Viper Documentation

Release

Vitalik Buterin

Nov 10, 2017

Contents

Viper is an **experimental**, contract-oriented, pythonic programming language that targets the Ethereum Virtual Machine (EVM)

CHAPTER 1

Principles and Goals

- **Security:** It should be possible and natural to build secure smart-contracts in Viper.
- **Language and compiler simplicity:** The language and the compiler implementation should strive to be simple.
- **Auditability:** Viper code should be maximally human-readable. Furthermore, it should be maximally difficult to write misleading code. Simplicity for the reader is more important than simplicity for the writer, and simplicity for readers with low prior experience with Viper (and low prior experience with programming in general) is particularly important.

Because of this Viper aims to provide the following features:

- **Bounds and overflow checking:** On array accesses as well as on arithmetic level.
- **Support for signed integers and decimal fixed point numbers**
- **Decidability:** It should be possible to compute a precise upper bound for the gas consumption of any function call.
- **Strong typing:** Including support for units (e.g. timestamp, timedelta, seconds, wei, wei per second, meters per second squared).
- **Small and understandable compiler code**
- **Limited support for pure functions:** Anything marked constant is not allowed to change the state.

Following the principles and goals, Viper **does not** provide the following features:

- **Modifiers**: For example in Solidity you can define a `function foo() mod1 { ... }`, where `mod1` can be defined elsewhere in the code to include a check that is done before execution, a check that is done after execution, some state changes, or possibly other things. Viper does not have this, because it makes it too easy to write misleading code. `mod1` just looks too innocuous for something that could add arbitrary pre-conditions, post-conditions or state changes. Also, it encourages people to write code where the execution jumps around the file, harming auditability. The usual use case for a modifier is something that performs a single check before execution of a program; our recommendation is to simply inline these checks as asserts.
- **Class inheritance:** Class inheritance requires people to jump between multiple files to understand what a program is doing, and requires people to understand the rules of precedence in case of conflicts ("Which class's

function 'X' is the one that's actually used?"). Hence, it makes code too complicated to understand which negatively impacts auditability.

- **Inline assembly:** Adding inline assembly would make it no longer possible to search for a variable name in order to find all instances where that variable is read or modified.
- **Operator overloading:** Operator overloading makes writing misleading code possible. For example "+" could be overloaded so that it executes commands the are not visible at first glance, such as sending funds the user did not want to send.
- **Recursive calling:** Recursive calling makes it impossible to set an upper bound on gas limits, opening the door for gas limit attacks.
- **Infinite-length loops:** Similar to recurisve calling, infinite-length loops make it impossible to set an upper bound on gas limits, opening the door for gas limit attacks.
- **Binary fixed point:** Decimal fixed point is better, because any decimal fixed point value written as a literal in code has an exact representation, whereas with binary fixed point approximations are often required (e.g. $(0.2)_{10}$ = $(0.001100110011...)_2$, which needs to be truncated), leading to unintuitive results, e.g. in Python 0.3 + 0.3 + 0.3 + 0.1 != 1.

CHAPTER 2

Compatibility-breaking Changelog

- **2017.07.25**: The `def foo() -> num(const): ...` syntax no longer works; you now need to do `def foo() -> num: ...` with a `@constant` decorator on the previous line.
- **2017.07.25**: Functions without a `@payable` decorator now fail when called with nonzero wei.
- **2017.07.25**: A function can only call functions that are declared above it (that is, A can call B only if B appears earlier in the code than A does). This was introduced to prevent infinite looping through recursion.

CHAPTER 3

Glossary

3.1 Installing Viper

Don't panic if the installation fails. Viper is still under development and undergoes constant changes. Installation will be much more simplified and optimized after a stable version release.

Take a deep breath, follow the instructions, and please create an issue if you encounter any errors.

Note: The easiest way to try out the language, experiment with examples, and compile code to `bytecode` or `LLL` is to use the online compiler at https://viper.tools.

3.1.1 Prerequisites

Installing Python 3.6

Viper can only be built using Python 3.6 and higher. If you are already running Python 3.6, skip to the next section, else follow the instructions here to make sure you have the correct Python version installed, and are using that version.

Ubuntu

16.04 and older

Start by making sure your packages are up-to-date:

```
sudo apt-get update
sudo apt-get -y upgrade
```

Install Python 3.6 and some necessary packages:

```
sudo apt-get install build-essential libssl-dev libffi-dev
wget https://www.python.org/ftp/python/3.6.2/Python-3.6.2.tgz
tar xfz Python-3.6.2.tgz
cd Python-3.6.2/
./configure --prefix /usr/local/lib/python3.6
sudo make
sudo make install
```

16.10 and newer

From Ubuntu 16.10 onwards, the Python 3.6 version is in the *universe* repository.

Run the following commands to install:

```
sudo apt-get update
sudo apt-get install python3.6
```

MacOS

Make sure you have Homebrew installed. If you don't have the *brew* command available on the terminal, follow these instructions to get Homebrew on your system.

To install Python 3.6, follow the instructions here: Installing Python 3 on Mac OS X

Also, ensure the GMP arithmetic library is installed using *brew*:

```
brew install gmp
```

Creating a virtual environment

It is **strongly recommended** to install Viper in **a virtual Python environment**, so that new packages installed and dependencies built are strictly contained in your Viper project and will not alter or affect your other development environment set-up.

To create a new virtual environment for Viper run the following commands:

```
virtualenv -p /usr/local/lib/python3.6/bin/python3 --no-site-packages ~/viper-venv
source ~/viper-venv/bin/activate
```

To find out more about virtual environments, check out: virtualenv guide.

3.1.2 Installation

Again, it is **strongly recommended to install Viper** in a **virtual Python environment**. This guide assumes you are in a virtual environment containing Python 3.6.

Get the latest version of Viper by cloning the Github repository, and run the install and test commands:

```
git clone https://github.com/ethereum/viper.git
cd viper
make
make test
```

Additionally, you may try to compile an example contract by running:

```
viper examples/crowdfund.v.py
```

If everything works correctly, you are now able to compile your own smart contracts written in Viper. However, please keep in mind that Viper is still experimental and not ready for production!

Note: For MacOS users:

Apple has deprecated use of OpenSSL in favor of its own TLS and crypto libraries. This means that you will need to export some OpenSSL settings yourself, before you can install Viper.

Use the following commands:

```
export CFLAGS="-I$(brew --prefix openssl)/include"
export LDFLAGS="-L$(brew --prefix openssl)/lib"
pip install scrypt
```

Now you can run the install and test commands again:

```
make install
make test
```

3.1.3 Docker

A Dockerfile is provided in the master branch of the repository. In order to build a Docker Image please run:

```
docker build https://github.com/ethereum/viper.git -t viper:1
docker run -it viper:1 /bin/bash
```

To ensure that everything works correctly after the installtion, please run the test commands and try compiling a contract:

```
make test
viper examples/crowdfund.v.py
```

3.2 Compiling a Contract

To compile a contract, use:

```
viper yourFileName.v.py
```

Note: Since .vy is not official a language supported by any syntax highlighters or linters, it is recommended to name your Viper file ending with *.v.py* in order to have Python syntax highlighting.

An online compiler is available as well, which lets you experiment with the language without having to install Viper. The online compiler allows you to compile to `bytecode` and/or `LLL`.

Note: While the viper version of the online compiler is updated on a regular basis it might be a bit behind the latest version found in the master branch of the repository.

3.3 Viper by Example

3.3.1 Simple Open Auction

As an introductory example of a smart contract written in Viper, we will begin with a simple open auction contract. As we dive into the code, it is important to remember that all Viper syntax is valid Python3 syntax, however not all Python3 functionality is available in Viper.

In this contract, we will be looking at a simple open auction contract where participants can submit bids during a limited time period. When the auction period ends, a predetermined beneficiary will receive the amount of the highest bid.

```
# Open Auction

# Auction params
# Beneficiary recieves money from the highest bidder
beneficiary: public(address)
auction_start: public(timestamp)
auction_end: public(timestamp)

# Current state of auction
highest_bidder: public(address)
highest_bid: public(wei_value)

# Set to true at the end, disallows any change
ended: public(bool)

# Create a simple auction with `_bidding_time`
# seconds bidding time on behalf of the
# beneficiary address `_beneficiary`.
def __init__(_beneficiary: address, _bidding_time: timedelta):
    self.beneficiary = _beneficiary
    self.auction_start = block.timestamp
    self.auction_end = self.auction_start + _bidding_time

# Bid on the auction with the value sent
# together with this transaction.
# The value will only be refunded if the
# auction is not won.
@payable
def bid():
    # Check if bidding period is over.
    assert block.timestamp < self.auction_end
    # Check if bid is high enough
    assert msg.value > self.highest_bid
    if not self.highest_bid == 0:
        # Sends money back to the previous highest bidder
        send(self.highest_bidder,self.highest_bid)
    self.highest_bidder = msg.sender
    self.highest_bid = msg.value

```

```
40
41  # End the auction and send the highest bid
42  # to the beneficiary.
43  def auction_end():
44      # It is a good guideline to structure functions that interact
45      # with other contracts (i.e. they call functions or send Ether)
46      # into three phases:
47      # 1. checking conditions
48      # 2. performing actions (potentially changing conditions)
49      # 3. interacting with other contracts
50      # If these phases are mixed up, the other contract could call
51      # back into the current contract and modify the state or cause
52      # effects (Ether payout) to be performed multiple times.
53      # If functions called internally include interaction with external
54      # contracts, they also have to be considered interaction with
55      # external contracts.
56
57      # 1. Conditions
58      # Check if auction endtime has been reached
59      assert block.timestamp >= self.auction_end
60      # Check if this function has already been called
61      assert not self.ended
62
63      # 2. Effects
64      self.ended = True
65
66      # 3. Interaction
67      send(self.beneficiary, self.highest_bid)
```

As you can see, this example only has a constructor, two methods to call, and a few variables to manage the contract state. Believe it or not, this is all we need for a basic implementation of an auction smart contract.

Let's get started!

```
# Auction params
# Beneficiary recieves money from the highest bidder
beneficiary: public(address)
auction_start: public(timestamp)
auction_end: public(timestamp)

# Current state of auction
highest_bidder: public(address)
highest_bid: public(wei_value)

# Set to true at the end, disallows any change
ended: public(bool)
```

We begin by declaring a few variables to keep track of our contract state. We initialize a global variable `beneficiary` by calling `public` on the datatype `address`. The `beneficiary` will be the receiver of money from the highest bidder. We also initialize the variables `auction_start` and `auction_end` with the datatype `timestamp` to manage the open auction period and `highest_bid` with datatype `wei_value`, the smallest denomination of ether, to manage auction state. The variable `ended` is a boolean to determine whether the auction is officially over.

You may notice all of the variables being passed into the `public` function. By declaring the variable *public*, the variable is callable by external contracts. Initializing the variables without the `public` function defaults to a private declaration and thus only accessible to methods within the same contract. The `public` function additionally creates a 'getter' function for the variable, accessible with a call such as `self.get_beneficiary(some_address)`.

Now, the constructor.

```
# Create a simple auction with `_bidding_time`
# seconds bidding time on behalf of the
# beneficiary address `_beneficiary`.
def __init__(_beneficiary: address, _bidding_time: timedelta):
    self.beneficiary = _beneficiary
    self.auction_start = block.timestamp
    self.auction_end = self.auction_start + _bidding_time
```

The contract is initialized with two arguments: `_beneficiary` of type `address` and `bidding_time` with type `timedelta`, the time difference between the start and end of the auction. We then store these two pieces of information into the contract variables `self.beneficiary` and `self.auction_end`. Notice that we have access to the current time by calling `block.timestamp`. `block` is an object available within any Viper contract and provides information about the block at the time of calling. Similar to `block`, another important object available to us within the contract is `msg`, which provides information on the method caller as we will soon see.

With initial setup out of the way, lets look at how our users can make bids.

```
# Bid on the auction with the value sent
# together with this transaction.
# The value will only be refunded if the
# auction is not won.
@payable
def bid():
    # Check if bidding period is over.
    assert block.timestamp < self.auction_end
    # Check if bid is high enough
    assert msg.value > self.highest_bid
    if not self.highest_bid == 0:
        # Sends money back to the previous highest bidder
        send(self.highest_bidder,self.highest_bid)
    self.highest_bidder = msg.sender
    self.highest_bid = msg.value
```

The `@payable` decorator will allow a user to send some ether to the contract in order to call the decorated method. In this case, a user wanting to make a bid would call the `bid()` method while sending an amount equal to their desired bid (not including gas fees). When calling any method within a contract, we are provided with a built-in variable `msg` and we can access the public address of any method caller with `msg.sender`. Similarly, the amount of ether a user sends can be accessed by calling `msg.value`.

Warning: `msg.sender` will change between internal function calls so that if you're calling a function from the outside, it's correct for the first function call. But then, for the function calls after, `msg.sender` will reference the contract itself as opposed to the sender of the transaction.

Here, we first check whether the current time is before the auction's end time using the `assert` function which takes any boolean statement. We also check to see if the new bid is greater than the highest bid. If the two `assert` statements pass, we can safely continue to the next lines; otherwise, the `bid()` method will throw an error and revert the transaction. If the two `assert` statements the check that the previous bid is not equal to zero pass, we can safely conclude that we have a valid new highest bid. We will send back the previous `highest_bid` to the previous `highest_bidder` and set our new `highest_bid` and `highest_bidder`.

```
# End the auction and send the highest bid
# to the beneficiary.
def auction_end():
    # It is a good guideline to structure functions that interact
```

```
    # with other contracts (i.e. they call functions or send Ether)
    # into three phases:
    # 1. checking conditions
    # 2. performing actions (potentially changing conditions)
    # 3. interacting with other contracts
    # If these phases are mixed up, the other contract could call
    # back into the current contract and modify the state or cause
    # effects (Ether payout) to be performed multiple times.
    # If functions called internally include interaction with external
    # contracts, they also have to be considered interaction with
    # external contracts.

    # 1. Conditions
    # Check if auction endtime has been reached
    assert block.timestamp >= self.auction_end
    # Check if this function has already been called
    assert not self.ended

    # 2. Effects
    self.ended = True

    # 3. Interaction
    send(self.beneficiary, self.highest_bid)
```

With the `auction_end()` method, we check whether our current time is past the `auction_end` time we set upon initialization of the contract. We also check that `self.ended` had not previously been set to True. We do this to prevent any calls to the method if the auction had already ended, which could potentially be malicious if the check had not been made. We then officially end the auction by setting `self.ended` to `True` and sending the highest bid amount to the beneficiary.

And there you have it - an open auction contract. Of course, this is a simplified example with barebones functionality and can be improved. Hopefully, this has provided some insight to the possibilities of Viper. As we move on to exploring more complex examples, we will encounter more design patterns and features of the Viper language.

And of course, no smart contract tutorial is complete without a note on security.

Note: It's always important to keep security in mind when designing a smart

---

contract. As any application becomes more complex, the greater the potential for introducing new risks. Thus, it's always good practice to keep contracts as readable and simple as possible.

Whenever you're ready, let's turn it up a notch in the next example.

3.3.2 Safe Remote Purchases

In this example, we have an escrow contract implementing a system for a trustless transaction between a buyer and a seller. In this system, a seller posts an item for sale and makes a deposit to the contract of twice the item's `value`. At this moment, the contract has a balance of 2 * `value`. The seller can reclaim the deposit and close the sale as long as a buyer has not yet made a purchase. If a buyer is interested in making a purchase, they would make a payment and submit an equal amount for deposit (totaling 2 * `value`) into the contract and locking the contract from further modification. At this moment, the contract has a balance of 4 * `value` and the seller would send the item to buyer. Upon the buyer's receipt of the item, the buyer will mark the item as received in the contract, thereby returning the buyer's deposit (not payment), releasing the remaining funds to the seller, and completing the transaction.

There are certainly others ways of designing a secure escrow system with less overhead for both the buyer and seller, but for the purpose of this example, we want to explore one way how an escrow system can be implemented trustlessly.

Let's go!

```
#Safe Remote Purchase (https://github.com/ethereum/solidity/blob/develop/docs/solidity-by-example.rst) ported to viper and optimized

#Rundown of the transaction:
#1. Seller posts item for sale and posts safety deposit of double the item value. Balance is 2*value.
#(1.1. Seller can reclaim deposit and close the sale as long as nothing was purchased.)
#2. Buyer purchases item (value) plus posts an additional safety deposit (Item value). Balance is 4*value
#3. Seller ships item
#4. Buyer confirms receiving the item. Buyer's deposit (value) is returned. Seller's deposit (2*value) + items value is returned. Balance is 0.

value: public(wei_value) #Value of the item
seller: public(address)
buyer: public(address)
unlocked: public(bool)
#@constant
#def unlocked() -> bool: #Is a refund possible for the seller?
#    return (self.balance == self.value*2)
#
@payable
def __init__():
    assert (msg.value % 2) == 0
    self.value = msg.value / 2 #Seller initializes contract by posting a safety deposit of 2*value of the item up for sale
    self.seller = msg.sender
    self.unlocked = true

def abort():
    assert self.unlocked #Is the contract still refundable
    assert msg.sender == self.seller #Only seller can refund his deposit before any buyer purchases the item
    selfdestruct(self.seller) #Refunds seller, deletes contract

@payable
def purchase():
    assert self.unlocked #Contract still open (item still up for sale)?
    assert msg.value == (2*self.value) #Is the deposit of correct value?
    self.buyer = msg.sender
    self.unlocked = false

def received():
    assert not self.unlocked #Is the item already purchased and pending confirmation of buyer
    assert msg.sender == self.buyer
    send(self.buyer, self.value) #Return deposit (=value) to buyer
    selfdestruct(self.seller) #Returns deposit (=2*value) and the purchase price (=value)
```

This is also a moderately short contract, however a little more complex in logic. Let's break down this contract bit by bit.

```
value: public(wei_value) #Value of the item
seller: public(address)
buyer: public(address)
unlocked: public(bool)
```

Like the other contracts, we begin by declaring our global variables public with their respective datatypes. Remember that the `public` function allows the variables to be *readable* by an external caller, but not *writeable*.

```
@payable
def __init__():
    assert (msg.value % 2) == 0
    self.value = msg.value / 2 #Seller initializes contract by posting a safety␣
↪deposit of 2*value of the item up for sale
    self.seller = msg.sender
    self.unlocked = true
```

With a `@payable` decorator on the constructor, the contract creator will be required to make an initial deposit equal to twice the item's `value` to initialize the contract, which will be later returned. This is in addition to the gas fees needed to deploy the contract on the blockchain, which is not returned. We `assert` that the deposit is divisible by 2 to ensure that the seller deposited a valid amount. The constructor stores the item's value in the contract variable `self.value` and saves the contract creator into `self.seller`. The contract variable `self.unlocked` is initialized to `True`.

```
def abort():
    assert self.unlocked #Is the contract still refundable
    assert msg.sender == self.seller #Only seller can refund his deposit before any␣
↪buyer purchases the item
    selfdestruct(self.seller) #Refunds seller, deletes contract
```

The `abort()` method is a method only callable by the seller and while the contract is still `unlocked` - meaning it is callable only prior to any buyer making a purchase. As we will see in the `purchase()` method that when a buyer calls the `purchase()` method and sends a valid amount to the contract, the contract will be locked and the seller will no longer be able to call `abort()`.

When the seller calls `abort()` and if the `assert` statements pass, the contract will call the `selfdestruct()` function and refunds the seller and subsequently destroys the contract.

```
@payable
def purchase():
    assert self.unlocked #Contract still open (item still up for sale)?
    assert msg.value == (2*self.value) #Is the deposit of correct value?
    self.buyer = msg.sender
    self.unlocked = false
```

Like the constructor, the `purchase()` method has a `@payable` decorator, meaning it can be called with a payment. For the buyer to make a valid purchase, we must first `assert` that the contract's unlocked property is `False` and that the amount sent is equal to twice the item's value. We then set the buyer to the `msg.sender` and lock the contract. At this point, the contract has a balance equal to 4 times the item value and the seller must send the item to the buyer.

```
def received():
    assert not self.unlocked #Is the item already purchased and pending confirmation␣
↪of buyer
    assert msg.sender == self.buyer
    send(self.buyer, self.value) #Return deposit (=value) to buyer
    selfdestruct(self.seller) #Returns deposit (=2*value) and the purchase price␣
↪(=value)
```

Finally, upon the buyer's receipt of the item, the buyer can confirm their receipt by calling the `received()` method

to distribute the funds as intended - the seller receives 3/4 of the contract balance and the buyer receives 1/4.

By calling `received()`, we begin by checking that the contract is indeed locked, ensuring that a buyer had previously paid. We also ensure that this method is only callable by the buyer. If these two `assert` statements pass, we refund the buyer their initial deposit and send the seller the remaining funds. The contract is finally destroyed and the transaction is complete.

Whenever we're ready, let's move on to the next example.

3.3.3 Crowdfund

Now, let's explore a straightforward example for a crowdfunding contract where prospective participants can contribute funds to a campaign. If the total contribution to the campaign reaches or surpasses a predetermined funding goal, the funds will be sent to the beneficiary at the end of the campaign deadline. Participants will be refunded their respective contributions if the total funding does not reach its target goal.

```
funders: {sender: address, value: wei_value}[num]
nextFunderIndex: num
beneficiary: address
deadline: timestamp
goal: wei_value
refundIndex: num
timelimit: timedelta

# Setup global variables
def __init__(_beneficiary: address, _goal: wei_value, _timelimit: timedelta):
    self.beneficiary = _beneficiary
    self.deadline = block.timestamp + _timelimit
    self.timelimit = _timelimit
    self.goal = _goal

# Participate in this crowdfunding campaign
@payable
def participate():
    assert block.timestamp < self.deadline
    nfi = self.nextFunderIndex
    self.funders[nfi] = {sender: msg.sender, value: msg.value}
    self.nextFunderIndex = nfi + 1

# Enough money was raised! Send funds to the beneficiary
def finalize():
    assert block.timestamp >= self.deadline and self.balance >= self.goal
    selfdestruct(self.beneficiary)

# Not enough money was raised! Refund everyone (max 30 people at a time
# to avoid gas limit issues)
def refund():
    assert block.timestamp >= self.deadline and self.balance < self.goal
    ind = self.refundIndex
    for i in range(ind, ind + 30):
        if i >= self.nextFunderIndex:
            self.refundIndex = self.nextFunderIndex
            return
        send(self.funders[i].sender, self.funders[i].value)
        self.funders[i] = None
    self.refundIndex = ind + 30
```

Most of this code should be relatively straightforward after going through our previous examples. Let's dive right in.

```
funders: {sender: address, value: wei_value}[num]
nextFunderIndex: num
beneficiary: address
deadline: timestamp
goal: wei_value
refundIndex: num
timelimit: timedelta
```

Like other examples, we begin by initiating our variables - except this time, we're not calling them with the `public` function. Variables initiated this way are, by default, private.

..note :: Unlike the existence of the function `public()`, there is no equivalent `private()` function. Variables simply default to private if initiated without the `public()` function.

The `funders` variable is initiated as a mapping where the key is a number, and the value is a struct representing the contribution of each participant. This struct contains each participant's public address and their respective value contributed to the fund. The key corresponding to each struct in the mapping will be represented by the variable `nextFunderIndex` which is incremented with each additional contributing participant. Variables initialized with `num` type without an explicit value, such as `nextFunderIndex`, defaults to `0`. The `beneficiary` will be the final receiver of the funds once the crowdfunding period is over - as determined by the `deadline` and `timelimit` variables. The `goal` variable is the target total contribution of all participants. `refundIndex` is a variable for bookkeeping purposes in order to avoid gas limit issues in the scenario of a refund.

```
# Setup global variables
def __init__(_beneficiary: address, _goal: wei_value, _timelimit: timedelta):
    self.beneficiary = _beneficiary
    self.deadline = block.timestamp + _timelimit
    self.timelimit = _timelimit
    self.goal = _goal
```

Our constructor function takes 3 arguments: the beneficiary's address, the goal in wei value, and the difference in time from start to finish of the crowdfunding. We initialize the arguments as contract variables with their corresponding names. Additionally, a `self.deadline` is initialized to set a definitive end time for the crowdfunding period.

Now lets take a look at how a person can participate in the crowdfund.

```
# Participate in this crowdfunding campaign
@payable
def participate():
    assert block.timestamp < self.deadline
    nfi = self.nextFunderIndex
    self.funders[nfi] = {sender: msg.sender, value: msg.value}
    self.nextFunderIndex = nfi + 1
```

Once again, we see the `@payable` decorator on a method, which allows a person to send some ether along with a call to the method. In this case, the `participate()` method accesses the sender's address with `msg.sender` and the corresponding amount sent with `msg.value`. This information is stored into a struct and then saved into the `funders` mapping with `self.nextFunderIndex` as the key. As more participants are added to the mapping, `self.nextFunderIndex` increments appropriately to properly index each participant.

```
# Enough money was raised! Send funds to the beneficiary
def finalize():
    assert block.timestamp >= self.deadline and self.balance >= self.goal
    selfdestruct(self.beneficiary)
```

The `finalize()` method is used to complete the crowdfunding process. However, to complete the crowdfunding, the method first checks to see if the crowdfunding period is over and that the balance has reached/passed its set goal.

If those two conditions pass, the contract calls the `selfdestruct()` function and sends the collected funds to the beneficiary.

Note: Notice that we have access to the total amount sent to the contract by

calling `self.balance`, a variable we never explicitly set. Similar to `msg` and `block`, `self.balance` is a built-in variable thats available in all Viper contracts.

We can finalize the campaign if all goes well, but what happens if the crowdfunding campaign isn't successful? We're going to need a way to refund all the participants.

```
# Not enough money was raised! Refund everyone (max 30 people at a time
# to avoid gas limit issues)
def refund():
    assert block.timestamp >= self.deadline and self.balance < self.goal
    ind = self.refundIndex
    for i in range(ind, ind + 30):
        if i >= self.nextFunderIndex:
            self.refundIndex = self.nextFunderIndex
            return
        send(self.funders[i].sender, self.funders[i].value)
        self.funders[i] = None
    self.refundIndex = ind + 30
```

In the `refund()` method, we first check that the crowdfunding period is indeed over and that the total collected balance is less than the `goal` with the `assert` statement . If those two conditions pass, we then loop through every participant and call `send()` to send each participant their respective contribution. For the sake of gas limits, we group the number of contributors in batches of 30 and refund them one at a time. Unfortunately, if there's a large number of of participants, multiple calls to `refund()` may be necessary.

3.3.4 Voting

In this contract, we will implement a system for participants to vote on a list of proposals. The chairperson of the contract will be able to give each participant the right to vote and each participant may choose to vote or delegate their vote to another voter. Finally, a winning proposal will be determined upon calling the `winning_proposals()` method, which iterates through all the proposals and returns the one with the greatest number of votes.

```
1  # Voting with delegation.
2
3  # Information about voters
4  voters: public({
5      # weight is accumulated by delegation
6      weight: num,
7      # if true, that person already voted
8      voted: bool,
9      # person delegated to
10     delegate: address,
11     # index of the voted proposal
12     vote: num
13 }[address])
14
15 # This is a type for a list of proposals.
16 proposals: public({
17     # short name (up to 32 bytes)
18     name: bytes32,
```

```
    # number of accumulated votes
    vote_count: num
}[num])

voter_count: public(num)
chairperson: public(address)

# Setup global variables
def __init__(_proposalNames: bytes32[2]):
    self.chairperson = msg.sender
    self.voter_count = 0
    for i in range(2):
        self.proposals[i] = {
            name: _proposalNames[i],
            vote_count: 0
        }

# Give `voter` the right to vote on this ballot.
# May only be called by `chairperson`.
def give_right_to_vote(voter: address):
    # Throws if sender is not chairpers
    assert msg.sender == self.chairperson
    # Throws if voter has already voted
    assert not self.voters[voter].voted
    # Throws if voters voting weight isn't 0
    assert self.voters[voter].weight == 0
    self.voters[voter].weight = 1
    self.voter_count += 1

# Delegate your vote to the voter `to`.
def delegate(_to: address):
    to = _to
    # Throws if sender has already voted
    assert not self.voters[msg.sender].voted
    # Throws if sender tries to delegate their vote to themselves
    assert not msg.sender == to
    # loop can delegate votes up to the current voter count
    for i in range(self.voter_count, self.voter_count+1):
        if self.voters[to].delegate:
        # Because there are not while loops, use recursion to forward the delegation
        # self.delegate(self.voters[to].delegate)
            assert self.voters[to].delegate != msg.sender
            to = self.voters[to].delegate
    self.voters[msg.sender].voted = True
    self.voters[msg.sender].delegate = to
    if self.voters[to].voted:
        # If the delegate already voted,
        # directly add to the number of votes
        self.proposals[self.voters[to].vote].vote_count += self.voters[msg.sender].
↪weight
    else:
        # If the delegate did not vote yet,
        # add to her weight.
        self.voters[to].weight += self.voters[msg.sender].weight

# Give your vote (including votes delegated to you)
# to proposal `proposals[proposal].name`.
def vote(proposal: num):
```

```
    assert not  self.voters[msg.sender].voted
    self.voters[msg.sender].voted = True
    self.voters[msg.sender].vote = proposal
    # If `proposal` is out of the range of the array,
    # this will throw automatically and revert all
    # changes.
    self.proposals[proposal].vote_count += self.voters[msg.sender].weight

# Computes the winning proposal taking all
# previous votes into account.
@constant
def winning_proposal() -> num:
    winning_vote_count = 0
    for i in range(5):
        if self.proposals[i].vote_count > winning_vote_count:
            winning_vote_count = self.proposals[i].vote_count
            winning_proposal = i
    return winning_proposal

# Calls winning_proposal() function to get the index
# of the winner contained in the proposals array and then
# returns the name of the winner
@constant
def winner_name() -> bytes32:
    return self.proposals[self.winning_proposal()].name
```

As we can see, this is contract of moderate length which we will dissect section by section. Let's begin!

```
# Information about voters
voters: public({
    # weight is accumulated by delegation
    weight: num,
    # if true, that person already voted
    voted: bool,
    # person delegated to
    delegate: address,
    # index of the voted proposal
    vote: num
}[address])

# This is a type for a list of proposals.
proposals: public({
    # short name (up to 32 bytes)
    name: bytes32,
    # number of accumulated votes
    vote_count: num
}[num])
```

The variable `voters` is initialized as a mapping where the key is the voter's public address and the value is a struct describing the voter's properties: `weight`, `voted`, `delegate`, and `vote`, along with their respective datatypes.

Similarly, the `proposals` variable is initialized as a `public` mapping with `num` as the key's datatype and a struct to represent each proposal with the properties `name` and `vote_count`. Like our last example, we can access any value by key'ing into the mapping with a number just as one would with an index in an array.

Then, `voter_count` and `chairperson` are initialized as `public` with their respective datatypes.

Let's move onto the constructor.

```
# Setup global variables
def __init__(_proposalNames: bytes32[2]):
    self.chairperson = msg.sender
    self.voter_count = 0
    for i in range(2):
        self.proposals[i] = {
            name: _proposalNames[i],
            vote_count: 0
        }
```

Warning: Both `msg.sender` and `msg.balance` change between internal function calls so that if you're calling a function from the outside, it's correct for the first function call. But then, for the function calls after, `msg.sender` and `msg.balance` reference the contract itself as opposed to the sender of the transaction.

In the constructor, we hard-coded the contract to accept an array argument of exactly two proposal names of type `bytes32` for the contracts initialization. Because upon initialization, the `__init__()` method is called by the contract creator, we have access to the contract creator's address with `msg.sender` and store it in the contract variable `self.chairperson`. We also initialize the contract variable `self.voter_count` to zero to initially represent the number of votes allowed. This value will be incremented as each participant in the contract is given the right to vote by the method `give_right_to_vote()`, which we will explore next. We loop through the two proposals from the argument and insert them into `proposals` mapping with their respective index in the original array as its key.

Now that the initial setup is done, lets take a look at the functionality.

```
# Give `voter` the right to vote on this ballot.
# May only be called by `chairperson`.
def give_right_to_vote(voter: address):
    # Throws if sender is not chairpers
    assert msg.sender == self.chairperson
    # Throws if voter has already voted
    assert not self.voters[voter].voted
    # Throws if voters voting weight isn't 0
    assert self.voters[voter].weight == 0
    self.voters[voter].weight = 1
    self.voter_count += 1
```

We need a way to control who has the ability to vote. The method `give_right_to_vote()` is a method callable by only the chairperson by taking a voter address and granting it the right to vote by incrementing the voter's `weight` property. We sequentially check for 3 conditions using `assert`. The `assert not` function will check for falsy boolean values - in this case, we want to know that the voter has not already voted. To represent voting power, we will set their `weight` to `1` and we will keep track of the total number of voters by incrementing `voter_count`.

```
# Delegate your vote to the voter `to`.
def delegate(_to: address):
    to = _to
    # Throws if sender has already voted
    assert not self.voters[msg.sender].voted
    # Throws if sender tries to delegate their vote to themselves
    assert not msg.sender == to
    # loop can delegate votes up to the current voter count
    for i in range(self.voter_count, self.voter_count+1):
        if self.voters[to].delegate:
        # Because there are not while loops, use recursion to forward the delegation
        # self.delegate(self.voters[to].delegate)
```

```
            assert self.voters[to].delegate != msg.sender
            to = self.voters[to].delegate
    self.voters[msg.sender].voted = True
    self.voters[msg.sender].delegate = to
    if self.voters[to].voted:
        # If the delegate already voted,
        # directly add to the number of votes
        self.proposals[self.voters[to].vote].vote_count += self.voters[msg.sender].
↪weight
    else:
        # If the delegate did not vote yet,
        # add to her weight.
        self.voters[to].weight += self.voters[msg.sender].weight
```

In the method `delegate`, firstly, we check to see that `msg.sender` has not already voted and secondly, that the target delegate and the `msg.sender` are not the same. Voters shouldn't be able to delegate votes to themselves. We, then, loop through all the voters to determine whether the person delegate to had further delegated their vote to someone else in order to follow the chain of delegation. We then mark the `msg.sender` as having voted if they delegated their vote. We increment the proposal's `vote_count` directly if the delegate had already voted or increase the delegate's vote `weight` if the delegate has not yet voted.

```
# Give your vote (including votes delegated to you)
# to proposal `proposals[proposal].name`.
def vote(proposal: num):
    assert not  self.voters[msg.sender].voted
    self.voters[msg.sender].voted = True
    self.voters[msg.sender].vote = proposal
    # If `proposal` is out of the range of the array,
    # this will throw automatically and revert all
    # changes.
    self.proposals[proposal].vote_count += self.voters[msg.sender].weight
```

Now, let's take a look at the logic inside the `vote()` method, which is surprisingly simple. The method takes the key of the proposal in the `proposals` mapping as an argument, check that the method caller had not already voted, sets the voter's `vote` property to the proposal key, and increments the proposals `vote_count` by the voter's `weight`.

With all the basic functionality complete, what's left is simply returning the winning proposal. To do this, we have two methods: `winning_proposal()`, which returns the key of the proposal, and `winner_name()`, returning the name of the proposal. Notice the `@constant` decorator on these two methods. We do this because the two methods only read the blockchain state and do not modify it. Remember, reading the blockchain state is free; modifying the state costs gas. By having the `@constant` decorator, we let the EVM know that this is a read-only function and we benefit by saving gas fees.

```
# Computes the winning proposal taking all
# previous votes into account.
@constant
def winning_proposal() -> num:
    winning_vote_count = 0
    for i in range(5):
        if self.proposals[i].vote_count > winning_vote_count:
            winning_vote_count = self.proposals[i].vote_count
            winning_proposal = i
    return winning_proposal
```

The `winning_proposal()` method returns the key of proposal in the `proposals` mapping. We will keep track of greatest number of votes and the winning proposal with the variables `winning_vote_count` and `winning_proposal`, respectively by looping through all the proposals.

```
# Calls winning_proposal() function to get the index
# of the winner contained in the proposals array and then
# returns the name of the winner
@constant
def winner_name() -> bytes32:
    return self.proposals[self.winning_proposal()].name
```

And finally, the `winner_name()` method returns the name of the proposal by key'ing into the `proposals` mapping with the return result of the `winning_proposal()` method.

And there you have it - a voting contract. Currently, many transactions are needed to assign the rights to vote to all participants. As an exercise, can we try to optimize this?

Now that we're familiar with basic contracts. Let's step up the difficulty.

3.3.5 Company Stock

This contract is just a tad bit more thorough than the ones we've previously encountered. In this example, we are going to look at a comprehensive contract that manages the holdings of all shares of a company. The contract allows for a person to buy, sell, and transfer shares of a company as well as allowing for the company to pay a person in ether. The company, upon initialization of the contract, holds all shares of the company at first but can sell them all.

Let's get started.

```
# Own shares of a company!
company: public(address)
total_shares: public(currency_value)
price: public(num (wei / currency))

# Store ledger of stockholder holdings
holdings: currency_value[address]

# Setup company
def __init__(_company: address, _total_shares: currency_value,
        initial_price: num(wei / currency) ):
    assert _total_shares > 0
    assert initial_price > 0

    self.company = _company
    self.total_shares = _total_shares

    self.price = initial_price

    # Company holds all the shares at first, but can sell them all
    self.holdings[self.company] = _total_shares

@constant
def stock_available() -> currency_value:
    return self.holdings[self.company]

# Give value to company and get stock in return
@payable
def buy_stock():
    # Note: full amount is given to company (no fractional shares),
    #       so be sure to send exact amount to buy shares
    buy_order = msg.value / self.price # rounds down

```

```
    # There are enough shares to buy
    assert self.stock_available() >= buy_order

    # Take the shares off the market and give to stockholder
    self.holdings[self.company] -= buy_order
    self.holdings[msg.sender] += buy_order

# So someone can find out how much they have
@constant
def get_holding(_stockholder: address) -> currency_value:
    return self.holdings[_stockholder]

# The amount the company has on hand in cash
@constant
def cash() -> wei_value:
    return self.balance

# Give stock back to company and get my money back!
def sell_stock(sell_order: currency_value):
    assert sell_order > 0 # Otherwise, will fail at send() below
    # Can only sell as much stock as you own
    assert self.get_holding(msg.sender) >= sell_order
    # Company can pay you
    assert self.cash() >= (sell_order * self.price)

    # Sell the stock, send the proceeds to the user
    # and put the stock back on the market
    self.holdings[msg.sender] -= sell_order
    self.holdings[self.company] += sell_order
    send(msg.sender, sell_order * self.price)

# Transfer stock from one stockholder to another
# (Assumes the receiver is given some compensation, but not enforced)
def transfer_stock(receiver: address, transfer_order: currency_value):
    assert transfer_order > 0 # AUDIT revealed this!
    # Can only trade as much stock as you own
    assert self.get_holding(msg.sender) >= transfer_order

    # Debit sender's stock and add to receiver's address
    self.holdings[msg.sender] -= transfer_order
    self.holdings[receiver] += transfer_order

# Allows the company to pay someone for services rendered
def pay_bill(vendor: address, amount: wei_value):
    # Only the company can pay people
    assert msg.sender == self.company
    # And only if there's enough to pay them with
    assert self.cash() >= amount

    # Pay the bill!
    send(vendor, amount)

# The amount a company has raised in the stock offering
@constant
def debt() -> wei_value:
    return (self.total_shares - self.holdings[self.company]) * self.price

# The balance sheet of the company
```

```
@constant
def worth() -> wei_value:
    return self.cash() - self.debt()
```

The contract contains a number of methods that modify the contract state as well as a few 'getter' methods to read it. As always, we begin by initiating our variables.

```
# Own shares of a company!
company: public(address)
total_shares: public(currency_value)
price: public(num (wei / currency))

# Store ledger of stockholder holdings
holdings: currency_value[address]
```

We initiate the `company` variable to be of type `address` that's public. The `total_shares` variable is of type `currency_value`, which in this case represents the total available shares of the company. The `price` variable represents the wei value of a share and `holdings` is a mapping that maps an address to the number of shares the address owns.

```
# Setup company
def __init__(_company: address, _total_shares: currency_value,
        initial_price: num(wei / currency) ):
    assert _total_shares > 0
    assert initial_price > 0

    self.company = _company
    self.total_shares = _total_shares

    self.price = initial_price

    # Company holds all the shares at first, but can sell them all
    self.holdings[self.company] = _total_shares
```

In the constructor, we set up the contract to check for valid inputs during the initialization of the contract via the two `assert` statements. If the inputs are valid, the contract variables are set accordingly and the company's address is initialized to hold all shares of the company in the `holdings` mapping.

```
@constant
def stock_available() -> currency_value:
    return self.holdings[self.company]
```

We will be seeing a few `@constant` decorators in this contract - which is used to decorate methods that simply read the contract state or return a simple calculation on the contract state without modifying it. Remember, reading the blockchain is free, writing on it is not. Since Viper is a statically typed language, we see an arrow following the definition of the `stock_available()` method, which simply represents the datatype which the function is expected to return. In the method, we simply key into `self.holdings` with the company's address and check it's holdings.

Now, lets take a look at a method that lets a person buy stock from the company's holding.

```
# Give value to company and get stock in return
@payable
def buy_stock():
    # Note: full amount is given to company (no fractional shares),
    #       so be sure to send exact amount to buy shares
    buy_order = msg.value / self.price # rounds down
```

```
    # There are enough shares to buy
    assert self.stock_available() >= buy_order

    # Take the shares off the market and give to stockholder
    self.holdings[self.company] -= buy_order
    self.holdings[msg.sender] += buy_order
```

The `buy_stock()` method is a `@payable` method which takes an amount of ether sent and calculates the `buy_order` (the stock value equivalence at the time of call). The number of shares is deducted from the company's holdings and transferred to the sender's in the `holdings` mapping.

Now that people can buy shares, how do we check someone's holdings?

```
# So someone can find out how much they have
@constant
def get_holding(_stockholder: address) -> currency_value:
    return self.holdings[_stockholder]
```

The `get_holdings()` is another `@constant` method that takes an `address` and returns its corresponding stock holdings by keying into `self.holdings`.

```
# The amount the company has on hand in cash
@constant
def cash() -> wei_value:
    return self.balance
```

To check the ether balance of the company, we can simply call the getter method `cash()`.

```
# Give stock back to company and get my money back!
def sell_stock(sell_order: currency_value):
    assert sell_order > 0 # Otherwise, will fail at send() below
    # Can only sell as much stock as you own
    assert self.get_holding(msg.sender) >= sell_order
    # Company can pay you
    assert self.cash() >= (sell_order * self.price)

    # Sell the stock, send the proceeds to the user
    # and put the stock back on the market
    self.holdings[msg.sender] -= sell_order
    self.holdings[self.company] += sell_order
    send(msg.sender, sell_order * self.price)
```

To sell a stock, we have the `sell_stock()` method which takes a number of stocks a person wishes to sell, and sends the equivalent value in ether to the seller's address. We first `assert` that the number of stocks the person wishes to sell is a value greater than `0`. We also `assert` to see that the user can only sell as much as the user owns and that the company has enough ether to complete the sale. If all conditions are met, the holdings are deducted from the seller and given to the company. The ethers are then sent to the seller.

```
# Transfer stock from one stockholder to another
# (Assumes the receiver is given some compensation, but not enforced)
def transfer_stock(receiver: address, transfer_order: currency_value):
    assert transfer_order > 0 # AUDIT revealed this!
    # Can only trade as much stock as you own
    assert self.get_holding(msg.sender) >= transfer_order

    # Debit sender's stock and add to receiver's address
```

```
    self.holdings[msg.sender] -= transfer_order
    self.holdings[receiver] += transfer_order
```

A stockholder can also transfer their stock to another stockholder with the `transfer_stock()` method. The method takes a receiver address and the number of shares to send. It first `asserts` that the amount being sent is greater than `0` and `asserts` whether the sender has enough stocks to send. If both conditions are satisfied, the transfer is made.

```
# Allows the company to pay someone for services rendered
def pay_bill(vendor: address, amount: wei_value):
    # Only the company can pay people
    assert msg.sender == self.company
    # And only if there's enough to pay them with
    assert self.cash() >= amount

    # Pay the bill!
    send(vendor, amount)
```

The company is also allowed to pay out an amount in ether to an address by calling the `pay_bill()` method. This method should only be callable by the company and thus first checks whether the method caller's address matches that of the company. Another important condition to check is that the company has enough funds to pay the amount. If both conditions satisfy, the contract sends its ether to an address.

```
# The amount a company has raised in the stock offering
@constant
def debt() -> wei_value:
    return (self.total_shares - self.holdings[self.company]) * self.price
```

We can also check how much the company has raised by multiplying the number of shares the company has sold and the price of each share. We can get this value by calling the `debt()` method.

```
# The balance sheet of the company
@constant
def worth() -> wei_value:
    return self.cash() - self.debt()
```

Finally, in this `worth()` method, we can check the worth of a company by subtracting its debt from its ether balance.

This contract has been the most thorough example so far in terms of its functionality and features. Yet despite the thoroughness of such a contract, the logic remained simple. Hopefully, by now, the Viper language has convinced you of its capabilities and readability in writing smart contracts.

3.4 Viper in Depth

This section should provide you with all you need to know about Viper. If something is missing here, please contact us on Gitter or make a pull request on Github.

3.4.1 Structure of a Contract

Contracts in Viper are contained within files, with each file being one smart-contract. Files in Viper are similar to classes in object-oriented languages. Each file can contain declarations of *State Variables*, *Functions*, and structure-structs-types.

State Variables

State variables are values which are permanently stored in contract storage.

```
storedData: num
```

See the *Types* section for valid state variable types and visibility-and-getters for possible choices for visibility.

Functions

Functions are the executable units of code within a contract.

```
@payable
function bid(): // Function
  // ...
}
```

function-calls can happen internally or externally and have different levels of visibility (visibility-and-getters) towards other contracts.

3.4.2 Types

Viper is a statically typed language, which means that the type of each variable (state and local) needs to be specified or at least known at compile-time. Viper provides several elementary types which can be combined to form complex types.

In addition, types can interact with each other in expressions containing operators.

Value Types

The following types are also called value types because variables of these types will always be passed by value, i.e. they are always copied when they are used as function arguments or in assignments.

Booleans

`bool`: The possible values are constants `true` and `false`.

Operators:

- `not` (logical negation)
- `and` (logical conjunction, "&&")
- `or` (logical disjunction, "||")
- == (equality)
- `not ... == ...` ``, ``!= (inequality)

The operators `or` and `and` apply the common short-circuiting rules. This means that in the expression `f(x) or g(y)`, if `f(x)` evaluates to `true`, `g(y)` will not be evaluated even if it may have side-effects.

Integers

`num`: equivalent to‘‘int128‘‘, a signed integer strictly between -2**127 and 2**127-1.

Operators:

- Comparisons: <=, <, ==, !=, >=, > (evaluate to `bool`)
- Arithmetic operators: +, −, unary −, unary +, *, /, % (remainder)

Decimals

`decimal`: a decimal fixed point value with the integer component represented as a `num` and the fractional component supporting up to ten decimal places.

Time

`timestamp`: a timestamp value with a base unit of one second, represented as a `num`.

`timedelta`: a number of seconds (note: two timedeltas can be added together, as can a timedelta and a timestamp, but not two timestamps), represented as a `num`.

Value

`wei_value`: an amount of ether with a base unit of one wei, represented as a `num`.

`currency_value`: represents an amount of currency and should be used to represent assets where ether is traded for value, represented as a `num`.

Address

`address`: Holds an Ethereum address (20 byte value).

Members of Addresses

- `balance` and `send`

It is possible to query the balance of an address using the property `balance` and to send Ether (in units of wei) to an address using the `send` function:

```
x: address

def foo(x: address):
    if (x.balance < 10 and self.balance >= 10):
        x.send(10)
```

Fixed-size byte arrays

`bytes32`: 32 bytes

```
# Declaration
hash: bytes32

# Assignment
self.hash = _hash
```

`bytes <= maxlen`: a byte array with the given maximum length

```
# Declaration
name: bytes <= 5

# Assignment
self.name = _name
```

`type[length]`: finite list

```
# Declaration
numbers: num[3]

# Assignment
self.numbers[0] = _num1
```

Structs

Structs are custom defined types that can group several variables. They can be accessed via `struct.argname`.

```
# Information about voters
voters: public({
    # weight is accumulated by delegation
    weight: num,
    # if true, that person already voted
    voted: bool,
    # person delegated to
    delegate: address,
    # index of the voted proposal
    vote: num
})
```

Mappings

Mapping types are declared as `_ValueType[_KeyType]`. Here `_KeyType` can be almost any type except for mappings, a contract, or a struct. `_ValueType` can actually be any type, including mappings.

Mappings can be seen as hash tables which are virtually initialized such that every possible key exists and is mapped to a value whose byte-representation is all zeros: a type's default value. The similarity ends here, though: The key data is not actually stored in a mapping, only its `keccak256` hash used to look up the value.

Because of this, mappings do not have a length or a concept of a key or value being "set".

Mappings are only allowed as state variables.

It is possible to mark mappings `public` and have Viper create a getter. The `_KeyType` will become a required parameter for the getter and it will return `_ValueType`.

Note: Mappings can only be accessed, not iterated over.

3.5 Contributing

Help is always appreciated!

To get started, you can try building-from-source in order to familiarize yourself with the components of Viper and the build process. Also, it may be useful to become well-versed at writing smart-contracts in Viper.

3.5.1 Types of Contributions

In particular, we need help in the following areas:

- Improving the documentation
- Responding to questions from other users on StackExchange and the Viper Gitter
- Fixing and responding to Viper's GitHub issues

3.5.2 How to Report Issues

To report an issue, please use the GitHub issues tracker. When reporting issues, please mention the following details:

- Which version of Viper you are using
- What was the source code (if applicable)
- Which platform are you running on
- Your operating system name and version
- Detailed steps to reproduce the issue
- What was the result of the issue
- What the expected behaviour is

Reducing the source code that caused the issue to a bare minimum is always very helpful and sometimes even clarifies a misunderstanding.

3.5.3 Fix Bugs

Go through the GitHub issues or report bugs at https://github.com/ethereum/viper/issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

3.5.4 Workflow for Pull Requests

In order to contribute, please fork off of the `master` branch and make your changes there. Your commit messages should detail *why* you made your change in addition to *what* you did (unless it is a tiny change).

If you need to pull in any changes from `master` after making your fork (for example, to resolve potential merge conflicts), please avoid using `git merge` and instead, `git rebase` your branch.

Implement Features

Additionally, if you are writing a new feature, please ensure you write appropriate Boost test cases and place them under `tests/`.

However, if you are making a larger change, please consult with the Gitter channel, first.

Also, even though we do CI testing, please make sure that the tests pass for supported Python version and ensure that it builds locally before submitting a pull request.

Thank you for your help!

3.6 Frequently Asked Questions

3.6.1 Basic Questions

3.6.2 Advanced Questions

Index