Vyper Documentation

Vitalik Buterin

Mar 11, 2019

Contents

Vyper is a 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 Vyper.
- **Language and compiler simplicity:** The language and the compiler implementation should strive to be simple.
- **Auditability:** Vyper 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 Vyper (and low prior experience with programming in general) is particularly important.

Because of this Vyper 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, Vyper **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. Vyper 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.
- **Function overloading** - This can cause lots of confusion on which function is called at any given time. Thus it's easier to write missleading code (`foo("hello")` logs "hello" but `foo("hello", "world")` steals you funds). Another problem with function overloading is that it makes the code much harder to search through as you have to keep track on which call refers to which function.
- **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. $(0.2)_{10}$ = $(0.001100110011\ldots)_2$, 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

- **2018.08.09**: Add support for default parameters.
- **2018.06.08**: Tagged first beta.
- **2018.05.23**: Changed *wei_value* to be *uint256*.
- **2018.04.03**: Changed bytes declaration from 'bytes <= n' to 'bytes[n]'.
- **2018.03.27**: Renaming `signed256` to `int256`.
- **2018.03.22**: Add modifiable and static keywords for external contract calls.
- **2018.03.20**: Renaming `__log__` to `event`.
- **2018.02.22**: Renaming num to int128, and num256 to uint256.
- **2018.02.13**: Ban functions with payable and constant decorators.
- **2018.02.12**: Division by num returns decimal type.
- **2018.02.09**: Standardize type conversions.
- **2018.02.01**: Functions cannot have the same name as globals.
- **2018.01.27**: Change getter from get_var to var.
- **2018.01.11**: Change version from 0.0.2 to 0.0.3
- **2018.01.04**: Types need to be specified on assignment (VIP545).
- **2017.01.02** Change `as_wei_value` to use quotes for units.
- **2017.12.25**: Change name from Viper to Vyper.
- **2017.12.22**: Add `continue` for loops
- **2017.11.29**: `@internal` renamed to `@private`.
- **2017.11.15**: Functions require either `@internal` or `@public` decorators.
- **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 Vyper

Don't panic if the installation fails. Vyper 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://vyper.online/.

3.1.1 Prerequisites

Installing Python 3.6

Vyper 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
```

Note: If you get the error *Python.h: No such file or directory* you need to install the python header files for the Python C API with

```
sudo apt-get install python3-dev
```

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 following libraries are installed using *brew*:

```
brew install gmp leveldb
```

Creating a virtual environment

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

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

```
sudo apt install virtualenv
virtualenv -p python3.6 --no-site-packages ~/vyper-venv
source ~/vyper-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 Vyper** in a **virtual Python environment**. This guide assumes you are in a virtual environment containing Python 3.6.

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

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

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

```
vyper examples/crowdfund.vy
```

If everything works correctly, you are now able to compile your own smart contracts written in Vyper. If any unexpected errors or exceptions are encountered, please feel free create an issue <https://github.com/ethereum/vyper/issues/new>.

Note: If you get the error *fatal error: openssl/aes.h: No such file or directory* in the output of *make*, then run *sudo apt-get install libssl-dev1*, then run *make* again.

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 Vyper.

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
make test
```

If you get the error *ld: library not found for -lyaml* in the output of *make*, make sure *libyaml* is installed using *brew info libyaml*. If it is installed, add its location to the compile flags as well:

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

You can then run *make* and *make test* again.

3.1.3 PIP

Each tagged version of vyper is also uploaded to pypi, and can be installed using pip.

```
pip install vyper
```

To install a specific version use:

```
pip install vyper==0.1.0b2
```

3.1.4 Docker

Dockerhub

Vyper can be downloaded as docker image from dockerhub:

```
docker pull ethereum/vyper
```

To run the compiler use the *docker run* command:

```
docker run vyper <contract_file.vy>
```

The normal paramaters are also supported, for example:

```
docker run vyper -f abi a.vy
[{'name': 'test1', 'outputs': [], 'inputs': [{'type': 'uint256', 'name': 'a'}, {'type
↪': 'bytes', 'name': 'b'}], 'constant': False, 'payable': False, 'type': 'function',
↪'gas': 441}, {'name': 'test2', 'outputs': [], 'inputs': [{'type': 'uint256', 'name
↪': 'a'}], 'constant': False, 'payable': False, 'type': 'function', 'gas': 316}]
```

Dockerfile

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/vyper.git -t vyper:1
docker run -it --entrypoint /bin/bash vyper:1
```

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

```
python setup.py test
vyper examples/crowdfund.vy
```

3.1.5 Snap

Vyper is published in the snap store. In any of the supported Linux distros, install it with:

```
sudo snap install vyper --edge --devmode
```

(Note that installing the above snap is the latest master)

3.2 Compiling a Contract

To compile a contract, use:

```
vyper yourFileName.vy
```

You can also compile to other formats such as ABI using the below format:

```
vyper -f ['abi', 'abi_python', 'bytecode', 'bytecode_runtime', 'ir', 'asm']␣
↪yourFileName.vy
```

It is also possible to use the *-f json* option, which is a legacy alias for *-f abi*.

Note: Since .vy is not officially a language supported by any syntax highlighters or linters, it is recommended to name your Vyper file ending with *.vy* 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 Vyper. The online compiler allows you to compile to `bytecode` and/or `LLL`.

Note: While the vyper 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 Testing a Contract

The following example demonstrates how to compile and deploy your vyper contract. It requires `pyethereum>=2.0.0` for the `tester` module

Note: We are working on integration with ethereum-tester, so this example will change.

3.3.1 Testing Using vyper-run Command

To allow quickly testing contracts, Vyper provides a command line tool for instantly executing a function:

```
vyper-run yourFileName.vy "your_function();" -i some_init_param, another_init_param
```

The vyper-run command is composed of 4 parts:

- vyper-run
- the name of the contract file you want to execute (for example: coolContract.vy)
- a string (wrapped in doubble quotes) with the function you want to trigger, you can trigger multiple functions by adding a semicolon at the end of each function and then call the next function (for example: `"my_function1(100,4);my_function2()"`) +
- (Optional) the parameters for the `__init__` function of the contract (for example: given `__init__(a: int128, b: int128)` the syntax would be `-i 8,27`).

Putting it all together:

```
vyper-run myContract.vy "my_function1();my_function2()" -i 1,3
```

The vyper-run command will print out the returned value of the called functions as well as all the logged events emitted during the function's execution.

3.4 Deploying a Contract

You have several options to deploy a Vyper contract to the public testnets.

One option is to take the bytecode generated by the vyper compiler and manually deploy it through mist or geth:

Or deploy it with current browser on myetherwallet contract menu.

We are working on integration with populus, this will be the preferred way of deploying vyper contracts in the future.

3.5 Structure of a Contract

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

3.5.1 State Variables

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

```
storedData: int128
```

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

3.5.2 Functions

Functions are the executable units of code within a contract.

```
@public
@payable
def bid(): // Function
  // ...
```

Function-calls can happen internally or externally and have different levels of visibility (visibility-and-getters) towards other contracts. Functions must be decorated with either @public or @private.

Default function

A contract can also have a default function, which is executed on a call to the contract if no other functions match the given function identifier (or if none was supplied at all, such as through someone sending it Eth). It is the same construct as fallback functions in Solidity.

This function is always named *__default__* and must be annotated with *@public*. It cannot have arguments and cannot return anything.

If the function is annotated as *@payable*, this function is executed whenever the contract is sent Ether (without data). This is why the default function cannot accept arguments and return values - it is a design decision of Ethereum to make no differentiation between sending ether to a contract or a user address.

Example:

```
Payment: event({amount: int128, from: indexed(address)})

@public
@payable
def __default__():
    log.Payment(msg.value, msg.sender)
```

Considerations

Just as in Solidity, Vyper generates a default function if one isn't found, in the form of a REVERT call. Note that this still generates an exception and thus will not succeed in receiving funds.

Ethereum specifies that the operations will be rolled back if the contract runs out of gas in execution. `send` calls to the contract come with a free stipend of 2300 gas, which does not leave much room to perform other operations except basic logging. **However**, if the sender includes a higher gas amount through a `call` instead of `send`, then more complex functionality can be run.

It is considered a best practice to ensure your payable default function is compatible with this stipend. The following operations will consume more than 2300 gas:

- Writing to storage
- Creating a contract
- Calling an external function which consumes a large amount of gas
- Sending Ether

Lastly, although the default function receives no arguments, it can still access the `msg` global, including:

- the address of who is interacting with the contract (`msg.sender`)
- the amount of ETH sent (`msg.value`)
- the gas provided (`msg.gas`).

3.5.3 Events

Events may be logged in specially indexed data structures that allow clients, including light clients, to efficiently search for them.

```
Payment: event({amount: int128, arg2: indexed(address)})

total_paid: int128

@public
@payable
def pay():
    self.total_paid += msg.value
    log.Payment(msg.value, msg.sender)
```

Events must be declared before global declarations and function definitions.

3.6 Vyper by Example

3.6.1 Simple Open Auction

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

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.

```
1 # Open Auction
2
3 # Auction params
4 # Beneficiary receives money from the highest bidder
```

(continues on next page)

(continued from previous page)

```
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`.
@public
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.
@public
@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


# End the auction and send the highest bid
# to the beneficiary.
@public
def end_auction():
    # 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
```

(continues on next page)

(continued from previous page)

```
    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)
```

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 receives 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.

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 Vyper 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.

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 and 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`.

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 Vyper. As we move on to exploring more complex examples, we will encounter more design patterns and features of the Vyper 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.

3.6.2 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
#Originally from
#https://github.com/ethereum/solidity/blob/develop/docs/solidity-by-example.rst
#ported to vyper 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.
```

(continues on next page)

(continued from previous page)

```
#(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)

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

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

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

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

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*.

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`.

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.

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 `True` 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.

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—where 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.

3.6.3 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.

```
# Setup private variables (only callable from within the contract)
funders: {sender: address, value: wei_value}[int128]
nextFunderIndex: int128
beneficiary: address
deadline: timestamp
goal: wei_value
refundIndex: int128
timelimit: timedelta


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

(continues on next page)

(continued from previous page)

```
18
19
20  # Participate in this crowdfunding campaign
21  @public
22  @payable
23  def participate():
24      assert block.timestamp < self.deadline
25
26      nfi: int128 = self.nextFunderIndex
27
28      self.funders[nfi] = {sender: msg.sender, value: msg.value}
29      self.nextFunderIndex = nfi + 1
30
31
32  # Enough money was raised! Send funds to the beneficiary
33  @public
34  def finalize():
35      assert block.timestamp >= self.deadline and self.balance >= self.goal
36
37      selfdestruct(self.beneficiary)
38
39
40  # Not enough money was raised! Refund everyone (max 30 people at a time
41  # to avoid gas limit issues)
42  @public
43  def refund():
44      assert block.timestamp >= self.deadline and self.balance < self.goal
45
46      ind: int128 = self.refundIndex
47
48      for i in range(ind, ind + 30):
49          if i >= self.nextFunderIndex:
50              self.refundIndex = self.nextFunderIndex
51              return
52
53          send(self.funders[i].sender, self.funders[i].value)
54          self.funders[i] = None
55
56      self.refundIndex = ind + 30
```

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

```
# Setup private variables (only callable from within the contract)
funders: {sender: address, value: wei_value}[int128]
nextFunderIndex: int128
beneficiary: address
deadline: timestamp
goal: wei_value
refundIndex: int128
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 the `int128` 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.

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.

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.

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 Vyper 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.

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.

3.6.4 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.

```
# Voting with delegation.

# Information about voters
voters: public({
    # weight is accumulated by delegation
    weight: int128,
```

(continues on next page)

(continued from previous page)

```
    # if true, that person already voted (which includes voting by delegating)
    voted: bool,
    # person delegated to
    delegate: address,
    # index of the voted proposal, which is not meaningful unless `voted` is True.
    vote: int128
}[address])

# This is a type for a list of proposals.
proposals: public({
    # short name (up to 32 bytes)
    name: bytes32,
    # int128ber of accumulated votes
    vote_count: int128
}[int128])

voter_count: public(int128)
chairperson: public(address)
int128_proposals: public(int128)


@public
@constant
def delegated(addr: address) -> bool:
    return self.voters[addr].delegate != ZERO_ADDRESS


@public
@constant
def directly_voted(addr: address) -> bool:
    return self.voters[addr].voted and (self.voters[addr].delegate == ZERO_ADDRESS)


# Setup global variables
@public
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
        }
        self.int128_proposals += 1

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

(continues on next page)

(continued from previous page)

```
64
65  # Used by `delegate` below, and can be called by anyone.
66  @public
67  def forward_weight(delegate_with_weight_to_forward: address):
68      assert self.delegated(delegate_with_weight_to_forward)
69      # Throw if there is nothing to do:
70      assert self.voters[delegate_with_weight_to_forward].weight > 0
71
72      target: address = self.voters[delegate_with_weight_to_forward].delegate
73      for i in range(4):
74          if self.delegated(target):
75              target = self.voters[target].delegate
76              # The following effectively detects cycles of length <= 5,
77              # in which the delegation is given back to the delegator.
78              # This could be done for any int128ber of loops,
79              # or even infinitely with a while loop.
80              # However, cycles aren't actually problematic for correctness;
81              # they just result in spoiled votes.
82              # So, in the production version, this should instead be
83              # the responsibility of the contract's client, and this
84              # check should be removed.
85              assert target != delegate_with_weight_to_forward
86          else:
87              # Weight will be moved to someone who directly voted or
88              # hasn't voted.
89              break
90
91      weight_to_forward: int128 = self.voters[delegate_with_weight_to_forward].weight
92      self.voters[delegate_with_weight_to_forward].weight = 0
93      self.voters[target].weight += weight_to_forward
94
95      if self.directly_voted(target):
96          self.proposals[self.voters[target].vote].vote_count += weight_to_forward
97          self.voters[target].weight = 0
98
99      # To reiterate: if target is also a delegate, this function will need
100     # to be called again, similarly to as above.
101
102 # Delegate your vote to the voter `to`.
103 @public
104 def delegate(to: address):
105     # Throws if the sender has already voted
106     assert not self.voters[msg.sender].voted
107     # Throws if the sender tries to delegate their vote to themselves or to
108     # the default address value of 0x0000000000000000000000000000000000000000
109     # (the latter might not be problematic, but I don't want to think about it).
110     assert to != msg.sender
111     assert to != ZERO_ADDRESS
112
113     self.voters[msg.sender].voted = True
114     self.voters[msg.sender].delegate = to
115
116     # This call will throw if and only if this delegation would cause a loop
117         # of length <= 5 that ends up delegating back to the delegator.
118     self.forward_weight(msg.sender)
119
120 # Give your vote (including votes delegated to you)
```

(continues on next page)

(continued from previous page)

```
# to proposal `proposals[proposal].name`.
@public
def vote(proposal: int128):
    # can't vote twice
    assert not self.voters[msg.sender].voted
    # can only vote on legitimate proposals
    assert proposal < self.int128_proposals

    self.voters[msg.sender].vote = proposal
    self.voters[msg.sender].voted = True

    # transfer msg.sender's weight to proposal
    self.proposals[proposal].vote_count += self.voters[msg.sender].weight
    self.voters[msg.sender].weight = 0

# Computes the winning proposal taking all
# previous votes into account.
@public
@constant
def winning_proposal() -> int128:
    winning_vote_count: int128 = 0
    winning_proposal: int128 = 0
    for i in range(2):
        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
@public
@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: int128,
    # if true, that person already voted (which includes voting by delegating)
    voted: bool,
    # person delegated to
    delegate: address,
    # index of the voted proposal, which is not meaningful unless `voted` is True.
    vote: int128
}[address])

# This is a type for a list of proposals.
proposals: public({
    # short name (up to 32 bytes)
    name: bytes32,
    # int128ber of accumulated votes
    vote_count: int128
}[int128])
```

(continues on next page)

(continued from previous page)

```
voter_count: public(int128)
chairperson: public(address)
int128_proposals: public(int128)
```

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 `int128` 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.

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.

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`.

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.

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.

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.

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.

3.6.5 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.

```
units: {
    currency_value: "Currency Value"
}

# Financial events the contract logs
Transfer: event({_from: indexed(address), _to: indexed(address), _value:␣
↪uint256(currency_value)})
Buy: event({_buyer: indexed(address), _buy_order: uint256(currency_value)})
Sell: event({_seller: indexed(address), _sell_order: uint256(currency_value)})
Pay: event({_vendor: indexed(address), _amount: wei_value})

# Initiate the variables for the company and it's own shares.
company: public(address)
total_shares: public(uint256(currency_value))
price: public(uint256 (wei / currency_value))

# Store a ledger of stockholder holdings.
holdings: uint256(currency_value)[address]

# Set up the company.
@public
def __init__(_company: address, _total_shares: uint256(currency_value),
        initial_price: uint256(wei / currency_value) ):
    assert _total_shares > 0
    assert initial_price > 0

    self.company = _company
    self.total_shares = _total_shares
    self.price = initial_price

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

@public
@constant
def stock_available() -> uint256(currency_value):
    return self.holdings[self.company]
```

(continues on next page)

(continued from previous page)

```
37
38  # Give some value to the company and get stock in return.
39  @public
40  @payable
41  def buy_stock():
42      # Note: full amount is given to company (no fractional shares),
43      #       so be sure to send exact amount to buy shares
44      buy_order: uint256(currency_value) = msg.value / self.price # rounds down
45
46      # Check that there are enough shares to buy.
47      assert self.stock_available() >= buy_order
48
49      # Take the shares off the market and give them to the stockholder.
50      self.holdings[self.company] -= buy_order
51      self.holdings[msg.sender] += buy_order
52
53      # Log the buy event.
54      log.Buy(msg.sender, buy_order)
55
56  # Find out how much stock any address (that's owned by someone) has.
57  @public
58  @constant
59  def get_holding(_stockholder: address) -> uint256(currency_value):
60      return self.holdings[_stockholder]
61
62  # Return the amount the company has on hand in cash.
63  @public
64  @constant
65  def cash() -> wei_value:
66      return self.balance
67
68  # Give stock back to the company and get money back as ETH.
69  @public
70  def sell_stock(sell_order: uint256(currency_value)):
71      assert sell_order > 0 # Otherwise, this would fail at send() below,
72          # due to an OOG error (there would be zero value available for gas).
73      # You can only sell as much stock as you own.
74      assert self.get_holding(msg.sender) >= sell_order
75      # Check that the company can pay you.
76      assert self.cash() >= (sell_order * self.price)
77
78      # Sell the stock, send the proceeds to the user
79      # and put the stock back on the market.
80      self.holdings[msg.sender] -= sell_order
81      self.holdings[self.company] += sell_order
82      send(msg.sender, sell_order * self.price)
83
84      # Log the sell event.
85      log.Sell(msg.sender, sell_order)
86
87  # Transfer stock from one stockholder to another. (Assume that the
88  # receiver is given some compensation, but this is not enforced.)
89  @public
90  def transfer_stock(receiver: address, transfer_order: uint256(currency_value)):
91      assert transfer_order > 0 # This is similar to sell_stock above.
92      # Similarly, you can only trade as much stock as you own.
93      assert self.get_holding(msg.sender) >= transfer_order
```

(continues on next page)

(continued from previous page)

```
94
95      # Debit the sender's stock and add to the receiver's address.
96      self.holdings[msg.sender] -= transfer_order
97      self.holdings[receiver] += transfer_order
98
99      # Log the transfer event.
100     log.Transfer(msg.sender, receiver, transfer_order)
101
102 # Allow the company to pay someone for services rendered.
103 @public
104 def pay_bill(vendor: address, amount: wei_value):
105     # Only the company can pay people.
106     assert msg.sender == self.company
107     # Also, it can pay only if there's enough to pay them with.
108     assert self.cash() >= amount
109
110     # Pay the bill!
111     send(vendor, amount)
112
113     # Log the payment event.
114     log.Pay(vendor, amount)
115
116 # Return the amount in wei that a company has raised in stock offerings.
117 @public
118 @constant
119 def debt() -> wei_value:
120     return (self.total_shares - self.holdings[self.company]) * self.price
121
122 # Return the cash holdings minus the debt of the company.
123 # The share debt or liability only is included here,
124 # but of course all other liabilities can be included.
125 @public
126 @constant
127 def worth() -> wei_value:
128     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. We first declare several events that the contract logs. We then declare our global variables, followed by function defintions.

```
Buy: event({_buyer: indexed(address), _buy_order: uint256(currency_value)})
Sell: event({_seller: indexed(address), _sell_order: uint256(currency_value)})
Pay: event({_vendor: indexed(address), _amount: wei_value})

# Initiate the variables for the company and it's own shares.
company: public(address)
total_shares: public(uint256(currency_value))
```

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.

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.

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 Vyper is a statically typed language, we see an arrow following the definition of the `stock_available()` method, which simply represents the data type 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.

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?

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

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

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.

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.

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.

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.

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 Vyper language has convinced you of its capabilities and readability in writing smart contracts.

3.7 Event Logging

Like Solidity and other EVM languages, Vyper can log events to be caught and displayed by user interfaces.

3.7.1 Example of Logging

This example is taken from the sample ERC20 contract and shows the basic flow of event logging.

```
# Events of the token.
Transfer: event({_from: indexed(address), _to: indexed(address), _value: num256})
Approval: event({_owner: indexed(address), _spender: indexed(address), _value: num256}
↪)
```

(continues on next page)

(continued from previous page)

```
# Transfer some tokens from message sender to another address
def transfer(_to : address, _value : num256) -> bool:

    ... Logic here to do the real work ...

    # All done, log the event for listeners
    log.Transfer(msg.sender, _to, _amount)
```

Let's look at what this is doing. First, we declare two event types to log. The two events are similar in that they contain two indexed address fields. Indexed fields do not make up part of the event data itself, but can be searched by clients that want to catch the event. Also, each event contains one single data field, in each case called _value. Events can contain several arguments with any names desired.

Next, in the `transfer` function, after we do whatever work is necessary, we log the event. We pass three arguments, corresponding with the three arguments of the Transfer event declaration.

Clients listening to the events will declare and handle the events they are interested in using a library such as web3.js:

```
var abi = /* abi as generated by the compiler */;
var MyToken = web3.eth.contract(abi);
var myToken = MyToken.at("0x1234...ab67" /* address */);

// watch for changes in the callback
var event = myToken.Transfer(function(error, result) {
    if (!error) {
        var args = result.args;
        console.log('value transferred = ', args._amount);
    }
});
```

In this example, the listening client declares the event to listen for. Any time the contract sends this log event, the callback will be invoked.

3.7.2 Declaring Events

Let's look at an event declaration in more detail.

```
Transfer: event({_from: indexed(address), _to: indexed(address), _value: num256})
```

Event declarations look like state variable declarations but use the special keyword event. event takes a as its argument that consist of all the arguments to be passed as part of the event. Typical events will contain two kinds of arguments:

- Indexed arguments, which can be searched for by listeners. Each indexed argument is identifier by the *indexed* keyword. Here, each indexed argument is an address. You can have any number of indexed arguments, but indexed arguments are not passed directly to listeners, although some of this information (such as the sender) may be available in the listener's *results* object.
- Value arguments, which are passed through to listeners. You can have any number of value arguments and they can have arbitrary names, but each is limited by the EVM to be no more than 32 bytes.

Note that while the argument definition syntax looks like a Python dictionary, it's actually an order-sensitive definition. (Python dictionaries maintain order starting with 3.7.) Thus, the first element (_from) will be matched up with the first argument passed in the log.Transfer call.

3.7.3 Logging Events

Once an event is declared, you can log (send) events. You can send events as many times as you want to. Please note that events sent do not take state storage and thus do not cost gas: this makes events a good way to save some information. However, the drawback is that events are not available to contracts, only to clients.

Logging events is done using the magic keyword *log*:

```
log.Transfer(msg.sender, _to, _amount)
```

The order and types of arguments sent needs to match up with the order of declarations in the dictionary.

3.7.4 Listening for Events

In the example listener above, the *result* arg actually passes a large amount of information. Here we're most interested in *result.args*. This is an object with properties that match the properties declared in the event. Note that this object does not contain the indexed properties, which can only be searched in the original *myToken.Transfer* that created the callback.

3.8 Contributing

Help is always appreciated!

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

3.8.1 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 Vyper Gitter
- Suggesting Improvements
- Fixing and responding to Vyper's GitHub issues

3.8.2 How to Suggest Improvements

To suggest an improvement, please create a Vyper Improvement Proposal (VIP for short) using the VIP Template.

3.8.3 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 Vyper 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.

3.8.4 Fix Bugs

Find or report bugs at our issues page. Anything tagged with "bug" is open to whoever wants to implement it.

3.8.5 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

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

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

Although 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!

3.9 Frequently Asked Questions

3.9.1 Basic Questions

What is Vyper?

Vyper is a smart contract development language. Vyper aims to be auditable, secure, and human-readable. Being simple to read is more important than being simple to write.

Vyper or Solidity?

For the majority of use-cases, this is personal preference. To support the aims of being secure, auditable, and human-readable, a number of programming constructs included in Solidity are not included in Vyper. If your use-case requires these, use Solidity not Vyper.

What is not included in Vyper?

The following constructs are not included because their use can lead to misleading or difficult to understand code:

- Modifiers
- Class inheritance
- Inline assembly

- Function overloading
- Operator overloading
- Binary fixed point.

Recursive calling and infinite-length loops are not included because they cannot set an upper bound on gas limits. An upper bound is required to prevent gas limit attacks and ensure the security of smart contracts built in Vyper.

How do for loops work?

Like Python for loops but with one significant difference. Vyper does not allow looping over variable lengths. Looping over variables introduces the possibility of infinite-length loops which make gas limit attacks possible.

How do structs work?

Structs group variables and are accessed using `struct.argname`. They are similar to Python dictionaries:

```
# define the struct
struct: {
 arg1: int128, arg2: decimal
}

#access arg1 in struct
struct.arg1 = 1
```

3.9.2 Advanced Questions

3.10 Built in Functions

Vyper contains a set amount of built in functions that would be timely and/or unachievable to write in Vyper.

3.10.1 Functions

floor

```
def floor(a) -> b:
  """
  :param a: value to round down
  :type a: decimal

  :output b: int128
  """
```

Rounds a decimal down to the nearest integer.

ceil

```
def ceil(a) -> b:
  """
  :param a: value to round up
  :type a: decimal

  :output b: int128
  """
```

Rounds a decimal up to the nearest integer.

convert

```
def convert(a, b) -> c:
  """
  :param a: value to convert
  :type a: either decimal, int128, uint256 or bytes32
  :param b: the destination type to convert to
  :type b: str_literal

  :output c: either decimal, int128, uint256 or bytes32
  """
```

Converts a variable/ literal from one type to another.

as_wei_value

```
def as_wei_value(a, b) -> c:
  """
  :param a: value for the ether unit
  :type a: uint256 or int128 or decimal
  :param b: ether unit name (e.g. ``"wei"``)
  :type b: str_literal

  :output c: wei_value
  """
```

The value of the input number as `wei`, converted based on the specified unit.

as_unitless_number

```
def as_unitless_number(a) -> b:
  """
  :param a: value to remove units from
  :type a: either decimal or int128

  :output b: either decimal or int128
  """
```

Turns a `int128`, `uint256`, `decimal` with units into one without units (used for assignment and math).

slice

```
def slice(a, start=b, len=c) -> d:
  """
  :param a: bytes to be sliced
  :type a: either bytes or bytes32
  :param b: start position of the slice
  :type b: int128
  :param c: length of the slice
  :type c: int128

  :output d: bytes
  """
```

Takes a list of bytes and copies, then returns a specified chunk.

len

```
def len(a) -> b:
  """
  :param a: value to get the length of
  :type a: bytes

  :output b: int128
  """
```

Returns the length of a given list of bytes.

concat

```
def concat(a, b, ...) -> c:
  """
  :param a: value to combine
  :type a: bytes
  :param b: value to combine
  :type b: bytes

  :output b: bytes
  """
```

Takes 2 or more bytes arrays of type `bytes32` or `bytes` and combines them into one.

sha3/ keccak256

```
def sha3(a) -> b:
  """
  :param a: value to hash
  :type a: either str_literal, bytes, bytes32

  :output b: bytes32
  """
```

Returns keccak256``(Ethereum's sha3) hash of input. Note that it can be called either by using ``sha3 or keccak256.

method_id

```
def method_id(a) -> b:
  """
  :param a: method declaration
  :type a: str_literal

  :output b: bytes
  """
```

Takes a function declaration and returns its method_id (used in data field to call it).

ecrecover

```
def ecrecover(hash, v, r, s) -> b:
  """
  :param hash: a signed hash
  :type hash: bytes32
  :param v:
  :type v: uint256
  :param r: elliptic curve point
  :type r: uint256
  :param s: elliptic curve point
  :type s: uint256

  :output b: address
  """
```

Takes a signed hash and vrs and returns the public key of the signer.

ecadd

```
def ecadd(a, b) -> sum:
  """
  :param a: pair to be added
  :type a: num252[2]
  :param b: pair to be added
  :type b: num252[2]

  :output sum: uint256[2]
  """
```

Takes two elliptical curves and adds them together.

ecmul

```
def ecmul(a, b) -> product:
  """
  :param a: pair to be multiplied
  :type a: num252[2]
  :param b: pair to be multiplied
  :type b: num252[2]
```

(continues on next page)

(continued from previous page)

```
  :output product: uint256[2]
  """
```

Takes two elliptical curves and multiplies them together.

extract32

```
def extract32(a, b, type=c) -> d:
  """
  :param a: where 32 bytes are extracted from
  :type a: bytes
  :param b: start point of bytes to be extracted
  :type b: int128
  :param c: type of output
  :type c: either bytes32, int128, or address

  :output d: either bytes32, int128, or address
  """
```

RLPList

```
def _RLPList(a, b) -> c:
  """
  :param a: encoded data
  :type a: bytes
  :param b: RLP list
  :type b: list

  :output c: LLLnode
  """
```

Takes encoded RLP data and an unencoded list of types. Usage:

```
vote_msg: bytes <= 1024 = ...

values = RLPList(vote_msg, [int128, int128, bytes32, bytes, bytes])

var1: int128 = values[0]
var2: int128 = values[1]
var3: bytes32 = values[2]
var4: bytes <= 1024 = values[3]
var5: bytes <= 1024 = values[4]
```

Note: RLP decoder needs to be deployed if one wishes to use it outside of the Vyper test suite. Eventually, the decoder will be available on mainnet at a fixed address. But for now, here's how to create RLP decoder on other chains:

1. send 6270960000000000 wei to 0xd2c560282c9C02465C2dAcdEF3E859E730848761

2. Publish this tx to create the contract:

```
0xf90237808506fc23ac00830330888080b902246102128061000e60003961022056600060007f010000000000000000000000
```

3. This is the contract address: 0xCb969cAAad21A78a24083164ffa81604317Ab603

3.11 Low Level Built in Functions

Vyper contains a set of built in functions which executes unique OPCODES such as send or selfdestruct.

3.11.1 Low Level Functions

send

```
def send(a, b):
  """
  :param a: the destination address to send ether to
  :type a: address
  :param b: the wei value to send to the address
  :type b: uint256
  """
```

Sends ether from the contract to the specified Ethereum address. Note that the amount to send should be specified in wei.

raw_call

```
def raw_call(a, b, outsize=c, gas=d, value=e) -> f:
  """
  :param a: the destination address to call to
  :type a: address
  :param b: the data to send the called address
  :type b: bytes
  :param c: the max-length for the bytes array returned from the call.
  :type c: fixed literal value
  :param d: the gas amount to attach to the call.
  :type d: uint256
  :param e: the wei value to send to the address (Optional)
  :type e: uint256

  :output f: bytes[outsize]
  """
```

Calls to the specified Ethereum address. The call should pass data and may optionaly send eth value (specified in wei) as well. The call must specify a gas amount to attach the call and and the outsize. Returns the data returned by the call as a bytes array with the outsize as the max length.

selfdestruct

```
def selfdestruct(a):
  """
  :param a: the address to send the contracts left ether to
  :type a: address
  """
```

Causes a self destruction of the contract, triggers the `SELFDESTRUCT` opcode (0xff). CAUTION! This method will delete the contract from the Ethereum blockchain. All none ether assets associated with this contract will be "burned" and the contract will be inaccessible.

assert

```
def assert(a):
  """
  :param a: the boolean condition to assert
  :type a: bool
  """
```

Asserts the specified condition, if the condition is equals to true the code will continue to run. Otherwise, the OPCODE `REVERT` (0xfd) will be triggered, the code will stop it's operation, the contract's state will be reverted to the state before the transaction took place and the remaining gas will be returned to the transaction's sender.

Note: To give it a more Python like syntax, the assert function can be called without parenthesis, the syntax would be `assert your_bool_condition`. Even though both options will compile, it's recommended to use the Pythonic version without parenthesis.

raw_log

```
def raw_log(a, b):
  """
  :param a: the address of the contract to duplicate.
  :type a: * (any input)
  :param b: the name of the logged event.
  :type b: bytes
  """
```

Emits a log without specifying the abi type, with the arguments entered as the first input.

create_with_code_of

```
def create_with_code_of(a, value=b):
  """
  :param a: the address of the contract to duplicate.
  :type a: address
  :param b: the wei value to send to the new contract instance
  :type b: uint256 (Optional)
  """
```

Duplicates a contract's code and deploys it as a new instance. You can also specify wei value to send to the new contract as `value=the_value`.

blockhash

```
def blockhash(a) -> hash:
  """
  :param a: the number of the block to get
  :type a: uint256

  :output hash: bytes32
  """
```

Returns the hash of the block at the specified height.

Note: The EVM only provides access to the most 256 blocks. This function will return 0 if the block number is greater than or equal to the current block number or more than 256 blocks behind the current block.

3.12 Types

Vyper 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. Vyper provides several elementary types which can be combined to form complex types.

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

3.12.1 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.

Boolean

Keyword: `bool`

A boolean is a type to store a logical/truth value.

Values

The only possible values are the constants `True` and `False`.

Operators

Operator	Description
`x not y`	Logical negation
`x and y`	Logical conjunction
`x or y`	Logical disjunction
`x == y`	Equality
`x != y`	Inequality

The operators `or` and `and` apply the common short-circuiting rules.

Signed Integer (128 bit)

Keyword: `int128`

A signed integer (128 bit) is a type to store positive and negative integers.

Values

Signed integer values between -2^{127} and $(2^{127} - 1)$, inclusive.

Operators

Comparisons

Comparisons return a boolean value.

Operator	Description
`x < y`	Less than
`x <= y`	Less than or equal to
`x == y`	Equals
`x != y`	Does not equal
`x >= y`	Greater than or equal to
`x > y`	Greater than

`x` and `y` must be of the type `int128`.

Arithmetic Operators

Operator	Description
`x + y`	Addition
`x - y`	Subtraction
`-x`	Unary minus/Negation
`x * y`	Multiplication
`x / y`	Division
`x**y`	Exponentiation
`x % y`	Modulo
`min(x, y)`	Minimum
`max(x, y)`	Maximum

`x` and `y` must be of the type `int128`.

Unsigned Integer (256 bit)

Keyword: `uint256`

An unsigned integer (256 bit) is a type to store non-negative integers.

Values

Integer values between 0 and (2^{256}-1).

Note: Integer literals are interpreted as `int128` by default. In cases where `uint256` is more appropriate, such as assignment, the literal might be interpreted as `uint256`. Example: `_variable: uint256 = _literal`. In order to explicitly cast a literal to a `uint256` use `convert(_literal, 'uint256')`.

Operators

Comparisons

Comparisons return a boolean value.

`x` and `y` must be of the type `uint256`.

Arithmetic Operators

`x` and `y` must be of the type `uint256`.

Bitwise Operators

Operator	Description
`bitwise_and(x, y)`	AND
`bitwise_not(x, y)`	NOT
`bitwise_or(x, y)`	OR
`bitwise_xor(x, y)`	XOR
`shift(x, _shift)`	Bitwise Shift

`x` and `y` must be of the type `uint256`. `_shift` must be of the type `int128`.

Note: Positive `_shift` equals a left shift; negative `_shift` equals a right shift. Values shifted above/below the most/least significant bit get discarded.

Decimals

Keyword: `decimal`

A decimal is a type to store a decimal fixed point value.

Values

A value with a precision of 10 decimal places between -2^{127} and $(2^{127} - 1)$.

Operators

Comparisons

Comparisons return a boolean value.

Operator	Description
`x < y`	Less than
`x <= y`	Less or equal
`x == y`	Equals
`x != y`	Does not equal
`x >= y`	Greater or equal
`x > y`	Greater than

`x` and `y` must be of the type `decimal`.

Arithmetic Operators

`x` and `y` must be of the type `decimal`.

Address

Keyword: `address`

The address type holds an Ethereum address.

Values

An address type can hold an Ethereum address which equates to 20 bytes or 160 bits. It returns in hexadecimal notation with a leading `0x`.

Members

Member	Description
`balance`	Query the balance of an address. Returns `wei_value`.
`codesize`	Query the code size of an address. Returns `int128`.

Syntax as follows: `_address.<member>`, where `_address` is of the type `address` and `<member>` is one of the above keywords.

Unit Types

Vyper allows the definition of types with discrete units e.g. meters, seconds, wei, ... These types may only be based on either `uint256`, `int128` or `decimal`. Vyper has 3 unit types built in, which are the following:

Note: Two `timedelta` can be added together, as can a `timedelta` and a `timestamp`, but not two `timestamps`.

Custom Unit Types

Vyper allows you to add additional not-provided unit label to either `uint256`, `int128` or `decimal`.

Custom units example:

```
# specify units used in the contract.
units: {
    cm: "centimeter",
    km: "kilometer"
}
```

Having defined the units they can be defined on variables as follows.

Custom units usage:

```
a: int128(cm)
b: uint256(km)
```

32-bit-wide Byte Array

Keyword: `bytes32` This is a 32-bit-wide byte array that is otherwise similiar to byte arrays.

Example:

```
# Declaration
hash: bytes32
# Assignment
self.hash = _hash
```

Operators

Keyword	Description
`len(x)`	Return the length as an integer.
`sha3(x)`	Return the sha3 hash as bytes32.
`concat(x, ...)`	Concatenate multiple inputs.
`slice(x, start=_start, len=_len)`	Return a slice of `_len` starting at `_start`.

Where `x` is a byte array and `_start` as well as `_len` are integer values.

Fixed-size Byte Arrays

Keyword: `bytes`

A byte array with a fixed size. The syntax being `bytes[maxLen]`, where `maxLen` is an integer which denotes the maximum number of bytes.

Strings

Fixed-size byte arrays can hold strings with equal or fewer characters than the maximum length of the byte array.

Example:

```
exampleString = "Test String"
```

Operators

Keyword	Description
`len(x)`	Return the length as an integer.
`sha3(x)`	Return the sha3 hash as bytes32.
`concat(x, ...)`	Concatenate multiple inputs.
`slice(x, start=_start, len=_len)`	Return a slice of `_len` starting at `_start`.

Where `x` is a byte array while `_start` and `_len` are integers.

3.12.2 Reference Types

Reference types do not fit into 32 bytes. Because of this, copying their value is not as feasible as with value types. Therefore only the location, i.e. the reference, of the data is passed.

Fixed-size Lists

Fixed-size lists hold a finite number of elements which belong to a specified type.

Syntax

Lists can be declared with `_name: _ValueType[_Integer]`. Multidimensional lists are also possible.

Example:

```
#Defining a list
exampleList: int128[3]
#Setting values
exampleList = [10, 11, 12]
exampleList[2] = 42
#Returning a value
return exampleList[0]
```

Structs

Structs are custom defined types that can group several variables.

Syntax

Structs can be accessed via `struct.argname`. **Example:**

```
#Defining a struct
exampleStruct: {
    value1: int128,
    value2: decimal
```

(continues on next page)

(continued from previous page)

```
}
#Accessing a value
exampleStruct.value1 = 1
```

Mappings

Mappings in Vyper 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".

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

Note: Mappings are only allowed as state variables.

Syntax

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.

Example:

```
#Defining a mapping
exampleMapping: decimal[int128]
#Accessing a value
exampleMapping[0] = 10.1
```

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

3.12.3 Builtin Constants

Vyper has a few convenience constants builtin.

Type	Name	Value
address	ZERO_ADDRESS	0x0000000000000000000000000000000000000000
int128	MAX_INT128	2**127 - 1
int128	MIN_INT128	-2**127
decimal	MAX_DECIMAL	(2**127 - 1)
decimal	MIN_DECIMAL	(-2**127)
uint256	MAX_UINT256	2**256 - 1

3.12.4 Initial Values and None

In Vyper, there is no `null` option like most programing languages have. Thus, every variable type has a default value. Nevertheless Vyper has the option to declare `None` which represent the default value of the type. Note that there is

no option to assign `None` when initiating a variable. Also, note that you can't make comparisons to None. In order to check if a variable is empty, you will need to compare it to its type's default value.

Here you can find a list of all types and default values:

Table 1: Default Variable Values

Type	Default Value
`bool`	`False`
`int128`	`0`
`uint256`	`0`
`decimal`	`0.0`
`address`	`0x0000000000000000000000000000000000000000`
`bytes32`	`\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00`

Note: In `bytes` the array starts with the bytes all set to `'\x00'`

Note: In reference types all the type's members are set to their initial values.

Index