

=================================== test session starts ======================
platform linux -- Python 3.5.2, pytest-3.1.3, py-1.4.34, pluggy-0.4.0
rootdir: /home/mary/projects/donations, inifile: pytest.ini
plugins: populus-1.8.0, hypothesis-3.14.0
collected 5 items

tests/test_donator.py ....
tests/test_donator3.py .
========================== 5 passed, 24 warnings in 3.52 seconds ==============
```

Passed.

The 2nd test will test the‘‘Ownable‘‘ function that allows to transfer ownership. Only the current owner can run it. Let’s test it.

Edit the test file:

```
$ nano tests/test_donator3.py
```

And add the following test function:

```
def test_transfer_ownership(chain):
    donator3, deploy_tx_hash = chain.provider.get_or_deploy_contract('Donator4')
    w3 = chain.web3
    first_owner = w3.eth.coinbase # alias

    # set unlocked test accounts, with Wei for gas
    password = "demopassword"
    second_owner =  w3.personal.newAccount(password=password)
    w3.personal.unlockAccount(second_owner,passphrase=password)
    w3.eth.sendTransaction({'value':ONE_ETH_IN_WEI,'to':second_owner,'from':w3.eth.
↪coinbase})

    # initial contract balance
    donation = 42 * ONE_ETH_IN_WEI
    effective_usd_rate = 5
    transaction = {'value': donation, 'from':w3.eth.coinbase}
    donator3.transact(transaction).donate(effective_usd_rate)
    assert w3.eth.getBalance(donator3.address) == donation

    # test: transfer ownership
    assert donator3.call().owner == first_owner
    transaction = {'from':first_owner}
```

```
    donator3.transact(transaction).transferOwnership(second_owner)
    assert donator3.call().owner == second_owner

    # test: first owner withdraw, should fail after transfer ownership
    with pytest.raises(TransactionFailed):
        donator3.transact({'from':first_owner}).withdrawAll()
    assert w3.eth.getBalance(donator3.address) == donation

    # test: second owner withdraw, ok after transfer ownership
    donator3.transact({'from':second_owner}).withdrawAll()
    assert w3.eth.getBalance(donator3.address) == 0

    # test: transfer ownership by non owner, should fail
    transaction = {'from':first_owner}
    with pytest.raises(TransactionFailed):
        donator3.transact(transaction).transferOwnership(second_owner)
    assert donator3.call().owner == second_owner
```

Run the test:

```
$ py.test --disable-pytest-warnings
```

The test should fail:

```
          # transfer ownership
>       assert donator3.call().owner == first_owner
E       AssertionError: assert functools.partial(<function call_contract_function at␣
↪0x7f6245b39bf8>, <web3.contract.PopulusContract object at 0x7f624466e748>, 'owner',
↪{'to': '0xc305c901078781c232a2a521c2af7980f8385ee9'}) ==
↪'0x82a978b3f5962a5b0957d9ee9eef472ee55b42f1'
```

Yes. `donator3.call().owner` is wrong. Confusing. Reminder: `owner` should be accessed as a *function*, and *not* as a property, The compiler builds these functions for every public state variable.

Fix every occurence of `call().owner` to `call().owner()`. E.g., into:

```
assert donator3.call().owner() == first_owner
```

Then Run again:

```
$ py.test --disable-pytest-warnings
```

Fails again:

```
          # test: transfer ownership
>       assert donator3.call().owner() == first_owner
E       AssertionError: assert '0x82A978B3f5...f472EE55B42F1' == '0x82a978b3f59...
↪f472ee55b42f1'
E         - 0x82A978B3f5962A5b0957d9ee9eEf472EE55B42F1
E
```

In the Ethereum world, it does not matter if an address is uppercase or lowercase (capitalisation is used for checksum, to avoid errors in client applications). We will use all lower case.

Fix the test as follows:

```
def test_transfer_ownership(chain):
    donator3, deploy_tx_hash = chain.provider.get_or_deploy_contract('Donator4')
```

```
    w3 = chain.web3
    first_owner = w3.eth.coinbase # alias

    # set unlocked test accounts, with Wei for gas
    password = "demopassword"
    second_owner =  w3.personal.newAccount(password=password)
    w3.personal.unlockAccount(second_owner,passphrase=password)
    w3.eth.sendTransaction({'value':ONE_ETH_IN_WEI,'to':second_owner,'from':w3.eth.
↪coinbase})

    # initial contract balance
    donation = 42 * ONE_ETH_IN_WEI
    effective_usd_rate = 5
    transaction = {'value': donation, 'from':w3.eth.coinbase}
    donator3.transact(transaction).donate(effective_usd_rate)
    assert w3.eth.getBalance(donator3.address) == donation

    # test: transfer ownership
    assert donator3.call().owner().lower() == first_owner.lower()
    transaction = {'from':first_owner}
    donator3.transact(transaction).transferOwnership(second_owner)
    assert donator3.call().owner().lower() == second_owner.lower()

    # test: first owner withdraw, should fail after transfer ownership
    with pytest.raises(TransactionFailed):
        donator3.transact({'from':first_owner}).withdrawAll()
    assert w3.eth.getBalance(donator3.address) == donation

    # test: second owner withdraw, ok after transfer ownership
    donator3.transact({'from':second_owner}).withdrawAll()
    assert w3.eth.getBalance(donator3.address) == 0

    # test: transfer ownership by non owner, should fail
    transaction = {'from':first_owner}
    with pytest.raises(TransactionFailed):
        donator3.transact(transaction).transferOwnership(second_owner)
    assert donator3.call().owner().lower() == second_owner.lower()
```

Run the test:

```
$ py.test --disable-pytest-warnings

============================================== test session starts ==============
platform linux -- Python 3.5.2, pytest-3.1.3, py-1.4.34, pluggy-0.4.0
rootdir: /home/may/projects/donations, inifile: pytest.ini
plugins: populus-1.8.0, hypothesis-3.14.0
collected 6 items

tests/test_donator.py ....
tests/test_donator3.py ..

========================================== 6 passed, 29 warnings in 1.93 seconds ====
```

Ok, all the inherited members passed: The `Ownable` constructor that sets `owner` ran when you deployed it's subclass, `Donator3`. The parent modifier `onlyOwner` works as a modifier to a subclass function, and the `transferOwnership` parent's funcion can be called by clients via the subclass interface.

Note: If you will deploy `Donator3` to a local chain, say `horton`, and look at the `registrar.json`, you will not see an entry for `Ownable`. The reason is that although Solidity has a full complex multiple inheritence model (and `super`), the final result is once contract. Solidity just copies the inherited code to this contract.

Interim Summary

- You used an open sourced contract
- You imported one contract to another
- You added an ownership control to a contract
- You used inheritence and tested it

Part 11: Events

One of the things to get used to with Ethereum platform is the feedback loop. When you work with a common Python project on your local machine, functions return in no time. When you call a remote REST API, you response takes 1-2 seconds. When you send a transaction, it's status is "pending", and you will have to wait until a miner picks it, include it in a block, and the block is accepted by the blockchain. This can take a few minutes.

To track when a transaction is accepted, you use an `event`. An event is a way to to write an indexable log entry to the transaction reciept. Once the transaction is accepted, the receipt and the logs are included in the block as well, and you can watch every new block until the log entry you are looking for appears.

Part 12: API

Usage

Populus is a versatile tool, designed to help you from the moment you start do develop a smart contract, until it's working and integrated in any Python project. The core of Populus is a Pythonic interface and command line tools to the Ethereum platform.

The main areas you will use Populus are:

[1] Smart Contracts Development: manage and work with your blockchain assets, the Solidity source files, compilation data, deployments etc

[2] Testing: a testing framework with py.test, `tester` chains, and py.test fixtures

[3] Integration to any Python project and application: with the Pythonic API

So far we covered the contract development, deployments, and testing. We touched the API with a few simple scripts. In this part, we will cover the important classes of the API, and describe a few important members of each class.

Orientation

Typically your entry point to the API is a `Project` object. From the project object you get a chain object, and from a chain object you get a contract object. Then you can work with this contract.

The chain also has a `web3` property with the full Web3.py API.

Why do we need a `chain` abstraction at all? Because it enables the core Populus idea, to work with contracts in *every* state: the source files, the compiled data, and the deployed contract instances on one or more chains.

True, Solidity source files and compiled data are *not* chain related, only the deployed instances on a given chain are. But when you call a contract from a `chain`, Populus will either find the instance on that chain, or compiles and deploys a new instance. Similar code is used regardless of the contract's state. E.g., the same code is used when Populus needs to re-deploy on each test run with the `tester` chain, and when you interact with a presistent contract isnstance on a local chain or `mainnet`.

So with the `chain` object, you have one consistent interface, no matter what the underlying contract state is. See *what is a contract*

Project

class `populus.project.`**`Project`**

The `Project` object is the main entry point to the Populus API.

Existing Project:

```
from populus.project import Project
# the directory should have a project.json file
p = Project(project_dir='/home/mary/project/donations')
```

New Project:

```
from populus.project import Project
# will create a new project.json file
# will not create the default project structure you get with the command line populus␣
↪init
p = Project(project_dir='/home/mary/project/donations',create_config_file=True)
```

PopulusContract

class `populus.contracts.provider.`**`PopulusContract`**

A subclass of `web3.contract.Contract`. It is a Python object, with Python methods, that lets you interact with a corresponding contract instance on a blockchain.

Usually you will not instanciate it directly, but will get it from a contract factory. Populus keeps track of deployments, addresses, compiled data, abi, etc, and uses this info to create the `PopulusContract` for you.

Chain

class `populus.chain.base.`**`BaseChain`**

The `chain` object is a Python object that corresponds to a running blockchain.

Get the chain from a project object in a context manager:

```
# for a chain name as it appears in the config
with p.get_chain('chainname') as chain:
    # chain object available here inside the context manager
```

`chainname` is any chain that is defined either (a) in the project config file, `project.json`, or (b) in the user-scope config file at `~/.populus/config.json`.

In both files, the chain settings appears under the `chains` key.

Note: If the same chain name appears in both the project config and the user config, the project config name will overide the user-scope config

---

Chain Classes

class `populus.chain.external.`**`ExternalChain`**

A chain object over a running local instance of geth. The default chain when you don't use a chain for tests

class `populus.chain.tester.`**`TesterChain`**

An ephemeral chain that saves data to memory and resets on every run, great for testing (similar to a blank slate DB for each test run)

class `populus.chain.testrpc.`**`TestRPCChain`**

Local chain with RPC client, for fast RPC response in testing

Web3

Full Web3 API to the running chain

```
w3 = chain.web3
```

Provider

class `populus.contracts.provider.`**`Provider`**

The `Provider` object is the handle to a *contract factory*. It is capable of handling all the possible states of a contract, and using a contract factory, returns a `PopulusContract`.

To get a provider:

```
prv = chain.provider
```

`provider.`**`get_contract`**`(...)`

Returns: `PopulusContract`

Tries to find a contract in the registrar, if exist, will verify the bytecode and return a `PopulusContract`

Note: Currently matching bytecode is only by the current installed solc version

---

`provider.`**`get_or_deploy_contract`**`(...)`

Returns: `PopulusContract`

Perhaps the most powerful line in the Populus API

[1] If the contract's is *already* deployed, same as `get_contract`

[2] If the contract is *not* deployed, Populus will compile it, prepare a deployment transaction, calculte the gas estimate, send and wait for the deployment to a new address, verify the byte code, saves the deployment details to the reigstrar, and *then* create the Python contract object that corresponds to this address and return it.

`def get_contract_data ("contract_identifier")`

Returns a dictionary with the contract's data: abi, bytecode, etc.

Registrar

class `populus.contracts.registrar.`**`Registrar`**

A handler of contracts instances and addresses on chains.

`def set_contract_address(...)`

set a contract address in the registrar

`populus.contracts.registrar.`**`get_contract_addresses`**`(...)`

Retrieve a contract address in the registrar

Config

class `populus.config.base.`**`Config`**

The `Config` class is a "magic" object. It behaves like a dictionary, but knows how to unpack nested keys:

```
>>> from populus.project import Project
>>> p = Project('/home/mary/projects/donations')
>>> p.config
{'chains': {'web3http': {'web3': {'foo': 'baz'}, 'chain': {'class': ....
```

```
>>> p.config.keys()
('chains', 'web3', 'compilation', 'contracts', 'version')
>>> type(p.config)
<class 'populus.config.base.Config'>
>>> p.config.get('chains')
{'web3http': {'web3': {}, 'chain': {cts.backends.testing: 50}, 'ProjectContracts'....
>>> p.config.get('chains.web3http')
{'web3': {}, 'chain': {'class': 'populus.chain.web3provider.Web3HTTPProviderChain'}...
↪..
>>> p.config.get('chains.web3http.web3')
{}
>>> p.config['chains.web3http.web3'] = {'foo':'baz'}
>>> p.config.get('chains.web3http.web3')
{'foo': 'baz'}
>>> p.config.get('chains.web3http.web3.foo')
'baz'
```

Usually you don't initiate a `Config` object yourself, but use an existing object that Populus built from the configuration files. Then use common dictionary methods, which are implemented in the `Config` class.

`populus.config.base.`**`items`**

Retrieves the top level keys, so the `value` can be another nested config

`populus.config.base.`**`items`**(*flatten=True*)

Retrieves the full path.

```
>>> p.config.items()
(('chains', {'web3http': {'web3': {'foo': 'baz'}, 'chain': {'class': 'populus.chain.
↪web3provider ....
>>> p.config.items(flatten=True)
(('chains.horton.chain.class', 'populus.chain.ExternalChain'), ('chains ...
```

Populus Configs Usage

`proj_obj.`**`project_config`**
: The configuration loaded from the project local config file, `project.json`

`proj_obj.`**`user_config`**
: The configuration loaded from the user config file, `~/.populus/config.json`

`proj_obj.`**`config`**
: The merged `project_config` and `user_config`: when `project_config` and `user_config` has the *same* key, the `project_config` overides `user_config`, and the key value in the merged `project.config` will be that of `project_config`

`proj_obj.`**`get_chain_config`**(...)
: The chain configuration

`chain_obj.`**`get_web3_config`**
: The chain's Web3 configuration

`proj_obj.`**`reload_config`**
: Reloads configuration from `project.json` and `~/.populus/config.json`. You should instanciate the chain objects after reload.

Assignment & Persistency

Populus initial configuration is loaded from the JSON files.

You can customise the config keys in runtime, but these changes are *not* persisten and will *not* be saved. The next time Populus run, the configs will reset to `project.json` and `~/.populus/config.json`.

Assignment of simple values works like any dictionary:

```
project_obj.config["chains.my_tester.chain.class"] = "populus.chain.tester.TesterChain
↪"
```

However, since config is nested, you can assign a dictionary, or another config object, to a key:

```
project_obj.config["chains.my_tester.chain"] = {"class":"populus.chain.tester.
↪TesterChain"}
```

You can even keep a another separate configuration file, and replace the entire project config in runtime, e.g. for testing, running in different environments, etc:

```
from populus.config import load_config
proj_object.config = load_config("path/to/another/config.json")
```

Reset all changes back to the default:

```
proj_obj.reload_config()
```

You will have to re-instanciate chains after the reload.

Note: JSON files may seem odd if you are used to Python settings files (like django), but we think that for blockchain development, the external, static files are safer than a programmble Python module.

JSON References

There is a caveat: `config_obj['foo.baz']` may not return the same value is `config_obj.get('foo.baz')`. The reasone is that the configuration files are loaded as JSON schema, which allows `$ref$`. So if the config is:

```
{'foo':{'baz':{'$ref':'fawltyTowers'}}}
```

And in another place on the file you have:

```
'fawltyTowers':['Basil','Sybil','Polly','Manual']
```

Then:

```
>>> config_obj['foo.baz'] # doesn't solve $ref
{'$ref':'fawltyTowers'}
>>> config_obj.get('foo.baz') # solves $ref
['Basil','Sybil','Polly','Manual']
```

To avoid this, if you assign your own config_obj, use `config_obj.unref()`, which will solve all of the refernces.

Backends

Populus is plugable, using backend. The interface is defined in a base class, and a backend can overide or implement part or all this functionality.

E.g., the default backend for the `Registrar` is the `JSONFileBackend`, which saves the deployments details to a JSON file. But if you would need to save these details to RDBMS, you can write your own backend, and as long as it implements the `Registrar` functions (`set_contract_address`, `get_contract_addresses`) it will work.

Contracts backends:

class `populus.contracts.filesystem.`**`JSONFileBackend`**

`is_provider: False, is_registrar: True`

Saves registrar details to a JSON file

class `populus.contracts.filesystem.`**`MemoryBackend`**

`is_provider: False, is_registrar: True`

Saves registrar details to memory, in a simple dict variable

class `populus.contracts.project.`**`ProjectContractsBackend`**

`is_provider: True, is_registrar: False`

Gets the contracts data from the project source dirs

class `populus.contracts.testing.`**`TestContractsBackend`**

`is_provider: True, is_registrar: False`

Gets the contracts data from the project tests dir

1.11 Solidity and Smart Contracts Gotchas

Solidity itself is a fairly simple language, on purpose. However, the "mental model" of smart contracts development against the blockchain is unique. We compiled a (by no means complete) list of subtle issues which may be non-obvious, or even confusing at times, to what you expect from a "common" programming environment. Items are not sorted by priority.

This is all the fun, isn't it? So here is our TOP 10, no wait 62, issues.

[1] Everything that the contract **runs** on the blockchain, every calculation, costs money, the gas. There is a price for each action the EVM takes on behalf of your contract. Try to offload as much computations as you can to the client. E.g., suppose you want to calculate the average donation in a contract that collects money and counts donations. The contract should only provide the total donations, and the number of donations, then calculate the average in the client code.

[2] Everything the contract **stores** in it's persistent storage costs money, the gas. Try to minimise storage only to what absulutly positively is required for the contract to run. Data like derived calculations, caching, aggregates etc, should be handled on the client.

[3] Whenever possible, use **events and logs for persistent data**. The logs are not accessible to the contract code, and are static, so you can't use it for code execution on the block chain. But you can read the logs from the client, and they are much cheaper.

[4] The pricing of contract **storage** is **not linear**. There is a high initial cost to setting the storage to non zero (touching the storage). Never reset and reintiate the storage.

[5] Everything the contracts uses for temporary memory costs money, the gas. The pricing of using **memory**, the part that is cleared once execution done (think RAM), is not linear either, and cost per the same unit of usage increases sharply once you used a lot of memory. Try to avoid unreasonable memory usage.

[6] Even when you free storage, the gas you paid for that storage is **partially** refundble if. Don't use storage as a temporary store.

[7] Each time you **deploy** a contract, it costs money, the gas. So the habbit of pushing the whole codebase after every small commit, can cost a lot of money. When possible, try to breakdown the functionality to small focused contracts. If you fix one, you don't have to re-deploy the others. Use library contracts (see below). Removing a contrat is partially refundble, but less than deployment.

[8] No, sorry. Refunds will never exceed the gas provided in the transaction that initiated the refundble action. In fact, **refund is maxed to 50% of the gas** in that tansaction.

[9] Whenever you just send money to a contract (a transaction with value > 0), **even without calling any function**, you run the contract's code. The contract gets an opportunity to call other functions. Etheruem is different from Bitcoin: even simple money sending runs code (infact Bitcoin has a simple stack based scripts, but the common case is simple money transfers)

[10] Every contract can have one un-named function, the fallback function, which the contract **runs when it's called even if no specific function was named in the transaction**. This is the function you will hit when sending money in a transaction that just has value.

[11] If a contract has no fallback function, or it has one without the `payable` modifier, then a simple transaction that just sends the contract Ether, without calling any function, will fail.

[12] Contracts are saved on the blockchain in a compiled EVM (ethereum virtual machine) bytecode. There is **no way** to understand from the bytecode what the contract actually does. The only option to verify is to get the Solidity source that the author of the contract claims is, recompile the Source *with the same compiler version the contract on the blockchain was compiled*, and verify that this compilation matches the bytecode on the blockchain.

[13] Never **send Eth** to a contract unless you absolutely positively trust it's author, or you verified the bytecode against a source compilation yourself.

[14] Never **call** a contract, never call a function of a contract, unless you absolutely positively trust it's author, or you verified the bytecode against a source compilation yourself.

[15] If you loose the ABI, you will not be able to call the contract other than the fallback function. The situation is very similar to having a compiled executable without the source. When you compile with Populus, it saves the ABI in a project file. The ABI tells the EVM how to correctly call the compiled bytecode and pad the arguments. Without it, there is no way to do it. **Don't loose the ABI**.

[16] There is actually a bit convuluted way to call a contract without the ABI. With the address `call` method it's possible to call the fallback function, just provide the arguments. It's also an easy way to call a contract if the fallback is the main function you work with. If the first argument of the `call` is a `byte4` type, **this first argument is assumed to be a function signature**, and then arguments 2..n are given to this function (but not loosing the ABI is better). To call and forward the entire remaining gas to the callee contract use `addr.call.value(x)()`

[17] When a contract sends money to another contract, **the called contract gets execution control** and can call your caller *before* it returns, and before you updated your state variables. This is a *re-entry attack*. Therefore, after this second call, your contract runs agian while the state variables do *not* reflect the already sent money. In other words: the callee can get the money, then call the sender in a state that does not tell that money was sent. To avoid it, always update the state variables that hold the amount which another account is allowed to get *before* you send money, and revert if the transaction failed.

[18] Moreover, the called contract can run code or recursion that will exceed the max EVM stack depth of 1024, and **trigger exception which will bubble up to your calling contract**.

[19] Safer, and cheaper for you, to **allow withdrawal of money rather than sending it**. The gas will be paid by the beneficiary, and you will not have to transfer execution control to another account.

[20] Contracts are **stateful**. Once you send money to a contract, it's there. You can't reinstall, or redeploy (or restart, or patch, or fix a bug, or git pull... you get the idea). If you didn't provide a mechanism to withdraw the funds in the first place, it's lost. If you want to update the source of a deployed contract, you can't. If you want to deploy a new version and didn't provide a mechanism to send the money to the new version, you are stuck with the old one.

[21] `call` and `delegatecall` invoke other contracts, but can't catch exceptions in these contracts. The only indication that you will get if the call excepted, is when these functions return `false`. This implies that providing an address of non-existent contract to `call` and `delegatecall` will **invoke nothing but still no exception**. You get `true` for *both* successful run *and* calling non-existent contract. Check existence of a contract in the address, *before* the call.

[22] `delegatecall` is a powerful option, yet you have to be careful. It runs another contract code but **in the context of your calling contract**. The callee has access to your calling contract Eth, storage, and the entire gas of the transaction. It can consume all, and send the money away to another account. You can't hedge the call or limit the gas. Use `delegatecall` with care, typically only for library contracts that you absolutely trust.

[23] `call` on the client is a web3 option that behaves exactly like sending a real transaction, but it will **not change the blockchain state**. It's kinda dry-run transaction, which is great for testing (and it's not related at all to the Solidity `call`). `call` is also useful to get info from the current state, without changing it. Since no state is changed, it runs on the local node, saving expensive gas.

[24] **Trusted** contract libraries are a good way to save gas of repeating deployments, for code that is actually reusable.

[25] The EVM stack limit is 1024 deep. For deeply nested operations, prefer working in **steps and loops, over recursions**.

[26] Assigning variables between memory and storage **always creates a copy**, which is expensive. Also any assignment to a state variables. Avoid it when possible.

[27] Assigning a memory variable to a storage variable always creates a pointer, which will not be aware if the **underlying state variable changed**

[28] Don't use rounding for Eth in the contract, since **it will cost you the lost money that was rounded**. Use the very fine grained Eth units instead.

[29] The default money unit, both in Solidity and Web3, like `msg.value`, or getting the balance, is always **Wei**.

[30] As of solc 0.4.17 Solidity **does not have a workable decimal point type**, and your decimals will be casted to ints. If needed, you will have to run your own fixed point calculations (many times you can retrieve the int variables, and run the decimal calculation on the client)

[31] Once you unlock your acount in a running node, typically with geth, the running process has full access to your funds. Keep it safe. **Unlock an account only in a local, protected instance**.

[32] If you connect to a remote node with rpc, use it only for actions that do not require unlocking an account, such as reading logs, blocks data etc. **Don't unlock accounts in remote rpc nodes**, since anybody who manages to get access to the node via the internet can use the account funds.

[33] If you have to unlock an account to deploy contracts, send transactions, etc, keep in this account **only the minimum Eth you need** for these actions.

[34] Anybody who has the **private key** can drain the account funds, no questions asked.

[35] Anybody who has the **wallet encrypted file and it's password** can drain the account funds, no questions asked.

[36] If you use a password file to unlock the account, make sure the file is well protected with the **right permissions**.

[37] If you look at your acount in sites like etherscan.io and there are funds in the account, yet localy the account balance is 0 and geth refuses to run actions that require funds for gas - then **your local node is not synced**. You must sync until the block with the transactions that sent money to this account.

[38] Once the contract is on the blockchain, there is **no builtin way to shut it down** or block it from responding to messages. If the contract has a bug, an issue, a hack that let hackers steal funds, you can't shutdown, or go to "maintenance" mode, unless you provided a mechanism for that in the contract code beforehand.

[39] Unless you provided a function that kills the contract, there is **no builtin way to delete** it from the blockchain.

[40] Scope and visibility in Solidity are only in terms of the running code. When the EVM runs your contract's code, it does care for `public`, `external` or `internal`. The EVM doesn't use these keywords, but visibility is enforced in the bytecode and the exposed interface (this is not just a compiler hint). However, the scope visibility definitions have **no effect** on the information that the blockchain exposes to the outside world.

[41] If you don't explicity set a `payable` modifier to a function, it will **reject the Eth that was sent in the transaction**. If no function has `payable`, the contract can't accept Ether.

[42] This **is** the answer.

[43] It's **not** possible to get a list of all the `mapping` variable keys or values, like `mydict.keys()` or `mydict.values()` in Python. You'll have to handle such lists yourself, if required.

[44] The contract's Constructor runs only once **when the contract is created**, and can't be called again. The constructor is optional.

[45] Inheritence in Solidity is different. Usually you have a Class, a Subclass, each is an independent object you can access. In Solidity, the inheritance is more syntatic. In the final compilation the compiler **copies the parent class members**, to create the bytecode of the derived contract with the *copied* memebers. In this context, `private` is just a notion of state variables and functions that the compiler will *not* copy.

[46] Memory reads are limited to a width of 256 bits, while writes can be either 8 bits or 256 bits wide

[47] `throw` and `revert` terminate and **revert all** changes to the state and to Ether balances. The used gas is not refunded.

[48] `function` is a **legit variable type**, and can be passed as an argument to another function. If a function type variable is not initialized, calling it will obviously result in an exception.

[49] Mappings are only allowed for **state** variables

[50] `delete` does not actually deletes, but assigns the initial value. It's a special **kind of assignment** actually. Deleting a local `var` variable that points to a state variable will except, since the "deleted" variable (the pointer) has no initial value to reset to.

[51] Declared variables are implictly initiated to their **initial default** value at the begining of the function.

[52] You can declare a function as `constant`, or the new term `view`, which theoretaclly should declare a "safe" function that does not alter the state. Yet the compiler **does not enfore it.**

[53] `internal` functions can be called only from the contract itself.

[54] To access an `external` function `f` from within the same contract it was declared in, use `this.f`. In other cases you don't need `this` (*this* is kinda bonus, no?)

[55] `private` is important only if there are **derived contracts**, where `private` denotes the members that the compiler does not copy to the derived contracts. Otherwise, from within a contract, `private` is the same as `internal`.

[56] `external` is available only for functions. `public`, `internal` and `private` are available for both functions and state variables. The **contract's interface** is built from it's `external` and `public` memebers.

[57] The compiler will **automatically** generate an accessor ("get" function) for the `public` state variables.

[58] `now` is the time stamp of the **current block**

[59] **Ethereum units** `wei`, `finney`, `szabo` or `ether` are reserved words, and can be used in experessions and literals.

[60] **Time units** `seconds`, `minutes`, `hours`, `days`, `weeks` and `years`, are reserved words, and can be used in experessions and literals.

[61] There is **no type conversion from non-boolean** to boolean types. `if (1) { ... }` is not valid Solidity.

[62] The `msg`, `block` and `tx` variables always exist in the **global namespace**, and you can use them and their members without any prior decleration or assignment

Nice! You got here. Yes, we know. You want more. See *Writing Contracts*

1.12 Additional Resources

1.12.1 Introduction to Ethereum

- A 101 Noob Intro to Programming Smart Contracts on Ethereum, from Consensys
- A Gentle Intorduction to Ethereum, from bitsonblocks

1.12.2 How the Platform Works

- The Etherum Whitepaper, by Vitalik Buterin
- The Bitcoin Book (Some of the Bitcoin concepts are different in Ethereum, but in any case it's useful to understand the difference)
- Even more subtleties

1.12.3 Protecting Yout Ether

- Protecting yourself and your funds, from MyEtherWallet

1.12.4 Writing Contracts

- Solidity security considerations
- Solidity style guide
- Ethereum smart contract security, from OpenZeppelin
- Best smart contracts practices, from Consensys
- Writing robust smart contracts in Solidity ,from colony.io

1.13 Release Notes

1.13.1 2.0.0

- Drop gevent support.
- Change upstream dependency for `ethereum-utils` to new library name `eth-utils`

1.13.2 1.11.2

- Bugfix for running tests without explicitely declared project root dir.

1.13.3 1.11.1

- Bugfix for wait utils in `wait_for_block`.

1.13.4 1.11.0

- Update to confguration API to support both *project* level configuration files as well as *user* level configuration files.
- `--wait-for-sync` cli argument in `$ populus deploy` now defaults to `False` (previous defaulted to `True`)
- Deprecation of Go-Ethereum based `Chain` objects in preparation for upcoming *Environments* API which will replace the *Chain* API.
- New `$ populus chain new` command for initializing local development chains.
- Removal of `$ populus config set` and `$ populus config delete` CLI comands.

1.13.5 1.10.3

- Add deprecation warning for upcoming removal of python 2 support.

1.13.6 1.10.2

- Deprecate support for `gevent` based threading.

1.13.7 1.10.1

- Upstream bugfix for change to internal `web3.py` API.

1.13.8 1.10.0

- Support for specifying project directory via any of `pytest.ini`, `--populus-project`, or via environment variable `PYTEST_POPULUS_PROJECT`.
- Support for running populus commands from outside of the project directory with `-p/--project`.

- Deprecate support for `solc<0.4.11`.
- Deprecate `Project.write_config()` in preparation for configuration API refactors.
- Deprecate `$ populus config set` and `populus config delete` commands in preparatin for configuration API refactors.

1.13.9 1.9.1

- Bugfix for `Wait.wait_for_block` API when interacting with a tester chain.

1.13.10 1.9.0

- Bugfix for `SolcStandardJSONBacken` compilation. Settings for this compiler now appropriately split between standard JSON input and compiler command line arguments.
- Deprecate: `project.contracts_source_dir` and releated setting `compilation.contracts_source_dir`.
- New API: `project.contracts_source_dirs` replaces deprecated singular `project.contracts_source_dir`.
- New setting `compilation.contracts_source_dirs`.

1.13.11 1.8.1

- Add `--logging` option to main cli command to set logging level.

1.13.12 1.8.0

- Change default compiler backend to `populus.compilation.backends.SolcAutoBackend` which will automatically select the appropriate solc compiler backend based on the current installed version olf solc.
- Add support for standard JSON based solc compilation.
- Bugfix: Compilation now correctly respects import remappings.

1.13.13 1.7.0

- Remove deprecated `chain.contract_factories` API.
- Remove deprecated `chain.get_contract_factory` API.
- Remove deprecated `chain.is_contract_available` API.
- Remove deprecated `chain.get_contract` API.
- Remove deprecated `chain.deployed_contracts` API.
- Remove deprecated `contracts` pytest fixture.
- Remove deprecated `project.compiled_contracts_file_path` API
- Remove deprecated `project.contracts_dir` API
- Remove deprecated `project.build_dir` API
- Remove deprecated `project.compiled_contracts` API

- Remove deprecated `project.blockchains_dir` API
- Remove deprecated `project.get_blockahin_data_dir` API
- Remove deprecated `project.get_blockchain_chaindata_dir` API
- Remove deprecated `project.get_blockchain_dapp_dir` API
- Remove deprecated `project.get_blockchain_ipc_path` API
- Remove deprecated `project.get_blockchain_nodekey_path` API

1.13.14 1.6.9

- Bump py-geth version to account for removed `--ipcapi` CLI flag.

1.13.15 1.6.8

- Allow for empty passwords when unlocking accounts.

1.13.16 1.6.7

- Bugfix for registrar address sorting to handle nodes which were fast synced and do not have access to the full chain history.

1.13.17 1.6.6

- Add support to contract provider to handle case where registrar has more than one address for a given contract.

1.13.18 1.6.5

- Bugfix for compilation of abstract contracts.

1.13.19 1.6.4

- Bugfix for `project.config` setter function not setting correct value.

1.13.20 1.6.3

- Add `TestContractsBackend` for loading test contracts.

1.13.21 1.6.2

- Fix incorrect example test file from `$ populus init` command.

1.13.22 1.6.1

- Fix warning message for outdated config file so that it actually shows up in terminal.

1.13.23 1.6.0

- Introduce new Registrar API.
- Introduce new Provider API
- Deprecate `Chain.get_contract_factory`, `Chain.get_contract` and `Chain.is_contract_available` APIs.
- Deprecate `Chain.contract_factories` API.
- Deprecate `Chain.deployed_contracts` API.
- Remove deprecated migrations API.

1.13.24 1.5.3

- Bump `web3.py` version to pull in upstream fixes for `ethereum-abi-utils`

1.13.25 1.5.2

- Bugfix for remaining `web3.utils` imports

1.13.26 1.5.1

- Update upstream `web3.py` dependency.
- Switch to use `ethereum-utils` library.

1.13.27 1.5.0

- Remove gevent dependency
- Mark migrations API for deprecation.
- Mark unmigrated_chain testing fixture for deprecation.
- Mark `contracts` fixture for deprecation. Replaced by `base_contract_factories` fixture.
- Deprecate and remove old `populus.ini` configuration scheme.
- Add new configuration API.

1.13.28 1.4.2

- Upstream version bumps for web3 and ethtestrpc
- Change to use new web3.providers.tester.EthereumTesterProvider for test fixtures.

1.13.29 1.4.1

- Stop-gap fix for race-condition error from upstream: https://github.com/pipermerriam/web3.py/issues/80

1.13.30 1.4.0

- Contract source directory now configurable via populus.ini file.
- Updates to upstream dependencies.

1.13.31 1.3.0

- Bugfix for geth data_dir directory on linux systems.

1.13.32 1.2.2

- Support solc 0.4.x

1.13.33 1.2.1

- Support legacy JSON-RPC spec for `eth_getTransactionReceipt` in wait API.

1.13.34 1.2.0

- All function in the `chain.wait` api now take a `poll_interval` parameter which controls how aggressively they will poll for changes.
- The `project` fixture now caches the compiled contracts across test runs.

1.13.35 1.1.0

This release begins the first deprecation cycle for APIs which will be removed in future releases.

- Deprecated: Entire migrations API
- New configuration API which replaces the `populus.ini` based configuration.
- Removal of `gevent` as a required dependency. Threading and other asynchronous operations now default to standard library tools with the option to enable the gevent with an environment variable `THREADING_BACKEND==gevent`

1.13.36 1.0.0

This is the first release of populus that should be considered stable.

- Remove `$ populus web` command
- Remove `populus.solidity` module in favor of `py-solc` package for solidity compilation.
- Remove `populus.geth` module in favor of `py-geth` for running geth.
- Complete refactor of pytest fixtures.
- Switch to `web3.py` for all blockchain interactions.

- Compilation: - Remove filtering. Compilation now always compiles all contracts. - Compilation now runs with optimization turned on by default. Can be disabled with `--no-optimizie`. - Remove use of `./project-dir/libraries` directory. All contracts are now expected to reside in the `./project-dir/contracts` directory.
- New `populus.Project` API.
- New Migrations API: - `$ populus chain init` for initializing a chain with the Registrar contract. - `$ populus makemigration` for creating migration files. - `$ populus migrate` for executing migrations.
- New configuration API: - New commands `$ populus config`, `$ populus config:set` and `$ populus config:unset` for managing configuratino.
- New Chain API: - Simple programatic running of project chains. - Access to `web3.eth.contract` objects for all project contracts. - Access to pre-linked code based on previously deployed contracts.

1.13.37 0.8.0

- Removal of the `--logfile` command line argument. This is a breaking change as it will break when used with older installs of `geth`.

1.13.38 0.7.5

- Bugfix: `populus init` now creates the `libraries` directory
- Bugfix: `populus compile --watch` no longer fails if the `libraries` directory isn't present.

1.13.39 0.7.4

- Bugfix for the `geth_accounts` fixture.
- Bugfix for project initialization fixtures.
- Allow returning of `indexed` event data from Event.get_log_data
- Fix EthTesterClient handling of TransactionErrors to allow continued EVM interactions.
- Bugfix for long Unix socket paths.
- Enable whisper when running a geth instance.
- Better error output from compile errors.
- Testing bugfixes.

1.13.40 0.7.3

- Add `denoms` pytest fixture
- Add `accounts` pytest fixture
- Experimental synchronous function calls on contracts with `function.s(...)`
- Bugfixes for function group argument validation.
- Bugfixes for error handling within EthTesterClient
- Inclusion of Binary Runtime in compilation

- Fixes for tests that were dependent on specific solidity versions.

1.13.41 0.7.2

- Make the ethtester client work with asynchronous code.

1.13.42 0.7.1

- Adds `ipc_client` fixture.

1.13.43 0.7.0

- When a contract function call that is supposed to return data returns no data an error was thown. Now a custom exception is thrown. This is a breaking change as previously for addresses this would return the empty address.

1.13.44 0.6.6

- Actually fix the address bug.

1.13.45 0.6.5

- Fix bug where addresses were getting double prefixed with `0x`

1.13.46 0.6.3

- Bugfix for Event.get_log_data
- Add `get_code` and `get_accounts` methods to EthTesterClient
- Add `0x` prefixing to addresses returned by functions with multiple return values.

1.13.47 0.6.3

- Shorted path to cli tests to stay under 108 character limit for unix sockets.
- Adds tracking of contract addresses deployed to test chains.
- New `redeploy` feature available within `populus attach` as well as notification that your contracts have changed and may require redeployment.

1.13.48 0.6.2

- Shorted path to cli tests to stay under 108 character limit for unix sockets.
- Allow passing `--verbosity` tag into `populus chain run`
- Expand documentation with example use case for populus deploy/chain/attach commands.

1.13.49 0.6.1

- Change the *default* gas for transactions to be a percentage of the max gas.

1.13.50 0.6.0

- **Improve `populus deploy` command.**
  - Optional dry run to test chain
  - Prompts user for confirmation on production deployments.
  - Derives gas needs based on dry-run deployment.
- Addition of `deploy_coinbase` testing fixture.
- Renamed `Contract._meta.rpc_client` to be `Contract._meta.blockchain_client` to be more appropriately named since the `EthTesterClient` is not an RPC client.
- Renamed `rpc_client` argument to `blockchain_client` in all relevant functions.
- Moved `get_max_gas` function onto blockchain clients.
- Moved `wait_for_transaction` function onto blockchain clients.
- Moved `wait_for_block` function onto blockchain clients.
- Bugfix when decoding large integers.
- Reduced `gasLimit` on genesis block for test chains to `3141592`.
- Updated dependencies to newer versions.

1.13.51 0.5.4

- Additional support for *library* contracts which will be included in compilation.
- `deployed_contracts` automatically derives deployment order and dependencies as well as linking library addresses.
- `deployed_contracts` now comes with the transaction receipts for the deploying transaction attached.
- Change to use `pyethash` from pypi

1.13.52 0.5.3

- New `populus attach` command for launching interactive python repl with contracts and rpc client loaded into local scope.
- Support for auto-linking of library contracts for the `deployed_contracts` testing fixture.

1.13.53 0.5.2

- Rename `rpc_server` fixture to `testrpc_server`
- Introduce `populus_config` module level fixture which holds all of the default values for other populus module level fixtures that are configurable.

- Add new configuration options for `deployed_contracts` fixture to allow declaration of which contracts are deployed, dependency ordering and constructor args.
- Improve overall documentation around fixtures.

1.13.54 0.5.1

- Introduce the `ethtester_client` which has the same API as the eth_rpc_client.Client class but interacts directly with the `ethereum.tester` module
- Add ability to control the manner through which the `deployed_contracts` fixture communicates with the blockchain via the `deploy_client` fixture.
- Re-organization of the contracts module.
- Support for multiple contract functions with the same name.
- Basic support for extracting logs and log data from transactions.

1.13.55 0.5.0

- Significant refactor to the `Contract` and related `Function` and `Event` objects used to interact with contracts.
- Major improvements to robustness of `geth_node` fixture.
- `deployed_contracts` testing fixture no longer provides it's own rpc server. Now you must either provide you own, or use the `geth_node` or `rpc_server` alongside it in tests.
- `geth_node` fixture now writes to a logfile located in `./chains/<chain-name>/logs/` for both cli and test case runs.

1.13.56 0.4.3

- Add support for address function args with a 0x prefix.

1.13.57 0.4.2

- Add `init` command for initializing a populus project.

1.13.58 0.4.1

- Missing `index.html` file.

1.13.59 0.4.0

- **Add blockchain management via `populus chain` commands which wraps `geth` library.**
  - `populus chain run <name>` for running the chain
  - `populus chain reset <name>` for resetting a chain
- **Add html/css/js development support.**
  - Development webserver via `populus web runserver`

  - Conversion of compiled contracts to web3 contract objects in javascript.

1.13.60 0.3.7

- Add support for decoding multiple values from a solidity function call.

1.13.61 0.3.6

- Add support for decoding `address`` return types from contract functions.

1.13.62 0.3.5

- Add support for contract constructors which take arguments via the new `constructor_args` parameter to the `Contract.deploy` method.

1.13.63 0.3.4

- Fix bug where null bytes were excluded from the returned bytes.

1.13.64 0.3.3

- Fix a bug in the `sendTransaction` methods for contract functions that did not pass along most of the `**kwargs`.
- Add new `Contract.get_balance()` method to contracts.

1.13.65 0.3.2

- Enable decoding of `bytes` types returned by contract function calls.

1.13.66 0.3.1

- Enable decoding of `boolean` values returned by contract function calls.

1.13.67 0.3.0

- Removed `watch` command in favor of passing `--watch` into the `compile` command.
- Add granular control to the `compile` command so that you can specify specific files, contract names, or a combination of the two.

1.13.68 0.2.0

- Update to `pypi` version of `eth-testrpc`
- Add new watch command which observes the project contracts and recompiles them when they change.
- Improved shell output for compile command.
- Re-organized portions of the `utils` module into a new `compilation` module.

1.13.69 0.1.4

- Fix broken import in `cli` module.

1.13.70 0.1.3

- Remove the local RPC client in favor of using https://github.com/pipermerriam/ethereum-rpc-client

1.13.71 0.1.2

- Add missing pytest dependency.

1.13.72 0.1.1

- Fix bug when deploying contracts onto a real blockchain.

1.13.73 0.1.0

- Project Creation

1.14 Viper Support

Populus has support for the viper compiler (a python-like experimental programming language).

1.14.1 Installation

To install the viper compiler:

You will see the *viper* binary is now installed.

```
$ viper
usage: viper [-h] [-f {abi,json,bytecode,bytecode_runtime,ir}]
             [--show-gas-estimates]
             input_file
viper: error: the following arguments are required: input_file
```

To use viper as you compiler backend you have to configure you *project.json* file to support viper, this is done by placing a *backend* key in the *compilation* section of your *project.json*, as shown below:

```
{
    "version": "7",
    "compilation": {
        "contracts_source_dirs": ["./contracts"],
        "import_remappings": [],
        "backend": {
            "class": "populus.compilation.backends.ViperBackend"
        }
    }
}
```

This will set the populus framework to only pick up viper contracts in the configured contracts directories. Now that everything is configured you can create a viper greeter contract:

```
# contracts/Greeter.vy

greeting: bytes <= 20


@public
def __init__():
    self.greeting = "Hello"


@public
def setGreeting(x: bytes <= 20):
    self.greeting = x


@public
def greet() -> bytes <= 40:
    return self.greeting
```

And run the default populus tests:

```
py.test
```

1.15 LLL Support

Populus provides partial support for LLL, in particular the `lllc` compiler, currently maintained in tandem with Solidity, in the ethereum/solidity repository.

1.15.1 Known limitations

This feature is highly experimental; mixing different languages, e.g. Solidity and LLL or Viper and LLL, is not yet possible.

Since LLL is very low-level, not all language constructs are currently supported. In particular, string literals, especially defined with the `lit` keyword, may be impossible to use.

Finally, LLL programs have no notion of `web3` ABI, as detailed in the Ethereum Contract ABI specification. For that reason, all `.lll` files must have an accompanying `.lll.abi` file, specifying in JSON the on-chain contract's interface.

1.15.2 Installation

On Ubuntu-based systems, `lllc` should be available in the same package as `solc`.

In general, `lllc` should be available in `PATH`; or an `LLLC_BINARY` environment variable must be set and pointing to the `lllc` executable.

To see if it's present:

```
$ lllc --version
LLLC, the Lovely Little Language Compiler
Version: 0.4.19-develop.2017.12.1+commit.c4cbbb05.Linux.g++
```

1.15.3 Using

Automatic initialisation of a `Greeter` project in LLL with `populus --init` is not yet possible.

This section describes how to do it manually.

Change compilation backend

The compilation backend must be changed from its default in *project.json* by placing a *backend* key in the *compilation* section, as shown below:

```
{
    "version": "7",
    "compilation": {
        "contracts_source_dirs": ["./contracts"],
        "import_remappings": [],
        "backend": {
            "class": "populus.compilation.backends.LLLBackend"
        }
    }
}
```

Populus will now only compile LLL contracts in the configured `contracts` directories.

Copy LLL-specific Greeter contract and its ABI specification

These files should be available in the Populus repository, as Greeter.lll and Greeter.lll.abi.

Place them in the `contracts` directory of your project.

Copy LLL-specific test

This file should be available in the Populus repository, as test_greeter_lll.py.

Place it in the `tests` directory of your project.

Remove the Solidity/Viper `test_greeter.py` if it's still present, so `pytest` doesn't trip.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

U

W