Vyper Documentation

Vyper Team (originally created by Vitalik Buterin)

Oct 10, 2020

Contents

CHAPTER 1

Vyper

Vyper is a contract-oriented, pythonic programming language that targets the Ethereum Virtual Machine (EVM).

1.1 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 provides the following features:

- **Bounds and overflow checking**: On array accesses and arithmetic.
- **Support for signed integers and decimal fixed point numbers**
- **Decidability**: It is possible to compute a precise upper bound for the gas consumption of any Vyper function call.
- **Strong typing**
- **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 your 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 that are not visible at a 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 recursive 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.

Vyper **does not** strive to be a 100% replacement for everything that can be done in Solidity; it will deliberately forbid things or make things harder if it deems fit to do so for the goal of increasing security.

CHAPTER 2

Installing Vyper

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

Note: The easiest way to experiment with the language is to use the Remix online compiler.

2.1 Docker

Vyper can be downloaded as docker image from dockerhub:

```
docker pull vyperlang/vyper
```

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

```
docker run -v $(pwd):/code vyperlang/vyper /code/<contract_file.vy>
```

Alternatively you can log into the docker image and execute vyper on the prompt.

```
docker run -v $(pwd):/code/ -it --entrypoint /bin/bash vyperlang/vyper
root@d35252d1fb1b:/code# vyper <contract_file.vy>
```

The normal paramaters are also supported, for example:

```
docker run -v $(pwd):/code vyperlang/vyper -f abi /code/<contract_file.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}]
```

Note: If you would like to know how to install Docker, please follow their documentation.

2.2 PIP

2.2.1 Installing Python

Vyper can only be built using Python 3.6 and higher. If you need to know how to install the correct version of python, follow the instructions from the official Python website.

2.2.2 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. For easy virtualenv management, we recommend either pyenv or Poetry.

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

2.2.3 Installing Vyper

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

```
pip install vyper
```

To install a specific version use:

```
pip install vyper==0.1.0b17
```

CHAPTER 3

Vyper by Example

3.1 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.

```
# Open Auction

# Auction params
# Beneficiary receives money from the highest bidder
beneficiary: public(address)
auctionStart: public(uint256)
auctionEnd: public(uint256)

# Current state of auction
highestBidder: public(address)
highestBid: public(uint256)

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

# Keep track of refunded bids so we can follow the withdraw pattern
pendingReturns: public(HashMap[address, uint256])

# Create a simple auction with `_bidding_time`
# seconds bidding time on behalf of the
# beneficiary address `_beneficiary`.
@external
def __init__(_beneficiary: address, _bidding_time: uint256):
```

(continues on next page)

(continued from previous page)

```
    self.beneficiary = _beneficiary
    self.auctionStart = block.timestamp
    self.auctionEnd = self.auctionStart + _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.
@external
@payable
def bid():
    # Check if bidding period is over.
    assert block.timestamp < self.auctionEnd
    # Check if bid is high enough
    assert msg.value > self.highestBid
    # Track the refund for the previous high bidder
    self.pendingReturns[self.highestBidder] += self.highestBid
    # Track new high bid
    self.highestBidder = msg.sender
    self.highestBid = msg.value

# Withdraw a previously refunded bid. The withdraw pattern is
# used here to avoid a security issue. If refunds were directly
# sent as part of bid(), a malicious bidding contract could block
# those refunds and thus block new higher bids from coming in.
@external
def withdraw():
    pending_amount: uint256 = self.pendingReturns[msg.sender]
    self.pendingReturns[msg.sender] = 0
    send(msg.sender, pending_amount)

# End the auction and send the highest bid
# to the beneficiary.
@external
def endAuction():
    # 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.auctionEnd
    # Check if this function has already been called
    assert not self.ended

    # 2. Effects
    self.ended = True

```

(continues on next page)

(continued from previous page)

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

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)
auctionStart: public(uint256)
auctionEnd: public(uint256)

# Current state of auction
highestBidder: public(address)
highestBid: public(uint256)

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

# Keep track of refunded bids so we can follow the withdraw pattern
pendingReturns: public(HashMap[address, uint256])
```

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 `auctionStart` and `auctionEnd` with the datatype `uint256` to manage the open auction period and `highestBid` with datatype `uint256`, the smallest denomination of ether, to manage auction state. The variable `ended` is a boolean to determine whether the auction is officially over. The variable `pendingReturns` is a `map` which enables the use of key-value pairs to keep proper track of the auctions withdrawal pattern.

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 through an external call such as `contract.beneficiary()`.

Now, the constructor.

```
@external
def __init__(_beneficiary: address, _bidding_time: uint256):
    self.beneficiary = _beneficiary
    self.auctionStart = block.timestamp
    self.auctionEnd = self.auctionStart + _bidding_time
```

The contract is initialized with two arguments: `_beneficiary` of type `address` and `_bidding_time` with type `uint256`, 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.auctionEnd`. 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.

```
@external
@payable
def bid():
```

(continues on next page)

(continued from previous page)

```
    # Check if bidding period is over.
    assert block.timestamp < self.auctionEnd
    # Check if bid is high enough
    assert msg.value > self.highestBid
    # Track the refund for the previous high bidder
    self.pendingReturns[self.highestBidder] += self.highestBid
    # Track new high bid
    self.highestBidder = msg.sender
    self.highestBid = 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`.

Note: `msg.sender` and `msg.value` can only be accessed from external functions. If you require these values within an internal function they must be passed as parameters.

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 `highestBid` to the previous `highestBidder` and set our new `highestBid` and `highestBidder`.

```
@external
def endAuction():
    # 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.auctionEnd
    # Check if this function has already been called
    assert not self.ended

    # 2. Effects
    self.ended = True

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

With the `endAuction()` method, we check whether our current time is past the `auctionEnd` 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 into 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.

3.2 Blind Auction

Before we dive into our other examples, let's briefly explore another type of auction that you can build with Vyper. Similar to its counterpart written in Solidity, this blind auction allows for an auction where there is no time pressure towards the end of the bidding period.

```
# Blind Auction # Adapted to Vyper from [Solidity by Example](https://github.com/
↪ethereum/solidity/blob/develop/docs/solidity-by-example.rst#blind-auction-1)

struct Bid:
  blindedBid: bytes32
  deposit: uint256

# Note: because Vyper does not allow for dynamic arrays, we have limited the
# number of bids that can be placed by one address to 128 in this example
MAX_BIDS: constant(int128) = 128

# Event for logging that auction has ended
event AuctionEnded:
    highestBidder: address
    highestBid: uint256

# Auction parameters
beneficiary: public(address)
biddingEnd: public(uint256)
revealEnd: public(uint256)

# Set to true at the end of auction, disallowing any new bids
ended: public(bool)

# Final auction state
highestBid: public(uint256)
highestBidder: public(address)

# State of the bids
bids: HashMap[address, Bid[128]]
bidCounts: HashMap[address, int128]

```

(continues on next page)

(continued from previous page)

```
# Allowed withdrawals of previous bids
pendingReturns: HashMap[address, uint256]


# Create a blinded auction with `_biddingTime` seconds bidding time and
# `_revealTime` seconds reveal time on behalf of the beneficiary address
# `_beneficiary`.
@external
def __init__(_beneficiary: address, _biddingTime: uint256, _revealTime: uint256):
    self.beneficiary = _beneficiary
    self.biddingEnd = block.timestamp + _biddingTime
    self.revealEnd = self.biddingEnd + _revealTime


# Place a blinded bid with:
#
# _blindedBid = keccak256(concat(
#       convert(value, bytes32),
#       convert(fake, bytes32),
#       secret)
# )
#
# The sent ether is only refunded if the bid is correctly revealed in the
# revealing phase. The bid is valid if the ether sent together with the bid is
# at least "value" and "fake" is not true. Setting "fake" to true and sending
# not the exact amount are ways to hide the real bid but still make the
# required deposit. The same address can place multiple bids.
@external
@payable
def bid(_blindedBid: bytes32):
    # Check if bidding period is still open
    assert block.timestamp < self.biddingEnd

    # Check that payer hasn't already placed maximum number of bids
    numBids: int128 = self.bidCounts[msg.sender]
    assert numBids < MAX_BIDS

    # Add bid to mapping of all bids
    self.bids[msg.sender][numBids] = Bid({
        blindedBid: _blindedBid,
        deposit: msg.value
        })
    self.bidCounts[msg.sender] += 1


# Returns a boolean value, `True` if bid placed successfully, `False` otherwise.
@internal
def placeBid(bidder: address, _value: uint256) -> bool:
    # If bid is less than highest bid, bid fails
    if (_value <= self.highestBid):
        return False

    # Refund the previously highest bidder
    if (self.highestBidder != ZERO_ADDRESS):
        self.pendingReturns[self.highestBidder] += self.highestBid

    # Place bid successfully and update auction state
```

(continues on next page)

(continued from previous page)

```
    self.highestBid = _value
    self.highestBidder = bidder

    return True


# Reveal your blinded bids. You will get a refund for all correctly blinded
# invalid bids and for all bids except for the totally highest.
@external
def reveal(_numBids: int128, _values: uint256[128], _fakes: bool[128], _secrets: bytes32[128]):
    # Check that bidding period is over
    assert block.timestamp > self.biddingEnd

    # Check that reveal end has not passed
    assert block.timestamp < self.revealEnd

    # Check that number of bids being revealed matches log for sender
    assert _numBids == self.bidCounts[msg.sender]

    # Calculate refund for sender
    refund: uint256 = 0
    for i in range(MAX_BIDS):
        # Note that loop may break sooner than 128 iterations if i >= _numBids
        if (i >= _numBids):
            break

        # Get bid to check
        bidToCheck: Bid = (self.bids[msg.sender])[i]

        # Check against encoded packet
        value: uint256 = _values[i]
        fake: bool = _fakes[i]
        secret: bytes32 = _secrets[i]
        blindedBid: bytes32 = keccak256(concat(
            convert(value, bytes32),
            convert(fake, bytes32),
            secret
        ))

        # Bid was not actually revealed
        # Do not refund deposit
        if (blindedBid != bidToCheck.blindedBid):
            assert 1 == 0
            continue

        # Add deposit to refund if bid was indeed revealed
        refund += bidToCheck.deposit
        if (not fake and bidToCheck.deposit >= value):
            if (self.placeBid(msg.sender, value)):
                refund -= value

        # Make it impossible for the sender to re-claim the same deposit
        zeroBytes32: bytes32 = EMPTY_BYTES32
        bidToCheck.blindedBid = zeroBytes32

    # Send refund if non-zero
```

(continues on next page)

(continued from previous page)

```
    if (refund != 0):
        send(msg.sender, refund)


# Withdraw a bid that was overbid.
@external
def withdraw():
    # Check that there is an allowed pending return.
    pendingAmount: uint256 = self.pendingReturns[msg.sender]
    if (pendingAmount > 0):
        # If so, set pending returns to zero to prevent recipient from calling
        # this function again as part of the receiving call before `transfer`
        # returns (see the remark above about conditions -> effects ->
        # interaction).
        self.pendingReturns[msg.sender] = 0

        # Then send return
        send(msg.sender, pendingAmount)


# End the auction and send the highest bid to the beneficiary.
@external
def auctionEnd():
    # Check that reveal end has passed
    assert block.timestamp > self.revealEnd

    # Check that auction has not already been marked as ended
    assert not self.ended

    # Log auction ending and set flag
    log AuctionEnded(self.highestBidder, self.highestBid)
    self.ended = True

    # Transfer funds to beneficiary
    send(self.beneficiary, self.highestBid)
```

While this blind auction is almost functionally identical to the blind auction implemented in Solidity, the differences in their implementations help illustrate the differences between Solidity and Vyper.

```
# State of the bids
bids: HashMap[address, Bid[128]]
bidCounts: HashMap[address, int128]
```

One key difference is that, because Vyper does not allow for dynamic arrays, we have limited the number of bids that can be placed by one address to 128 in this example. Bidders who want to make more than this maximum number of bids would need to do so from multiple addresses.

3.3 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.
#    (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(uint256) #Value of the item
seller: public(address)
buyer: public(address)
unlocked: public(bool)
ended: public(bool)

@external
@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

@external
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.

@external
@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

@external
```

(continues on next page)

(continued from previous page)

```
def received():
    # 1. Conditions
    assert not self.unlocked # Is the item already purchased and pending
                             # confirmation from the buyer?
    assert msg.sender == self.buyer
    assert not self.ended

    # 2. Effects
    self.ended = True

    # 3. Interaction
    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(uint256) #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 data types. Remember that the `public` function allows the variables to be *readable* by an external caller, but not *writeable*.

```
@external
@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
```

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`.

```
@external
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.
```

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.

```
@external
@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
```

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.

```
@external
def received():
    # 1. Conditions
    assert not self.unlocked # Is the item already purchased and pending
                             # confirmation from the buyer?
    assert msg.sender == self.buyer
    assert not self.ended

    # 2. Effects
    self.ended = True

    # 3. Interaction
    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.
```

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.

3.4 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)

struct Funder :
  sender: address
  value: uint256

funders: HashMap[int128, Funder]
```

(continues on next page)

(continued from previous page)

```
nextFunderIndex: int128
beneficiary: address
deadline: public(uint256)
goal: public(uint256)
refundIndex: int128
timelimit: public(uint256)


# Setup global variables
@external
def __init__(_beneficiary: address, _goal: uint256, _timelimit: uint256):
    self.beneficiary = _beneficiary
    self.deadline = block.timestamp + _timelimit
    self.timelimit = _timelimit
    self.goal = _goal


# Participate in this crowdfunding campaign
@external
@payable
def participate():
    assert block.timestamp < self.deadline, "deadline not met (yet)"

    nfi: int128 = self.nextFunderIndex

    self.funders[nfi] = Funder({sender: msg.sender, value: msg.value})
    self.nextFunderIndex = nfi + 1


# Enough money was raised! Send funds to the beneficiary
@external
def finalize():
    assert block.timestamp >= self.deadline, "deadline not met (yet)"
    assert self.balance >= self.goal, "invalid balance"

    selfdestruct(self.beneficiary)

# Not enough money was raised! Refund everyone (max 30 people at a time
# to avoid gas limit issues)
@external
def refund():
    assert block.timestamp >= self.deadline and self.balance < self.goal

    ind: int128 = 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] = empty(Funder)

    self.refundIndex = ind + 30
```

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

```
struct Funder :
  sender: address
  value: uint256

funders: HashMap[int128, Funder]
nextFunderIndex: int128
beneficiary: address
deadline: public(uint256)
goal: public(uint256)
refundIndex: int128
timelimit: public(uint256)
```

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.

```
@external
def __init__(_beneficiary: address, _goal: uint256, _timelimit: uint256):
    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.

```
@external
@payable
def participate():
    assert block.timestamp < self.deadline, "deadline not met (yet)"

    nfi: int128 = self.nextFunderIndex

    self.funders[nfi] = Funder({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.

```
@external
def finalize():
    assert block.timestamp >= self.deadline, "deadline not met (yet)"
    assert self.balance >= self.goal, "invalid balance"

    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 that's 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.

```
@external
def refund():
    assert block.timestamp >= self.deadline and self.balance < self.goal

    ind: int128 = 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] = empty(Funder)

    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 participants, multiple calls to `refund()` may be necessary.

3.5 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 `winningProposals()` method, which iterates through all the proposals and returns the one with the greatest number of votes.

```
# Voting with delegation.

# Information about voters
struct Voter:
```

(continues on next page)

(continued from previous page)

```
    # 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

# Users can create proposals
struct Proposal:
    # short name (up to 32 bytes)
    name: bytes32
    # number of accumulated votes
    voteCount: int128

voters: public(HashMap[address, Voter])
proposals: public(HashMap[int128, Proposal])
voterCount: public(int128)
chairperson: public(address)
int128Proposals: public(int128)


@view
@internal
def _delegated(addr: address) -> bool:
    return self.voters[addr].delegate != ZERO_ADDRESS


@view
@external
def delegated(addr: address) -> bool:
    return self._delegated(addr)


@view
@internal
def _directlyVoted(addr: address) -> bool:
    return self.voters[addr].voted and (self.voters[addr].delegate == ZERO_ADDRESS)


@view
@external
def directlyVoted(addr: address) -> bool:
    return self._directlyVoted(addr)


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

(continues on next page)

(continued from previous page)

```
        self.int128Proposals += 1

# Give a `voter` the right to vote on this ballot.
# This may only be called by the `chairperson`.
@external
def giveRightToVote(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.voterCount += 1

# Used by `delegate` below, callable externally via `forwardWeight`
@internal
def _forwardWeight(delegate_with_weight_to_forward: address):
    assert self._delegated(delegate_with_weight_to_forward)
    # Throw if there is nothing to do:
    assert self.voters[delegate_with_weight_to_forward].weight > 0

    target: address = self.voters[delegate_with_weight_to_forward].delegate
    for i in range(4):
        if self._delegated(target):
            target = self.voters[target].delegate
            # The following effectively detects cycles of length <= 5,
            # in which the delegation is given back to the delegator.
            # This could be done for any int128ber of loops,
            # or even infinitely with a while loop.
            # However, cycles aren't actually problematic for correctness;
            # they just result in spoiled votes.
            # So, in the production version, this should instead be
            # the responsibility of the contract's client, and this
            # check should be removed.
            assert target != delegate_with_weight_to_forward
        else:
            # Weight will be moved to someone who directly voted or
            # hasn't voted.
            break

    weight_to_forward: int128 = self.voters[delegate_with_weight_to_forward].weight
    self.voters[delegate_with_weight_to_forward].weight = 0
    self.voters[target].weight += weight_to_forward

    if self._directlyVoted(target):
        self.proposals[self.voters[target].vote].voteCount += weight_to_forward
        self.voters[target].weight = 0

    # To reiterate: if target is also a delegate, this function will need
    # to be called again, similarly to as above.

# Public function to call _forwardWeight
@external
def forwardWeight(delegate_with_weight_to_forward: address):
    self._forwardWeight(delegate_with_weight_to_forward)

```

(continues on next page)

(continued from previous page)

```
# Delegate your vote to the voter `to`.
@external
def delegate(to: address):
    # Throws if the sender has already voted
    assert not self.voters[msg.sender].voted
    # Throws if the sender tries to delegate their vote to themselves or to
    # the default address value of 0x0000000000000000000000000000000000000000
    # (the latter might not be problematic, but I don't want to think about it).
    assert to != msg.sender
    assert to != ZERO_ADDRESS

    self.voters[msg.sender].voted = True
    self.voters[msg.sender].delegate = to

    # This call will throw if and only if this delegation would cause a loop
        # of length <= 5 that ends up delegating back to the delegator.
    self._forwardWeight(msg.sender)

# Give your vote (including votes delegated to you)
# to proposal `proposals[proposal].name`.
@external
def vote(proposal: int128):
    # can't vote twice
    assert not self.voters[msg.sender].voted
    # can only vote on legitimate proposals
    assert proposal < self.int128Proposals

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

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

# Computes the winning proposal taking all
# previous votes into account.
@view
@internal
def _winningProposal() -> int128:
    winning_vote_count: int128 = 0
    winning_proposal: int128 = 0
    for i in range(2):
        if self.proposals[i].voteCount > winning_vote_count:
            winning_vote_count = self.proposals[i].voteCount
            winning_proposal = i
    return winning_proposal

@view
@external
def winningProposal() -> int128:
    return self._winningProposal()


# Calls winningProposal() function to get the index
# of the winner contained in the proposals array and then
# returns the name of the winner
@view
```

(continues on next page)

(continued from previous page)

```
@external
def winnerName() -> bytes32:
    return self.proposals[self._winningProposal()].name
```

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

```
# Information about voters
struct Voter:
    # 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

# Users can create proposals
struct Proposal:
    # short name (up to 32 bytes)
    name: bytes32
    # number of accumulated votes
    voteCount: int128

voters: public(HashMap[address, Voter])
proposals: public(HashMap[int128, Proposal])
voterCount: public(int128)
chairperson: public(address)
int128Proposals: 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 data types.

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

Let's move onto the constructor.

```
@external
def __init__(_proposalNames: bytes32[2]):
    self.chairperson = msg.sender
    self.voterCount = 0
    for i in range(2):
        self.proposals[i] = Proposal({
            name: _proposalNames[i],
            voteCount: 0
        })
        self.int128Proposals += 1
```

Note: `msg.sender` and `msg.value` can only be accessed from external functions. If you require these values within an internal function they must be passed as parameters.

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 `giveRightToVote()`, 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.

```
@external
def giveRightToVote(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.voterCount += 1
```

Note: Throughout this contract, we use a pattern where `@external` functions return data from `@internal` functions that have the same name prepended with an underscore. This is because Vyper does not allow calls between external functions within the same contract. The internal function handles the logic and allows internal access, while the external function acts as a getter to allow external viewing.

We need a way to control who has the ability to vote. The method `giveRightToVote()` 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 `voterCount`.

```
@external
def delegate(to: address):
    # Throws if the sender has already voted
    assert not self.voters[msg.sender].voted
    # Throws if the sender tries to delegate their vote to themselves or to
    # the default address value of 0x0000000000000000000000000000000000000000
    # (the latter might not be problematic, but I don't want to think about it).
    assert to != msg.sender
    assert to != ZERO_ADDRESS

    self.voters[msg.sender].voted = True
    self.voters[msg.sender].delegate = to

    # This call will throw if and only if this delegation would cause a loop
        # of length <= 5 that ends up delegating back to the delegator.
    self._forwardWeight(msg.sender)
```

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 `voterCount` directly if the delegate had already voted or increase the delegate's vote `weight` if the delegate has not yet voted.

```
@external
def vote(proposal: int128):
    # can't vote twice
    assert not self.voters[msg.sender].voted
    # can only vote on legitimate proposals
    assert proposal < self.int128Proposals

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

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

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 `voteCount` 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: `winningProposal()`, which returns the key of the proposal, and `winnerName()`, returning the name of the proposal. Notice the `@view` 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 `@view` 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.
@view
@internal
def _winningProposal() -> int128:
    winning_vote_count: int128 = 0
    winning_proposal: int128 = 0
    for i in range(2):
        if self.proposals[i].voteCount > winning_vote_count:
            winning_vote_count = self.proposals[i].voteCount
            winning_proposal = i
    return winning_proposal

@view
@external
def winningProposal() -> int128:
    return self._winningProposal()

```

The `_winningProposal()` 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 `winningVoteCount` and `winningProposal`, respectively by looping through all the proposals.

`winningProposal()` is an external function allowing access to `_winningProposal()`.

```
@view
@external
def winnerName() -> bytes32:
    return self.proposals[self._winningProposal()].name
```

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

```
# Financial events the contract logs

event Transfer:
    sender: indexed(address)
    receiver: indexed(address)
    value: uint256

event Buy:
    buyer: indexed(address)
    buy_order: uint256

event Sell:
    seller: indexed(address)
    sell_order: uint256

event Pay:
    vendor: indexed(address)
    amount: uint256


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

# Store a ledger of stockholder holdings.
holdings: HashMap[address, uint256]

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

    self.company = _company
    self.totalShares = _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

# Find out how much stock the company holds
```

(continues on next page)

(continued from previous page)

```
@view
@internal
def _stockAvailable() -> uint256:
    return self.holdings[self.company]

# Public function to allow external access to _stockAvailable
@view
@external
def stockAvailable() -> uint256:
    return self._stockAvailable()

# Give some value to the company and get stock in return.
@external
@payable
def buyStock():
    # Note: full amount is given to company (no fractional shares),
    #       so be sure to send exact amount to buy shares
    buy_order: uint256 = msg.value / self.price # rounds down

    # Check that there are enough shares to buy.
    assert self._stockAvailable() >= buy_order

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

    # Log the buy event.
    log Buy(msg.sender, buy_order)

# Find out how much stock any address (that's owned by someone) has.
@view
@internal
def _getHolding(_stockholder: address) -> uint256:
    return self.holdings[_stockholder]

# Public function to allow external access to _getHolding
@view
@external
def getHolding(_stockholder: address) -> uint256:
    return self._getHolding(_stockholder)

# Return the amount the company has on hand in cash.
@view
@external
def cash() -> uint256:
    return self.balance

# Give stock back to the company and get money back as ETH.
@external
def sellStock(sell_order: uint256):
    assert sell_order > 0 # Otherwise, this would fail at send() below,
        # due to an OOG error (there would be zero value available for gas).
    # You can only sell as much stock as you own.
    assert self._getHolding(msg.sender) >= sell_order
    # Check that the company can pay you.
    assert self.balance >= (sell_order * self.price)

```

(continues on next page)

(continued from previous page)

```
    # 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)

    # Log the sell event.
    log Sell(msg.sender, sell_order)

# Transfer stock from one stockholder to another. (Assume that the
# receiver is given some compensation, but this is not enforced.)
@external
def transferStock(receiver: address, transfer_order: uint256):
    assert transfer_order > 0 # This is similar to sellStock above.
    # Similarly, you can only trade as much stock as you own.
    assert self._getHolding(msg.sender) >= transfer_order

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

    # Log the transfer event.
    log Transfer(msg.sender, receiver, transfer_order)

# Allow the company to pay someone for services rendered.
@external
def payBill(vendor: address, amount: uint256):
    # Only the company can pay people.
    assert msg.sender == self.company
    # Also, it can pay only if there's enough to pay them with.
    assert self.balance >= amount

    # Pay the bill!
    send(vendor, amount)

    # Log the payment event.
    log Pay(vendor, amount)

# Return the amount in wei that a company has raised in stock offerings.
@view
@internal
def _debt() -> uint256:
    return (self.totalShares - self._stockAvailable()) * self.price

# Public function to allow external access to _debt
@view
@external
def debt() -> uint256:
    return self._debt()

# Return the cash holdings minus the debt of the company.
# The share debt or liability only is included here,
# but of course all other liabilities can be included.
@view
@external
def worth() -> uint256:
    return self.balance - self._debt()
```

Note: Throughout this contract, we use a pattern where `@external` functions return data from `@internal` functions that have the same name prepended with an underscore. This is because Vyper does not allow calls between external functions within the same contract. The internal function handles the logic, while the external function acts as a getter to allow viewing.

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 definitions.

```
event Transfer:
    sender: indexed(address)
    receiver: indexed(address)
    value: uint256

event Buy:
    buyer: indexed(address)
    buy_order: uint256

event Sell:
    seller: indexed(address)
    sell_order: uint256

event Pay:
    vendor: indexed(address)
    amount: uint256


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

# Store a ledger of stockholder holdings.
holdings: HashMap[address, uint256]
```

We initiate the `company` variable to be of type `address` that's public. The `totalShares` 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.

```
@external
def __init__(_company: address, _total_shares: uint256, initial_price: uint256):
    assert _total_shares > 0
    assert initial_price > 0

    self.company = _company
    self.totalShares = _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
```

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.

```
# Find out how much stock the company holds
@view
@internal
def _stockAvailable() -> uint256:
    return self.holdings[self.company]

# Public function to allow external access to _stockAvailable
@view
@external
def stockAvailable() -> uint256:
    return self._stockAvailable()
```

We will be seeing a few `@view` 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 `_stockAvailable()` 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. Because `_stockAvailable()` is an internal method, we also include the `stockAvailable()` method to allow external access.

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

```
@external
@payable
def buyStock():
    # Note: full amount is given to company (no fractional shares),
    #       so be sure to send exact amount to buy shares
    buy_order: uint256 = msg.value / self.price # rounds down

    # Check that there are enough shares to buy.
    assert self._stockAvailable() >= buy_order

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

    # Log the buy event.
    log Buy(msg.sender, buy_order)
```

The `buyStock()` method is a `@payable` method which takes an amount of ether sent and calculates the `buyOrder` (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?

```
@view
@internal
def _getHolding(_stockholder: address) -> uint256:
    return self.holdings[_stockholder]

# Public function to allow external access to _getHolding
@view
@external
def getHolding(_stockholder: address) -> uint256:
    return self._getHolding(_stockholder)
```

The `_getHolding()` is another `@view` method that takes an `address` and returns its corresponding stock holdings by keying into `self.holdings`. Again, an external function `getHolding()` is included to allow access.

```
@view
@external
def cash() -> uint256:
    return self.balance
```

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

```
@external
def sellStock(sell_order: uint256):
    assert sell_order > 0 # Otherwise, this would fail at send() below,
        # due to an OOG error (there would be zero value available for gas).
    # You can only sell as much stock as you own.
    assert self._getHolding(msg.sender) >= sell_order
    # Check that the company can pay you.
    assert self.balance >= (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)

    # Log the sell event.
    log Sell(msg.sender, sell_order)
```

To sell a stock, we have the `sellStock()` 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.

```
@external
def transferStock(receiver: address, transfer_order: uint256):
    assert transfer_order > 0 # This is similar to sellStock above.
    # Similarly, you can only trade as much stock as you own.
    assert self._getHolding(msg.sender) >= transfer_order

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

    # Log the transfer event.
    log Transfer(msg.sender, receiver, transfer_order)
```

A stockholder can also transfer their stock to another stockholder with the `transferStock()` 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.

```
@external
def payBill(vendor: address, amount: uint256):
    # Only the company can pay people.
    assert msg.sender == self.company
    # Also, it can pay only if there's enough to pay them with.
    assert self.balance >= amount

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

(continues on next page)

(continued from previous page)

```

    # Log the payment event.
    log Pay(vendor, amount)
```

The company is also allowed to pay out an amount in ether to an address by calling the `payBill()` 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.

```
@view
@internal
def _debt() -> uint256:
    return (self.totalShares - self._stockAvailable()) * self.price

# Public function to allow external access to _debt
@view
@external
def debt() -> uint256:
    return self._debt()
```

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. Internally, we get this value by calling the `_debt()` method. Externally it is accessed via `debt()`.

```
@view
@external
def worth() -> uint256:
    return self.balance - 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 Vyper language has convinced you of its capabilities and readability in writing smart contracts.

CHAPTER 4

Structure of a Contract

Vyper contracts are contained within files. Each file contains exactly one contract.

This section provides a quick overview of the types of data present within a contract, with links to other sections where you can obtain more details.

4.1 Version Pragma

Vyper supports a version pragma to ensure that a contract is only compiled by the intended compiler version, or range of versions. Version strings use NPM style syntax.

```
# @version ^0.2.0
```

In the above example, the contract only compiles with Vyper versions `0.2.x`.

4.2 State Variables

State variables are values which are permanently stored in contract storage. They are declared outside of the body of any functions, and initially contain the *default value* for their type.

```
storedData: int128
```

State variables are accessed via the *self* object.

```
self.storedData = 123
```

See the documentation on *Types* or Scoping and Declarations for more information.

4.3 Functions

Functions are executable units of code within a contract.

```
@external
def bid():
    ...
```

Functions may be called internally or externally depending on their *visibility*. Functions may accept input arguments and return variables in order to pass values between them.

See the *Functions* documentation for more information.

4.4 Events

Events provide an interface for the EVM's logging facilities. Events may be logged with specially indexed data structures that allow clients, including light clients, to efficiently search for them.

```
event Payment:
    amount: int128
    sender: indexed(address)

total_paid: int128

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

See the *Event* documentation for more information.

4.5 Interfaces

An interface is a set of function definitions used to enable calls between smart contracts. A contract interface defines all of that contract's externally available functions. By importing the interface, your contract now knows how to call these functions in other contracts.

Interfaces can be added to contracts either through inline definition, or by importing them from a seperate file.

```
interface FooBar:
    def calculate() -> uint256: view
    def test1(): nonpayable
```

```
from foo import FooBar
```

Once defined, an interface can then be used to make external calls to a given address:

```
@external
def test(some_address: address):
    FooBar(some_address).calculate()
```

See the *Interfaces* documentation for more information.

4.6 Structs

A struct is custom defined type that can allows you to group several variables together:

```
struct MyStruct:
    value1: int128
    value2: decimal
```

See the *Structs* documentation for more information.

CHAPTER 5

Types

Vyper is a statically typed language. The type of each variable (state and local) must 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.

5.1 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.

5.1.1 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

Short-circuiting of boolean operators (`or` and `and`) is consistent with the behavior of Python.

5.1.2 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.

Interger literals cannot have a decimal point even if the decimal value is zero. For example, `2.0` cannot be interpreted as an integer.

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

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

5.1.3 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).

Interger literals cannot have a decimal point even if the decimal value is zero. For example, `2.0` cannot be interpreted as an integer.

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.

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 `uint256`.

Arithmetic Operators

Operator	Description
`x + y`	Addition
`x - y`	Subtraction
`x * y`	Multiplication
`x / y`	Division
`x**y`	Exponentiation
`x % y`	Modulo

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

5.1.4 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 $(2^{127} - 1)$.

In order for a literal to be interpreted as `decimal` it must include a decimal point.

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

Operator	Description
`x + y`	Addition
`x - y`	Subtraction
`-x`	Unary minus/Negation
`x * y`	Multiplication
`x / y`	Division
`x % y`	Modulo

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

5.1.5 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. Address literals must be written in hexadecimal notation with a leading `0x` and must be checksummed.

Members

Member	Type	Description
balance	uint256	Balance of an address
codehash	bytes32	Keccak of code at an address, EMPTY_BYTES32 if no contract is deployed
codesize	uint256	Size of code deployed an address, in bytes
is_contract	bool	Boolean indicating if a contract is deployed at an address

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

Note: Operations such as SELFDESTRUCT and CREATE2 allow for the removal and replacement of bytecode at an address. You should never assume that values of address members will not change in the future.

5.1.6 32-bit-wide Byte Array

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

Example:

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

Operators

Keyword	Description
keccak256(x)	Return the keccak256 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.

5.1.7 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. On the ABI level the Fixed-size bytes array is annotated as bytes.

Bytes literals may be given as bytes strings, hexadecimal, or binary.

```
bytes_string: Bytes[100] = b"\x01"
hex_bytes: Bytes[100] = 0x01
binary_bytes: Bytes[100] = 0b00000001
```

5.1.8 Strings

Keyword: String

Fixed-size strings can hold strings with equal or fewer characters than the maximum length of the string. On the ABI level the Fixed-size bytes array is annotated as string.

```
example_str: String[100] = "Test String"
```

5.2 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.

5.2.1 Fixed-size Lists

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

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

```
# Defining a list
exampleList: int128[3]

# Setting values
exampleList = [10, 11, 12]
exampleList[2] = 42

# Returning a value
return exampleList[0]
```

5.2.2 Structs

Structs are custom defined types that can group several variables.

Struct types can be used inside mappings and arrays. Structs can contain arrays and other structs, but not mappings.

Struct members can be accessed via struct.argname.

5.2.3 Mappings

Mappings are hash tables that 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 key data is not stored in a mapping, instead its keccak256 hash used to look up a value. For this reason mappings do not have a length or a concept of a key or value being "set".

Mapping types are declared as HashMap[_KeyType, _ValueType].

- _KeyType can be any base or bytes type. Mappings, interfaces or structs are not support as key types.
- _ValueType can actually be any type, including mappings.

Note: Mappings are only allowed as state variables.

```
# Defining a mapping
exampleMapping: HashMap[int128, decimal]

# Accessing a value
exampleMapping[0] = 10.1
```

Note: Mappings have no concept of length and so cannot be iterated over.

5.3 Initial Values

Unlike most programming languages, Vyper does not have a concept of `null`. Instead, every variable type has a default value. To check if a variable is empty, you must compare it to the default value for it's given type.

To reset a variable to it's default value, assign to it the built-in `empty()` function which constructs a zero value for that type.

Note: Memory variables must be assigned a value at the time they are declared.

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

Type	Default Value
`address`	`0x0000000000000000000000000000000000000000`
`bool`	`False`
`bytes32`	`0x0000000000000000000000000000000000000000000000000000000000000000`
`decimal`	`0.0`
`int128`	`1`
`uint256`	`1`

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.

5.4 Type Conversions

All type conversions in Vyper must be made explicitly using the built-in `convert(a: atype, btype)` function. Currently, the following type conversions are supported:

In (atype)	Out (btype)	Allowable Values	Additional Notes
bool	decimal	All	0.0 or 1.0
bool	int128	All	0 or 1
bool	uint256	All	0 or 1
bool	bytes32	All	0x00 or 0x01
bool	Bytes	All	
decimal	bool	All	Returns a != 0.0
decimal	int128	All	Value is truncated
decimal	uint256	a >= 0.0	Value is truncated
decimal	bytes32	All	
decimal	Bytes	All	
int128	bool	All	Returns a != 0
int128	decimal	All	
int128	uint256	a >= 0	Cannot convert negative values
int128	bytes32	All	
int128	Bytes	All	
uint256	bool	All	Returns a != 0
uint256	decimal	a <= MAX_DECIMAL	
uint256	int128	a <= MAX_INT128	
uint256	bytes32	All	
uint256	Bytes	All	
bytes32	bool	All	True if a is not empty
bytes32	decimal	All	
bytes32	int128	All	
bytes32	uint256	All	
bytes32	Bytes	All	

CHAPTER 6

Environment Variables and Constants

6.1 Environment Variables

Environment variables always exist in the namespace and are primarily used to provide information about the blockchain or current transaction.

6.1.1 Block and Transaction Properties

Name	Type	Value
`block.coinbase`	`address`	Current block miner's address
`block.difficulty`	`uint256`	Current block difficulty
`block.number`	`uint256`	Current block number
`block.prevhash`	`bytes32`	Equivalent to `blockhash(block.number - 1)`
`block.timestamp`	`uint256`	Current block epoch timestamp
`chain.id`	`uint256`	Chain ID
`msg.gas`	`uint256`	Remaining gas
`msg.sender`	`address`	Sender of the message (current call)
`msg.value`	`uint256`	Number of wei sent with the message
`tx.origin`	`address`	Sender of the transaction (full call chain)

Note: `msg.sender` and `msg.value` can only be accessed from external functions. If you require these values within a private function they must be passed as parameters.

6.1.2 The self Variable

`self` is an environment variable used to reference a contract from within itself. Along with the normal *address* members, `self` allows you to read and write to state variables and to call private functions within the contract.

Name	Type	Value
`self`	`address`	Current contract's address
`self.balance`	`uint256`	Current contract's balance

Accessing State Variables

`self` is used to access a contract's *state variables*, as shown in the following example:

```
state_var: uint256

@external
def set_var(value: uint256) -> bool:
    self.state_var = value
    return True

@external
@view
def get_var() -> uint256:
    return self.state_var
```

Calling Internal Functions

`self` is also used to call *internal functions* within a contract:

```
@internal
def _times_two(amount: uint256) -> uint256:
    return amount * 2

@external
def calculate(amount: uint256) -> uint256:
    return self._times_two(amount)
```

6.2 Built In Constants

Vyper has a few convenience constants builtin.

Name	Type	Value
`ZERO_ADDRESS`	`address`	`0x0000000000000000000000000000000000000000`
`EMPTY_BYTES32`	`bytes32`	`0x0000000000000000000000000000000000000000000000000000000000000000`
`MAX_INT128`	`int128`	`2**127 - 1`
`MIN_INT128`	`int128`	`-2**127`
`MAX_DECIMAL`	`decimal`	`(2**127 - 1)`
`MIN_DECIMAL`	`decimal`	`(-2**127)`
`MAX_UINT256`	`uint256`	`2**256 - 1`

6.3 Custom Constants

Custom constants can be defined at a global level in Vyper. To define a constant make use of the `constant` keyword.

```
TOTAL_SUPPLY: constant(uint256) = 10000000
total_supply: public(uint256)

@external
def __init__():
    self.total_supply = TOTAL_SUPPLY
```

CHAPTER 7

Statements

Vyper's statements are syntactically similar to Python, with some notable exceptions.

7.1 Control Flow

7.1.1 break

The `break` statement terminates the nearest enclosing `for` loop.

```
for i in [1, 2, 3, 4, 5]:
    if i == a:
        break
```

In the above example, the `for` loop terminates if `i == a`.

7.1.2 continue

The `continue` statement begins the next cycle of the nearest enclosing `for` loop.

```
for i in [1, 2, 3, 4, 5]:
    if i != a:
        continue
    ...
```

In the above example, the `for` loop begins the next cycle immediately whenever `i != a`.

7.1.3 pass

`pass` is a null operation — when it is executed, nothing happens. It is useful as a placeholder when a statement is required syntactically, but no code needs to be executed:

```
# this function does nothing (yet!)

@external
def foo():
    pass
```

7.1.4 return

`return` leaves the current function call with the expression list (or None) as a return value.

```
return RETURN_VALUE
```

An important distinction between Vyper and Python is that Vyper does not implicitly return `None` at the end of a function if no `return` statement is given. All functions must end with a `return` statement, or another terminating action such as `raise`.

It is not allowed to have additional, unreachable statements after a `return` statement.

7.2 Event Logging

7.2.1 log

The `log` statement is used to log an event:

```
log MyEvent(...)
```

The event must have been previously declared

See *Event Logging* for more information on events.

7.3 Assertions and Exceptions

Vyper uses state-reverting exceptions to handle errors. Exceptions trigger the `REVERT` opcode (`0xFD`) with the provided reason given as the error message. When an exception is raised the code stops operation, the contract's state is reverted to the state before the transaction took place and the remaining gas is returned to the transaction's sender. When an exception happen in a sub-call, it "bubbles up" (i.e., exceptions are rethrown) automatically.

If the reason string is set to `UNREACHABLE`, an `INVALID` opcode (`0xFE`) is used instead of `REVERT`. In this case, calls that revert do not receive a gas refund. This is not a recommended practice for general usage, but is available for interoperability with various tools that use the `INVALID` opcode to perform dynamic analysis.

7.3.1 raise

The `raise` statement triggers an exception and reverts the current call.

```
raise "something went wrong"
```

The error string is not required.

7.3.2 assert

The `assert` statement makes an assertion about a given condition. If the condition evaluates falsely, the transaction is reverted.

```
assert x > 5, "value too low"
```

The error string is not required.

This method's behavior is equivalent to:

```
if not cond:
    raise "reason"
```

CHAPTER 8

Control Structures

8.1 Functions

Functions are executable units of code within a contract. Functions may only be declared within a contract's *module scope*.

```
@external
def bid():
    ...
```

Functions may be called internally or externally depending on their *visibility*. Functions may accept input arguments and return variables in order to pass values between them.

8.1.1 Visibility

All functions must include exactly one visibility decorator.

External Functions

External functions (marked with the `@external` decorator) are a part of the contract interface and may only be called via transactions or from other contracts.

```
@external
def add_seven(a: int128) -> int128:
    return a + 7
```

A Vyper contract cannot call directly between two external functions. If you must do this, you can use an *interface*.

Internal Functions

Internal functions (marked with the `@internal` decorator) are only accessible from other functions within the same contract. They are called via the *self* object:

```
@internal
def _times_two(amount: uint256) -> uint256:
    return amount * 2

@external
def calculate(amount: uint256) -> uint256:
    return self._times_two(amount)
```

Internal functions do not have access to `msg.sender` or `msg.value`. If you require these values within an internal function you must pass them as parameters.

8.1.2 Mutability

You can optionally declare a function's mutability by using a *decorator*. There are four mutability levels:

- **Pure**: does not read from the contract state or any environment variables.
- **View**: may read from the contract state, but does not alter it.
- **Nonpayable**: may read from and write to the contract state, but cannot receive Ether.
- **Payable**: may read from and write to the contract state, and can receive Ether.

```
@view
@external
def readonly():
    # this function cannot write to state
    ...

@payable
@external
def send_me_money():
    # this function can receive ether
    ...
```

Functions default to nonpayable when no mutability decorator is used.

8.1.3 Re-entrancy Locks

The `@nonreentrant(<key>)` decorator places a lock on a function, and all functions with the same `<key>` value. An attempt by an external contract to call back into any of these functions causes the transaction to revert.

```
@external
@nonreentrant("lock")
def make_a_call(_addr: address):
    # this function is protected from re-entrancy
    ...
```

8.1.4 The *__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__`. It must be annotated with `@external`. It cannot expect any input arguments and cannot return any values.

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.

```
event Payment:
    amount: int128
    sender: indexed(address)

@external
@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` object, including:

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

8.1.5 The *__init__* Function

`__init__` is a special initialization function that may only be called at the time of deploying a contract. It can be used to set initial values for storage variables. A common use case is to set an `owner` variable with the creator the contract:

```
owner: address

def __init__():
    self.owner = msg.sender
```

You cannot call to other contract functions from the initialization function.

8.1.6 Decorators Reference

All functions must include one *visibility* decorator (`@external` or `@internal`). The remaining decorators are optional.

Decorator	Description
`@external`	Function can only be called externally
`@internal`	Function can only be called within current contract
`@pure`	Function does not read contract state or environment variables
`@view`	Function does not alter contract state
`@payable`	Function is able to receive Ether
`@nonreentrant(<unique_key>)`	Function cannot be called back into during an external call

8.2 `if` statements

The `if` statement is a control flow construct used for conditional execution:

```
if CONDITION:
    ...
```

`CONDITION` is a boolean or boolean operation. The boolean is evaluated left-to-right, one expression at a time, until the condition is found to be true or false. If true, the logic in the body of the `if` statement is executed.

Note that unlike Python, Vyper does not allow implicit conversion from non-boolean types within the condition of an `if` statement. `if 1:  pass` will fail to compile with a type mismatch.

You can also include `elif` and `else` statements, to add more conditional statements and a body that executes when the conditionals are false:

```
if CONDITION:
    ...
elif OTHER_CONDITION:
    ...
else:
    ...
```

8.3 `for` loops

The `for` statement is a control flow construct used to iterate over a value:

```
for i in <ITERABLE>:
    ...
```

The iterated value can be a static array, or generated from the builtin `range` function.

8.3.1 Array Iteration

You can use `for` to iterate through the values of any array variable:

```
foo: int128[3] = [4, 23, 42]
for i in foo:
    ...
```

In the above, example, the loop executes three times with `i` assigned the values of `4`, `23`, and then `42`.

You can also iterate over a literal array, as long as a common type can be determined for each item in the array:

```
for i in [4, 23, 42]:
    ...
```

Some restrictions:

- You cannot iterate over a multi-dimensional array. `i` must always be a base type.
- You cannot modify a value in an array while it is being iterated, or call to a function that might modify the array being iterated.

8.3.2 Range Iteration

Ranges are created using the `range` function. The following examples are valid uses of `range`:

```
for i in range(STOP):
    ...
```

`STOP` is a literal integer greater than zero. `i` begins as zero and increments by one until it is equal to `STOP`.

```
for i in range(start, stop):
    ...
```

`START` and `STOP` are literal integers, with `STOP` being a greater value than `START`. `i` begins as `START` and increments by one until it is equal to `STOP`.

```
for i in range(a, a + N):
    ...
```

`a` is a variable with an integer type and `N` is a literal integer greater than zero. `i` begins as `a` and increments by one until it is equal to `a + N`.

CHAPTER 9

Scoping and Declarations

9.1 Variable Declaration

The first time a variable is referenced you must declare it's *type*:

```
data: int128
```

In the above example we declare variable `data` with a type of `int128`.

Depending on the active scope, an initial value may or may not be assigned:

- For storage variables (declared in the module scope), an initial value **cannot** be set
- For memory variables (declared within a function), an initial value **must** be set
- For calldata variables (function input arguments), a default value **may** be given

9.1.1 Declaring Public Variables

Storage variables can be marked as `public` during declaration:

```
data: public(int128)
```

The compiler automatically creates getter functions for all public storage variables. For the example above below, the compiler will generate a function called `data` that does not take any arguments and returns an `int128`, the value of the state variable data.

For public arrays, you can only retrieve a single element via the generated getter. This mechanism exists to avoid high gas costs when returning an entire array. The getter will accept an argument to specity which element to return, for example `data(0)`.

9.1.2 Tuple Assignment

You cannot directly declare tuple types. However, in certain cases you can use literal tuples during assignment. For example, when a function returns multiple values:

```
@internal
def foo() -> (int128: int128):
    return 2, 3

@external
def bar():
    a: int128 = 0
    b: int128 = 0

    # the return value of `foo` is assigned using a tuple
    (a, b) = self.foo()

    # Can also skip the parenthesis
    a, b = self.foo()
```

9.2 Scoping Rules

Vyper follows C99 scoping rules. Variables are visible from the point right after their declaration until the end of the smallest block that contains the declaration.

9.2.1 Module Scope

Variables and other items declared outside of a code block (functions, constants, event and struct definitions, . . .), are visible even before they were declared. This means you can use module-scoped items before they are declared.

An exception to this rule is that you can only call functions that have already been declared.

Accessing Module Scope from Functions

Values that are declared in the module scope of a contract, such as storage variables and functions, are accessed via the `self` object:

```
a: int128

@internal
def foo() -> int128
    return 42

@external
def foo() -> int128:
    b: int128 = self.foo()
    return self.a  + b
```

Name Shadowing

It is not permitted for a memory or calldata variable to shadow the name of a storage variable. The following examples will not compile:

```
a: int128

@external
def foo() -> int128:
    # memory variable cannot have the same name as a storage variable
    a: int128 = self.a
    return a
```

```
a: int128

@external
def foo(a: int128) -> int128:
    # input argument cannot have the same name as a storage variable
    return a
```

9.2.2 Function Scope

Variables that are declared within a function, or given as function input arguments, are visible within the body of that function. For example, the following contract is valid because each declaration of `a` only exists within one function's body.

```
@external
def foo(a: int128):
    pass

@external
def bar(a: uint256):
    pass

@external
def baz():
    a: bool = True
```

The following examples will not compile:

```
@external
def foo(a: int128):
    # `a` has already been declared as an input argument
    a: int128 = 21
```

```
@external
def foo(a: int128):
    a = 4

@external
def bar():
    # `a` has not been declared within this function
    a += 12
```

9.2.3 Block Scopes

Logical blocks created by `for` and `if` statements have their own scope. For example, the following contract is valid because `x` only exists within the block scopes for each branch of the `if` statement:

```
@external
def foo(a: bool) -> int128:
    if a:
        x: int128 = 3
    else:
        x: bool = False
```

In a `for` statement, the target variable exists within the scope of the loop. For example, the following contract is valid because `i` is no longer available upon exitting the loop:

```
@external
def foo(a: bool) -> int128:
    for i in [1, 2, 3]:
        pass
    i: bool = False
```

The following contract fails to compile because `a` has not been declared outside of the loop.

```
@external
def foo(a: bool) -> int128:
    for i in [1, 2, 3]:
        a: int128 = i
    a += 3
```

CHAPTER 10

Built in Functions

Vyper provides a collection of built in functions available in the global namespace of all contracts.

10.1 Bitwise Operations

bitwise_and (*x: uint256*, *y: uint256*) → uint256

Perform a "bitwise and" operation. Each bit of the output is 1 if the corresponding bit of x AND of y is 1, otherwise it's 0.

```
@external
@view
def foo(x: uint256, y: uint256) -> uint256:
    return bitwise_and(x, y)
```

```
>>> ExampleContract.foo(31337, 8008135)
12353
```

bitwise_not (*x: uint256*) → uint256

Return the complement of x - the number you get by switching each 1 for a 0 and each 0 for a 1.

```
@external
@view
def foo(x: uint256) -> uint256:
    return bitwise_not(x)
```

```
>>> ExampleContract.foo(0)
115792089237316195423570985008687907853269984665640564039457584007913129639935
```

bitwise_or (*x: uint256*, *y: uint256*) → uint256

Perform a "bitwise or" operation. Each bit of the output is 0 if the corresponding bit of x AND of y is 0, otherwise it's 1.

```
@external
@view
def foo(x: uint256, y: uint256) -> uint256:
    return bitwise_or(x, y)
```

```
>>> ExampleContract.foo(31337, 8008135)
8027119
```

bitwise_xor (*x: uint256*, *y: uint256*) → uint256

Perform a "bitwise exclusive or" operation. Each bit of the output is the same as the corresponding bit in `x` if that bit in `y` is 0, and it's the complement of the bit in `x` if that bit in `y` is 1.

```
@external
@view
def foo(x: uint256, y: uint256) -> uint256:
    return bitwise_xor(x, y)
```

```
>>> ExampleContract.foo(31337, 8008135)
8014766
```

shift (*x: uint256*, *_shift: int128*) → uint256

Return `x` with the bits shifted `_shift` places. A positive `_shift` value equals a left shift, a negative value is a right shift.

```
@external
@view
def foo(x: uint256, y: int128) -> uint256:
    return shift(x, y)
```

```
>>> ExampleContract.foo(2, 8)
512
```

10.2 Chain Interaction

create_forwarder_to (*target: address*, *value: uint256 = 0*) → address

Deploys a small contract that duplicates the logic of the contract at `target`, but has it's own state since every call to `target` is made using `DELEGATECALL` to `target`. To the end user, this should be indistinguishable from an independantly deployed contract with the same code as `target`.

Note: It is very important that the deployed contract at `target` is code you know and trust, and does not implement the `selfdestruct` opcode as this will affect the operation of the forwarder contract.

- `target`: Address of the contract to duplicate
- `value`: The wei value to send to the new contract address (Optional, default 0)

Returns the address of the duplicated contract.

```
@external
def foo(_target: address) -> address:
    return create_forwarder_to(_target)
```

---

raw_call (*to: address*, *data: Bytes*, *max_outsize: int = 0*, *gas: uint256 = gasLeft*, *value: uint256 = 0*, *is_delegate_call: bool = False*, *is_static_call: bool = False*) → Bytes[max_outsize]

Call to the specified Ethereum address.

- `to`: Destination address to call to
- `data`: Data to send to the destination address
- `max_outsize`: Maximum length of the bytes array returned from the call. If the returned call data exceeds this length, only this number of bytes is returned.
- `gas`: The amount of gas to attach to the call. If not set, all remaining gas is forwarded.
- `value`: The wei value to send to the address (Optional, default `0`)
- `is_delegate_call`: If `True`, the call will be sent as `DELEGATECALL` (Optional, default `False`)
- `is_static_call`: If `True`, the call will be sent as `STATICCALL` (Optional, default `False`)

Returns the data returned by the call as a `Bytes` list, with `max_outsize` as the max length.

Returns `None` if `max_outsize` is omitted or set to `0`.

Note: The actual size of the returned data may be less than `max_outsize`. You can use `len` to obtain the actual size.

Returns the address of the duplicated contract.

```
@external
@payable
def foo(_target: address) -> Bytes[32]:
    response: Bytes[32] = raw_call(_target, 0xa9059cbb, max_outsize=32, value=msg.
↪value)
    return response
```

raw_log (*topics: bytes32[4]*, *data: Union[Bytes, bytes32]*) → None

Provides low level access to the `LOG` opcodes, emitting a log without having to specify an ABI type.

- `topics`: List of `bytes32` log topics. The length of this array determines which opcode is used.
- `data`: Unindexed event data to include in the log. May be given as `Bytes` or `bytes32`.

```
@external
def foo(_topic: bytes32, _data: Bytes[100]):
    raw_log([_topic], _data)
```

selfdestruct (*to: address*) → None

Trigger the `SELFDESTRUCT` opcode (`0xFF`), causing the contract to be destroyed.

- `to`: Address to forward the contract's ether balance to

Warning: This method delete the contract from the blockchain. All non-ether assets associated with this contract are "burned" and the contract is no longer accessible.

```
@external
def do_the_needful():
    selfdestruct(msg.sender)
```

send (*to: address*, *value: uint256*) → None

Send ether from the contract to the specified Ethereum address.

- `to`: The destination address to send ether to
- `value`: The wei value to send to the address

Note: The amount to send is always specified in `wei`.

```
@external
def foo(_receiver: address, _amount: uint256):
    send(_receiver, _amount)
```

10.3 Cryptography

ecadd (*a: uint256[2], b: uint256[2]*) → uint256[2]

Take two points on the Alt-BN128 curve and add them together.

```
@external
@view
def foo(x: uint256[2], y: uint256[2]) -> uint256[2]:
    return ecadd(x, y)
```

```
>>> ExampleContract.foo([1, 2], [1, 2])
[
    1368015179489954701390400359078579693043519447331113978918064868415326638035,
    9918110051302171585080402603319702774565515993150576347155970296011118125764,
]
```

ecmul (*point: uint256[2], scalar: uint256*) → uint256[2]

Take a point on the Alt-BN128 curve (`p`) and a scalar value (`s`), and return the result of adding the point to itself `s` times, i.e. `p * s`.

- `point`: Point to be multiplied
- `scalar`: Scalar value

```
@external
@view
def foo(point: uint256[2], scalar: uint256) -> uint256[2]:
    return ecmul(point, scalar)
```

```
>>> ExampleContract.foo([1, 2], 3)
[
    3353031288059533942658390886683067124040920775575537747144343083137631628272,
    19321533766552368860946552437480515441416830039777911637913418824951667761761,
]
```

ecrecover (*hash: bytes32*, *v: uint256*, *r: uint256*, *s: uint256*) → address

Recover the address associated with the public key from the given elliptic curve signature.

- `r`: first 32 bytes of signature
- `s`: second 32 bytes of signature
- `v`: final 1 byte of signature

Returns the associated address, or `0` on error.

keccak256 (*_value*) → bytes32

Return a `keccak256` hash of the given value.

- `_value`: Value to hash. Can be a literal string, `Bytes`, or `bytes32`.

```
@external
@view
def foo(_value: Bytes[100]) -> bytes32
    return keccak256(_value)
```

```
>>> ExampleContract.foo(b"potato")
0x9e159dfcfe557cc1ca6c716e87af98fdcb94cd8c832386d0429b2b7bec02754f
```

sha256 (*_value*) → bytes32

Return a `sha256` (SHA2 256bit output) hash of the given value.

- `_value`: Value to hash. Can be a literal string, `Bytes`, or `bytes32`.

```
@external
@view
def foo(_value: Bytes[100]) -> bytes32
    return sha256(_value)
```

```
>>> ExampleContract.foo(b"potato")
0xe91c254ad58860a02c788dfb5c1a65d6a8846ab1dc649631c7db16fef4af2dec
```

10.4 Data Manipulation

concat (*a*, *b*, **args*) → Union[Bytes, String]

Take 2 or more bytes arrays of type `bytes32`, `Bytes` or `String` and combine them into a single value.

If the input arguments are `String` the return type is `String`. Otherwise the return type is `Bytes`.

```
@external
@view
def foo(a: String[5], b: String[5], c: String[5]) -> String[100]
    return concat(a, " ", b, " ", c, "!")
```

```
>>> ExampleContract.foo("why","hello","there")
"why hello there!"
```

convert (*value*, *type_*) → Any

Converts a variable or literal from one type to another.

- `value`: Value to convert
- `type_`: The destination type to convert to (`bool`, `decimal`, `int128`, `uint256` or `bytes32`)

Returns a value of the type specified by `type_`.

For more details on available type conversions, see *Type Conversions*.

extract32 (*b: Bytes*, *start: int128*, *output_type=bytes32*) → Any

Extract a value from a `Bytes` list.

- `b`: `Bytes` list to extract from

- `start`: Start point to extract from
- `output_type`: Type of output (`bytes32`, `int128`, or `address`). Defaults to `bytes32`.

Returns a value of the type specified by `output_type`.

```
@external
@view
def foo(Bytes[32]) -> address:
    return extract32(b, 12, output_type=address)
```

```
>>> ExampleContract.foo(
↪"0x0000000000000000000000009f8F72aA9304c8B593d555F12eF6589cC3A579A2")
"0x9f8F72aA9304c8B593d555F12eF6589cC3A579A2"
```

slice(*b: Union[Bytes, bytes32, String], start: uint256, length: uint256*) → Union[Bytes, String]

Copy a list of bytes and return a specified slice.

- `b`: value being sliced
- `start`: start position of the slice
- `length`: length of the slice

If the value being sliced is a `Bytes` or `bytes32`, the return type is `Bytes`. If it is a `String`, the return type is `String`.

```
@external
@view
def foo(s: string[32]) -> string[5]:
    return slice(s, 4, 5)
```

```
>>> ExampleContract.foo("why hello! how are you?")
"hello"
```

10.5 Math

ceil(*value: decimal*) → int128

Round a decimal up to the nearest integer.

- `value`: Decimal value to round up

```
@external
@view
def foo(value: decimal) -> uint256:
    return ceil(value)
```

```
>>> ExampleContract.foo(3.1337)
4
```

floor(*value: decimal*) → int128

Round a decimal down to the nearest integer.

- `value`: Decimal value to round down

```
@external
@view
def foo(value: decimal) -> uint256:
    return floor(value)
```

```
>>> ExampleContract.foo(3.1337)
3
```

max (*a: numeric*, *b: numeric*) → numeric

Return the creater value of a and b. The input values may be any numeric type as long as they are both of the same type. The output value is the same as the input values.

```
@external
@view
def foo(a: uint256, b: uint256) -> uint256:
    return max(a, b)
```

```
>>> ExampleContract.foo(23, 42)
42
```

min (*a: numeric*, *b: numeric*) → numeric

Returns the lesser value of a and b. The input values may be any numeric type as long as they are both of the same type. The output value is the same as the input values.

```
@external
@view
def foo(a: uint256, b: uint256) -> uint256:
    return min(a, b)
```

```
>>> ExampleContract.foo(23, 42)
23
```

pow_mod256 (*a: uint256*, *b: uint256*) → uint256

Return the result of `a ** b % (2 ** 256)`.

This method is used to perform exponentiation without overflow checks.

```
@external
@view
def foo(a: uint256, b: uint256) -> uint256:
    return pow_mod256(a, b)
```

```
>>> ExampleContract.foo(2, 3)
8
>>> ExampleContract.foo(100, 100)
59041770658110225754900818312084884949620587934026984283048776718299468660736
```

sqrt (*d: decimal*) → decimal

Return the square root of the provided decimal number, using the Babylonian square root algorithm.

```
@external
@view
def foo(d: decimal) -> decimal:
    return sqrt(d)
```

```
>>> ExampleContract.foo(9.0)
3.0
```

uint256_addmod(*a: uint256*, *b: uint256*, *c: uint256*) → uint256

Return the modulo of `(a + b) % c`. Reverts if `c == 0`.

```
@external
@view
def foo(a: uint256, b: uint256, c: uint256) -> uint256:
    return uint256_addmod(a, b, c)
```

```
>>> (6 + 13) % 8
3
>>> ExampleContract.foo(6, 13, 8)
3
```

uint256_mulmod(*a: uint256*, *b: uint256*, *c: uint256*) → uint256

Return the modulo from `(a * b) % c`. Reverts if `c == 0`.

```
@external
@view
def foo(a: uint256, b: uint256, c: uint256) -> uint256:
    return uint256_mulmod(a, b, c)
```

```
>>> (11 * 2) % 5
2
>>> ExampleContract.foo(11, 2, 5)
2
```

10.6 Utilities

as_wei_value(*_value*, *unit: str*) → uint256

Take an amount of ether currency specified by a number and a unit and return the integer quantity of wei equivalent to that amount.

- `_value`: Value for the ether unit. Any numeric type may be used, however the value cannot be negative.
- `unit`: Ether unit name (e.g. `"wei"`, `"ether"`, `"gwei"`, etc.) indicating the denomination of `_value`. Must be given as a literal string.

```
@external
@view
def foo(s: String[32]) -> uint256:
    return as_wei_value(1.337, "ether")
```

```
>>> ExampleContract.foo(1)
1337000000000000000
```

blockhash(*block_num: uint256*) → bytes32

Return the hash of the block at the specified height.

Note: The EVM only provides access to the most 256 blocks. This function returns `EMPTY_BYTES32` if the block number is greater than or equal to the current block number or more than 256 blocks behind the current block.

```
@external
@view
def foo() -> bytes32:
    return blockhash(block.number - 16)
```

```
>>> ExampleContract.foo()
0xf3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
```

empty (*typename*) → Any

Return a value which is the default (zeroed) value of its type. Useful for initializing new memory variables.

- `typename`: Name of the type

```
@external
@view
def foo():
    x: uint256[2][5] = empty(uint256[2][5])
```

len (*b: Union[Bytes, String]*) → uint256

Return the length of a given `Bytes` or `String`.

```
@external
@view
def foo(s: String[32]) -> uint256:
    return len(s)
```

```
>>> ExampleContract.foo("hello")
5
```

method_id (*method, output_type: type = Bytes[4]*) → Union[bytes32, Bytes[4]]

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

- `method`: Method declaration as given as a literal string
- `output_type`: The type of output (`Bytes[4]` or `bytes32`). Defaults to `Bytes[4]`.

Returns a value of the type specified by `output_type`.

```
@external
@view
def foo() -> Bytes[4]:
    return method_id('transfer(address,uint256)', output_type=Bytes[4])
```

```
>>> ExampleContract.foo()
b"\xa9\x05\x9c\xbb"
```

CHAPTER 11

Interfaces

An interface is a set of function definitions used to enable communication between smart contracts. A contract interface defines all of that contract's externally available functions. By importing the interface, your contract now knows how to call these functions in other contracts.

11.1 Declaring and using Interfaces

Interfaces can be added to contracts either through inline definition, or by importing them from a seperate file.

The `interface` keyword is used to define an inline external interface:

```
interface FooBar:
    def calculate() -> uint256: view
    def test1(): nonpayable
```

The defined interface can then be use to make external calls, given a contract address:

```
@external
def test(some_address: address):
    FooBar(some_address).calculate()
```

The interface name can also be used as a type annotation for storage variables. You then assign an address value to the variable to access that interface. Note that assignment of an address requires the value to be cast using the interface type e.g. FooBar(<address_var>):

```
foobar_contract: FooBar

@external
def __init__(foobar_address: address):
    self.foobar_contract = FooBar(foobar_address)

@external
```

(continues on next page)

(continued from previous page)

```
def test():
    self.foobar_contract.calculate()
```

Specifying `payable` or `nonpayable` annotation indicates that the call made to the external contract will be able to alter storage, whereas the `view` `pure` call will use a `STATICCALL` ensuring no storage can be altered during execution. Additionally, `payable` allows non-zero value to be sent along with the call.

```
interface FooBar:
    def calculate() -> uint256: pure
    def query() -> uint256: view
    def update(): nonpayable
    def pay(): payable

@external
def test(some_address: address):
    FooBar(some_address).calculate()  # cannot change storage
    FooBar(some_address).query()  # cannot change storage, but reads itself
    FooBar(some_address).update()  # storage can be altered
    FooBar(some_address).pay(value=1)  # storage can be altered, and value can be sent
```

11.2 Importing Interfaces

Interfaces are imported with `import` or `from ... import` statements.

Imported interfaces are written using standard Vyper syntax. The body of each function is ignored when the interface is imported. If you are defining a standalone interface, it is normally specified by using a `pass` statement:

```
@external
def test1():
    pass

@external
def calculate() -> uint256:
    pass
```

You can also import a fully implemented contract and Vyper will automatically convert it to an interface. It is even possible for a contract to import itself to gain access to it's own interface.

```
import greeter as Greeter

name: public(String[10])

@external
def __init__(_name: String[10]):
    self.name = _name

@view
@external
def greet() -> String[16]:
    return concat("Hello ", Greeter(msg.sender).name())
```

11.2.1 Imports via `import`

With absolute `import` statements, you **must** include an alias as a name for the imported package. In the following example, failing to include `as Foo` will raise a compile error:

```
import contract.foo as Foo
```

11.2.2 Imports via `from ... import`

Using `from` you can perform both absolute and relative imports. You may optionally include an alias - if you do not, the name of the interface will be the same as the file.

```
# without an alias
from contract import foo

# with an alias
from contract import foo as Foo
```

Relative imports are possible by prepending dots to the contract name. A single leading dot indicates a relative import starting with the current package. Two leading dots indicate a relative import from the parent of the current package:

```
from . import foo
from ..interfaces import baz
```

11.2.3 Searching For Interface Files

When looking for a file to import Vyper will first search relative to the same folder as the contract being compiled. For absolute imports, it also searches relative to the root path for the project. Vyper checks for the file name with a `.vy` suffix first, then `.json`.

When using the command line compiler, the root path defaults to to the current working directory. You can change it with the `-p` flag:

```
$ vyper my_project/contracts/my_contract.vy -p my_project
```

In the above example, the `my_project` folder is set as the root path. A contract cannot perform a relative import that goes beyond the top-level folder.

11.3 Built-in Interfaces

Vyper includes common built-in interfaces such as ERC20 and ERC721. These are imported from `vyper.interfaces`:

```
from vyper.interfaces import ERC20

implements: ERC20
```

You can see all the available built-in interfaces in the Vyper GitHub repo.

11.4 Implementing an Interface

You can define an interface for your contract with the `implements` statement:

```
import an_interface as FooBarInterface

implements: FooBarInterface
```

This imports the defined interface from the vyper file at `an_interface.vy` (or `an_interface.json` if using ABI json interface type) and ensures your current contract implements all the necessary external functions. If any interface functions are not included in the contract, it will fail to compile. This is especially useful when developing contracts around well-defined standards such as ERC20.

11.5 Extracting Interfaces

Vyper has a built-in format option to allow you to make your own vyper interfaces easily.

```
$ vyper -f interface examples/voting/ballot.vy

# Functions

@view
@external
def delegated(addr: address) -> bool:
    pass

# ...
```

If you want to do an external call to another contract, vyper provides an external interface extract utility as well.

```
$ vyper -f external_interface examples/voting/ballot.vy

# External Contracts
interface Ballot:
    def delegated(addr: address) -> bool: view
    def directlyVoted(addr: address) -> bool: view
    def giveRightToVote(voter: address): nonpayable
    def forwardWeight(delegate_with_weight_to_forward: address): nonpayable
    # ...
```

The output can then easily be copy-pasted to be consumed.

CHAPTER 12

Event Logging

Vyper can log events to be caught and displayed by user interfaces.

12.1 Example of Logging

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

```
# Events of the token.
event Transfer:
    sender: indexed(address)
    receiver: indexed(address)
    value: uint256

event Approval:
    owner: indexed(address)
    spender: indexed(address)
    value: uint256

# Transfer some tokens from message sender to another address
def transfer(_to : address, _value : uint256) -> bool:

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

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

Let's look at what this is doing.

1. 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.

2. 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.returnValues;
        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.

12.2 Declaring Events

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

```
event Transfer:
    sender: indexed(address)
    receiver: indexed(address)
    value: uint256
```

Event declarations look similar to struct declarations, containing one or more arguments that are passed to the event. Typical events will contain two kinds of arguments:

- **Indexed** arguments, which can be searched for by listeners. Each indexed argument is identified 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.

It is also possible to create an event with no arguments. In this case, use the `pass` statement:

```
event Foo: pass
```

12.3 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 `log` statement:

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

The order and types of arguments given must match the order of arguments used when declaring the event..

12.4 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.returnValues`. 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.

CHAPTER 13

NatSpec Metadata

Vyper contracts can use a special form of docstring to provide rich documentation for functions, return variables and more. This special form is named the Ethereum Natural Language Specification Format (NatSpec).

This documentation is segmented into developer-focused messages and end-user-facing messages. These messages may be shown to the end user (the human) at the time that they will interact with the contract (i.e. sign a transaction).

13.1 Example

Vyper supports structured documentation for contracts and external functions using the doxygen notation format.

Note: The compiler does not parse docstrings of internal functions. You are welcome to NatSpec in comments for internal functions, however they are not processed or included in the compiler output.

```
"""
@title A simulator for Bug Bunny, the most famous Rabbit
@license MIT
@author Warned Bros
@notice You can use this contract for only the most basic simulation
@dev
    Simply chewing a carrot does not count, carrots must pass
    the throat to be considered eaten
"""

@external
@payable
def doesEat(food: string[30], qty: uint256) -> bool:
    """
    @notice Determine if Bugs will accept `qty` of `food` to eat
    @dev Compares the entire string and does not rely on a hash
    @param food The name of a food to evaluate (in English)
```

(continues on next page)

(continued from previous page)

```
    @param qty The number of food items to evaluate
    @return True if Bugs will eat it, False otherwise
    """
```

13.2 Tags

All tags are optional. The following table explains the purpose of each NatSpec tag and where it may be used:

Tag	Description	Context
`@title`	Title that describes the contract	contract
`@licence`	License of the contract	contract
`@author`	Name of the author	contract, function
`@notice`	Explain to an end user what this does	contract, function
`@dev`	Explain to a developer any extra details	contract, function
`@param`	Documents a single parameter	function
`@return`	Documents one or all return variable(s)	function

Some rules / restrictions:

1. A single tag description may span multiple lines. All whitespace between lines is interpreted as a single space.
2. If a docstring is included with no NatSpec tags, it is interpreted as a `@notice`.
3. Each use of `@param` must be followed by the name of an input argument. Including invalid or duplicate argument names raises a *NatSpecSyntaxException*.
4. The preferred use of `@return` is one entry for each output value, however you may also use it once for all outputs. Including more `@return` values than output values raises a *NatSpecSyntaxException*.

13.3 Documentation Output

When parsed by the compiler, documentation such as the one from the above example will produce two different JSON outputs. One is meant to be consumed by the end user as a notice when a function is executed and the other to be used by the developer.

If the above contract is saved as `carrots.vy` then you can generate the documentation using:

```
vyper -f userdoc,devdoc carrots.vy
```

13.3.1 User Documentation

The above documentation will produce the following user documentation JSON as output:

```
{
  "methods": {
    "doesEat(string,uint256)": {
      "notice": "Determine if Bugs will accept `qty` of `food` to eat"
    }
  },
```

(continues on next page)

(continued from previous page)

```
  "notice": "You can use this contract for only the most basic simulation"
}
```

Note that the key by which to find the methods is the function's canonical signature as defined in the contract ABI, not simply the function's name.

13.3.2 Developer Documentation

Apart from the user documentation file, a developer documentation JSON file should also be produced and should look like this:

```
{
  "author": "Warned Bros",
  "license": "MIT",
  "details": "Simply chewing a carrot does not count, carrots must pass the throat to␣
↪be considered eaten",
  "methods": {
    "doesEat(string,uint256)": {
      "details" : "Compares the entire string and does not rely on a hash",
      "params": {
        "food": "The name of a food to evaluate (in English)",
        "qty": "The number of food items to evaluate"
      },
      "returns": {
        "_0": "True if Bugs will eat it, False otherwise"
      }
    }
  },
  "title" : "A simulator for Bug Bunny, the most famous Rabbit"
}
```

CHAPTER 14

Compiling a Contract

14.1 Command-Line Compiler Tools

Vyper includes the following command-line scripts for compiling contracts:

- `vyper`: Compiles vyper contract files into `LLL` or bytecode
- `vyper-json`: Provides a JSON interface to the compiler

Note: The `--help` flag gives verbose explanations of how to use each of these scripts.

14.1.1 vyper

`vyper` provides command-line access to the compiler. It can generate various outputs including simple binaries, ASTs, interfaces and source mappings.

To compile a contract:

```
$ vyper yourFileName.vy
```

Include the `-f` flag to specify which output formats to return. Use `vyper --help` for a full list of output options.

```
$ vyper -f abi,bytecode,bytecode_runtime,ir,asm,source_map,method_identifiers␣
↪yourFileName.vy
```

The `-p` flag allows you to set a root path that is used when searching for interface files to import. If none is given, it will default to the current working directory. See *Searching For Interface Files* for more information.

```
$ vyper -p yourProject yourProject/yourFileName.vy
```

14.1.2 vyper-json

`vyper-json` provides a JSON interface for the compiler. It expects a *JSON formatted input* and returns the compilation result in a *JSON formatted output*.

To compile from JSON supplied via `stdin`:

```
$ vyper-json
```

To compile from a JSON file:

```
$ vyper-json yourProject.json
```

By default, the output is sent to `stdout`. To redirect to a file, use the `-o` flag:

```
$ vyper-json -o compiled.json
```

Importing Interfaces

`vyper-json` searches for imported interfaces in the following sequence:

1. Interfaces defined in the `interfaces` field of the input JSON
2. Derived interfaces generated from contracts in the `sources` field of the input JSON
3. (Optional) The local filesystem, if a root path was explicitely declared via the `-p` flag.

See *Searching For Interface Files* for more information on Vyper's import system.

14.2 Online Compilers

14.2.1 Remix IDE

Remix IDE is a compiler and Javascript VM for developing and testing contracts in Vyper as well as Solidity.

Note: While the vyper version of the Remix IDE compiler is updated on a regular basis it might be a bit behind the latest version found in the master branch of the repository. Make sure the byte code matches the output from your local compiler.

14.3 Setting the Target EVM Version

When you compile your contract code you can specify the Ethereum virtual machine version to compile for to avoid particular features or behaviours.

Warning: Compiling for the wrong EVM version can result in wrong, strange and failing behaviour. Please ensure, especially if running a private chain, that you use matching EVM versions.

When compiling via `vyper`, include the `--evm-version` flag:

```
$ vyper --evm-version [VERSION]
```

When using the JSON interface, include the "evmVersion" key within the "settings" field:

```
{
    "settings": {
        "evmVersion": "[VERSION]"
    }
}
```

14.3.1 Target Options

The following is a list of supported EVM versions, and changes in the compiler introduced with each version. Backward compatibility is not guaranteed between each version.

byzantium

- The oldest EVM version supported by Vyper.

constantinople

- The EXTCODEHASH opcode is accessible via address.codehash
- shift makes use of SHL/SHR opcodes.

petersburg

- The compiler behaves the same way as with consantinople.

istanbul (*default*)

- The CHAINID opcode is accessible via chain.id
- The SELFBALANCE opcode is used for calls to self.balance
- Gas estimates changed for SLOAD and BALANCE

14.4 Compiler Input and Output JSON Description

Especially when dealing with complex or automated setups, the recommended way to compile is to use *vyper-json* and the JSON-input-output interface.

Where possible, the Vyper JSON compiler formats follow those of Solidity.

14.4.1 Input JSON Description

The following example describes the expected input format of vyper-json. Comments are of course not permitted and used here only for explanatory purposes.

```
{
    // Required: Source code language. Must be set to "Vyper".
    "language": "Vyper",
    // Required
    // Source codes given here will be compiled.
    "sources": {
        "contracts/foo.vy": {
```

(continues on next page)

(continued from previous page)

```
            // Optional: keccak256 hash of the source file
            "keccak256": "0x234...",
            // Required: literal contents of the source file
            "content": "@external\ndef foo() -> bool:\n    return True"
        }
    },
    // Optional
    // Interfaces given here are made available for import by the sources
    // that are compiled. If the suffix is ".vy", the compiler will expect
    // a contract-as-interface using proper Vyper syntax. If the suffix is
    // "abi" the compiler will expect an ABI object.
    "interfaces": {
        "contracts/bar.vy": {
            "content": ""
        },
        "contracts/baz.json": {
            "abi": []
        }
    },
    // Optional
    "settings": {
        "evmVersion": "istanbul",  // EVM version to compile for. Can be byzantium,␣
↪constantinople, petersburg or istanbul.
        // The following is used to select desired outputs based on file names.
        // File names are given as keys, a star as a file name matches all files.
        // Outputs can also follow the Solidity format where second level keys
        // denoting contract names - all 2nd level outputs are applied to the file.
        //
        // To select all possible compiler outputs: "outputSelection: { '*': ["*"] }"
        // Note that this might slow down the compilation process needlessly.
        //
        // The available output types are as follows:
        //
        //    abi - The contract ABI
        //    ast - Abstract syntax tree
        //    interface - Derived interface of the contract, in proper Vyper syntax
        //    ir - LLL intermediate representation of the code
        //    userdoc - Natspec user documentation
        //    devdoc - Natspec developer documentation
        //    evm.bytecode.object - Bytecode object
        //    evm.bytecode.opcodes - Opcodes list
        //    evm.deployedBytecode.object - Deployed bytecode object
        //    evm.deployedBytecode.opcodes - Deployed opcodes list
        //    evm.deployedBytecode.sourceMap - Deployed source mapping (useful for␣
↪debugging)
        //    evm.methodIdentifiers - The list of function hashes
        //
        // Using `evm`, `evm.bytecode`, etc. will select every target part of that␣
↪output.
        // Additionally, `*` can be used as a wildcard to request everything.
        //
        "outputSelection": {
            "*": ["evm.bytecode", "abi"],  // Enable the abi and bytecode outputs for␣
↪every single contract
            "contracts/foo.vy": ["ast"]  // Enable the ast output for contracts/foo.vy
        }
    }
```

(continues on next page)

(continued from previous page)

```
}
```

14.4.2 Output JSON Description

The following example describes the output format of `vyper-json`. Comments are of course not permitted and used here only for explanatory purposes.

```
{
    // The compiler version used to generate the JSON
    "compiler": "vyper-0.1.0b12",
    // Optional: not present if no errors/warnings were encountered
    "errors": [
        {
        // Optional: Location within the source file.
        "sourceLocation": {
            "file": "source_file.vy",
            "lineno": 5,
            "col_offset": 11
        },
        // Mandatory: Exception type, such as "JSONError", "StructureException", etc.
        "type": "TypeMismatch",
        // Mandatory: Component where the error originated, such as "json", "compiler
↪", "vyper", etc.
        "component": "compiler",
        // Mandatory ("error" or "warning")
        "severity": "error",
        // Mandatory
        "message": "Unsupported type conversion: int128 to bool"
        // Optional: the message formatted with source location
        "formattedMessage": "line 5:11 Unsupported type conversion: int128 to bool"
        }
    ],
    // This contains the file-level outputs. Can be limited/filtered by the␣
↪outputSelection settings.
    "sources": {
        "source_file.vy": {
            // Identifier of the source (used in source maps)
            "id": 0,
            // The AST object
            "ast": {},
        }
    },
    // This contains the contract-level outputs. Can be limited/filtered by the␣
↪outputSelection settings.
    "contracts": {
        "source_file.vy": {
            // The contract name will always be the file name without a suffix
            "source_file": {
                // The Ethereum Contract ABI.
                // See https://github.com/ethereum/wiki/wiki/Ethereum-Contract-ABI
                "abi": [],
                // Natspec developer documentation
                "devdoc": {},
                // Intermediate representation (string)
                "ir": "",
```

(continues on next page)

(continued from previous page)

```
                // Natspec developer documentation
                "userdoc": {},
                // EVM-related outputs
                "evm": {
                    "bytecode": {
                        // The bytecode as a hex string.
                        "object": "00fe",
                        // Opcodes list (string)
                        "opcodes": ""
                    },
                    "deployedBytecode": {
                        // The deployed bytecode as a hex string.
                        "object": "00fe",
                        // Deployed opcodes list (string)
                        "opcodes": "",
                        // The deployed source mapping as a string.
                        "sourceMap": ""
                    },
                    // The list of function hashes
                    "methodIdentifiers": {
                        "delegate(address)": "5c19a95c"
                    }
                }
            }
        }
    }
}
```

Errors

Each error includes a `component` field, indicating the stage at which it occurred:

- `json`: Errors that occur while parsing the input JSON. Usually a result of invalid JSON or a required value that is missing.
- `parser`: Errors that occur while parsing the contracts. Usually a result of invalid Vyper syntax.
- `compiler`: Errors that occur while compiling the contracts.
- `vyper`: Unexpected errors that occur within Vyper. If you receive an error of this type, please open an issue.

You can also use the `--traceback` flag to receive a standard Python traceback when an error is encountered.

CHAPTER 15

Compiler Exceptions

Vyper raises one or more of the following exceptions when an issue is encountered while compiling a contract.

Whenever possible, exceptions include a source highlight displaying the location of the error within the code:

```
vyper.exceptions.VariableDeclarationException: line 79:17 Persistent variable␣
↪undeclared: highstBid
     78     # If bid is less than highest bid, bid fails
---> 79     if (value <= self.highstBid):
-------------------------^
     80         return False
```

exception ArgumentException
Raises when calling a function with invalid arguments, for example an incorrect number of positional arguments or an invalid keyword argument.

exception CallViolation
Raises on an illegal function call, such as attempting to call between two external functions.

exception ArrayIndexException
Raises when an array index is out of bounds.

exception EventDeclarationException
Raises when an event declaration is invalid.

exception EvmVersionException
Raises when a contract contains an action that cannot be performed with the active EVM ruleset.

exception FunctionDeclarationException
Raises when a function declaration is invalid, for example because of incorrect or mismatched return values.

exception ImmutableViolation
Raises when attempting to perform a change a variable, constant or definition that cannot be changed. For example, trying to update a constant, or trying to assign to a function definition.

exception InterfaceViolation
Raises when an interface is not fully implemented.

exception InvalidAttribute
Raises on a reference to an attribute that does not exist.

exception InvalidLiteral
Raises when no valid type can be found for a literal value.

```
@external
def foo():
    bar: decimal = 3.123456789123456789
```

This example raises `InvalidLiteral` because the given literal value has too many decimal places and so cannot be assigned any valid Vyper type.

exception InvalidOperation
Raises when using an invalid operator for a given type.

```
@external
def foo():
    a: String[10] = "hello" * 2
```

This example raises `InvalidOperation` because multiplication is not possible on string types.

exception InvalidReference
Raises on an invalid reference to an existing definition.

```
baz: int128

@external
def foo():
    bar: int128 = baz
```

This example raises `InvalidReference` because `baz` is a storage variable. The reference to it should be written as `self.baz`.

exception InvalidType
Raises when using an invalid literal value for the given type.

```
@external
def foo():
    bar: int128 = 3.5
```

This example raises `InvalidType` because `3.5` is a valid literal value, but cannot be cast as `int128`.

exception IteratorException
Raises when an iterator is constructed or used incorrectly.

exception JSONError
Raises when the compiler JSON input is malformed.

exception NamespaceCollision
Raises when attempting to assign a variable to a name that is already in use.

exception NatSpecSyntaxException
Raises when a contract contains an invalid *NatSpec* docstring.

```
vyper.exceptions.SyntaxException: line 14:5 No description given for tag '@param'
     13     @dev the feet are sticky like rice
---> 14     @param
-------------^
     15     @return always True
```

exception NonPayableViolation

Raises when attempting to access `msg.value` from within a function that has not been marked as `@payable`.

```
@public
def _foo():
    bar: uint256 = msg.value
```

exception OverflowException

Raises when a numeric value is out of bounds for the given type.

exception StateAccessViolation

Raises when attempting to perform a modifying action within view-only or stateless context. For example, writing to storage in a `@view` function, reading from storage in a `@pure` function.

exception StructureException

Raises on syntax that is parsable, but invalid in some way.

```
vyper.exceptions.StructureException: line 181:0 Invalid top-level statement
     180
---> 181 '''
---------^
     182
```

exception SyntaxException

Raises on invalid syntax that cannot be parsed.

```
vyper.exceptions.SyntaxException: line 4:20 invalid syntax
        3 struct Bid:
---> 4   blindedBid bytes32
---------------------------^
        5   deposit: uint256
```

exception TypeMismatch

Raises when attempting to perform an action between two or more objects with known, dislike types.

```
@external
def foo(:
    bar: int128 = 3
    foo: decimal = 4.2

    if foo + bar > 4:
        pass
```

`foo` has a type of `int128` and `bar` has a type of `decimal`, so attempting to add them together raises a `TypeMismatch`.

exception UndeclaredDefinition

Raises when attempting to access an object that has not been declared.

exception VariableDeclarationException

Raises on an invalid variable declaration.

```
vyper.exceptions.VariableDeclarationException: line 79:17 Persistent variable␣
↪undeclared: highstBid
     78      # If bid is less than highest bid, bid fails
---> 79      if (value <= self.highstBid):
-------------------------^
     80          return False
```

exception VersionException
Raises when a contract version string is malformed or incompatible with the current compiler version.

exception ZeroDivisionException
Raises when a divide by zero or modulo zero situation arises.

15.1 CompilerPanic

exception CompilerPanic

```
$ vyper v.vy
Error compiling: v.vy
vyper.exceptions.CompilerPanic: Number of times repeated
must be a constant nonzero positive integer: 0 Please create an issue.
```

A compiler panic error indicates that there is a problem internally to the compiler and an issue should be reported right away on the Vyper Github page. Open an issue if you are experiencing this error. Please Open an Issue

CHAPTER 16

Deploying a Contract

Once you are ready to deploy your contract to a public test net or the main net, you have several options:

- Take the bytecode generated by the vyper compiler and manually deploy it through mist or geth:

```
vyper yourFileName.vy
# returns bytecode
```

- Take the byte code and ABI and depoly it with your current browser on myetherwallet's contract menu:

```
vyper -f abi yourFileName.vy
# returns ABI
```

- Use the remote compiler provided by the Remix IDE to compile and deploy your contract on your net of choice. Remix also provides a JavaScript VM to test deploy your contract.

Note: While the vyper version of the Remix IDE compiler is updated on a regular basis it might be a bit behind the latest version found in the master branch of the repository. Make sure the byte code matches the output from your local compiler.

CHAPTER 17

Testing a Contract

For testing Vyper contracts we recommend the use of pytest along with one of the following packages:

- Brownie: A development and testing framework for smart contracts targeting the Ethereum Virtual Machine
- Ethereum Tester: A tool suite for testing ethereum applications

Example usage for each package is provided in the sections listed below.

17.1 Testing with Brownie

Brownie is a Python-based development and testing framework for smart contracts. It includes a pytest plugin with fixtures that simplify testing your contract.

This section provides a quick overview of testing with Brownie. To learn more, you can view the Brownie documentation on writing unit tests or join the Gitter channel.

17.1.1 Getting Started

In order to use Brownie for testing you must first initialize a new project. Create a new directory for the project, and from within that directory type:

```
$ brownie init
```

This will create an empty project structure within the directory. Store your contract sources within the project's `contracts/` directory and your tests within `tests/`.

17.1.2 Writing a Basic Test

Assume the following simple contract `Storage.vy`. It has a single integer variable and a function to set that value.

```
storedData: public(int128)

@external
def __init__(_x: int128):
  self.storedData = _x

@external
def set(_x: int128):
  self.storedData = _x
```

We create a test file `tests/test_storage.py` where we write our tests in pytest style.

```
import pytest

INITIAL_VALUE = 4


@pytest.fixture
def storage_contract(Storage, accounts):
    # deploy the contract with the initial value as a constructor argument
    yield Storage.deploy(INITIAL_VALUE, {'from': accounts[0]})


def test_initial_state(storage_contract):
    # Check if the constructor of the contract is set up properly
    assert storage_contract.storedData() == INITIAL_VALUE


def test_set(storage_contract, accounts):
    # set the value to 10
    storage_contract.set(10, {'from': accounts[0]})
    assert storage_contract.storedData() == 10  # Directly access storedData

    # set the value to -5
    storage_contract.set(-5, {'from': accounts[0]})
    assert storage_contract.storedData() == -5
```

In this example we are using two fixtures which are provided by Brownie:

- `accounts` provides access to the `Accounts` container, containing all of your local accounts
- `Storage` is a dynamically named fixture that provides access to a `ContractContainer` object, used to deploy your contract

Note: To run the tests, use the `brownie test` command from the root directory of your project.

17.1.3 Testing Events

For the remaining examples, we expand our simple storage contract to include an event and two conditions for a failed transaction: `AdvancedStorage.vy`

```
event DataChange:
    setter: indexed(address)
    value: int128

```

(continues on next page)

(continued from previous page)

```
storedData: public(int128)

@external
def __init__(_x: int128):
  self.storedData = _x

@external
def set(_x: int128):
  assert _x >= 0, "No negative values"
  assert self.storedData < 100, "Storage is locked when 100 or more is stored"
  self.storedData = _x
  log DataChange(msg.sender, _x)

@external
def reset():
  self.storedData = 0
```

To test events, we examine the `TransactionReceipt` object which is returned after each successful transaction. It contains an `events` member with information about events that fired.

```
import brownie

INITIAL_VALUE = 4


@pytest.fixture
def adv_storage_contract(AdvancedStorage, accounts):
    yield AdvancedStorage.deploy(INITIAL_VALUE, {'from': accounts[0]})

def test_events(adv_storage_contract, accounts):
    tx1 = adv_storage_contract.set(10, {'from': accounts[0]})
    tx2 = adv_storage_contract.set(20, {'from': accounts[1]})
    tx3 = adv_storage_contract.reset({'from': accounts[0]})

    # Check log contents
    assert len(tx1.events) == 1
    assert tx1.events[0]['value'] == 10

    assert len(tx2.events) == 1
    assert tx2.events[0]['setter'] == accounts[1]

    assert not tx3.events   # tx3 does not generate a log
```

17.1.4 Handling Reverted Transactions

Transactions that revert raise a `VirtualMachineError` exception. To write assertions around this you can use `brownie.reverts` as a context manager. It functions very similarly to `pytest.raises`.

`brownie.reverts` optionally accepts a string as an argument. If given, the error string returned by the transaction must match it in order for the test to pass.

```
import brownie

INITIAL_VALUE = 4

```

(continues on next page)

(continued from previous page)

```

@pytest.fixture
def adv_storage_contract(AdvancedStorage, accounts):
    yield AdvancedStorage.deploy(INITIAL_VALUE, {'from': accounts[0]})


def test_failed_transactions(adv_storage_contract, accounts):
    # Try to set the storage to a negative amount
    with brownie.reverts("No negative values"):
        adv_storage_contract.set(-10, {"from": accounts[1]})

    # Lock the contract by storing more than 100. Then try to change the value

    adv_storage_contract.set(150, {"from": accounts[1]})
    with brownie.reverts("Storage is locked when 100 or more is stored"):
        adv_storage_contract.set(10, {"from": accounts[1]})

    # Reset the contract and try to change the value
    adv_storage_contract.reset({"from": accounts[1]})
    adv_storage_contract.set(10, {"from": accounts[1]})
    assert adv_storage_contract.storedData() == 10
```

17.2 Testing with Ethereum Tester

Ethereum Tester is a tool suite for testing Ethereum based applications.

This section provides a quick overview of testing with `eth-tester`. To learn more, you can view the documentation at the Github repo or join the Gitter channel.

17.2.1 Getting Started

Prior to testing, the Vyper specific contract conversion and the blockchain related fixtures need to be set up. These fixtures will be used in every test file and should therefore be defined in conftest.py.

Note: Since the testing is done in the pytest framework, you can make use of pytest.ini, tox.ini and setup.cfg and you can use most IDEs' pytest plugins.

```
import pytest
from eth_tester import EthereumTester, PyEVMBackend
from eth_tester.exceptions import TransactionFailed
from eth_utils.toolz import compose
from web3 import Web3
from web3.contract import Contract, mk_collision_prop
from web3.providers.eth_tester import EthereumTesterProvider

from vyper import compiler

from .grammar.conftest import get_lark_grammar

LARK_GRAMMAR = get_lark_grammar()

```

(continues on next page)

(continued from previous page)

```python

class VyperMethod:
    ALLOWED_MODIFIERS = {"call", "estimateGas", "transact", "buildTransaction"}

    def __init__(self, function, normalizers=None):
        self._function = function
        self._function._return_data_normalizers = normalizers

    def __call__(self, *args, **kwargs):
        return self.__prepared_function(*args, **kwargs)

    def __prepared_function(self, *args, **kwargs):
        if not kwargs:
            modifier, modifier_dict = "call", {}
            fn_abi = [
                x
                for x in self._function.contract_abi
                if x.get("name") == self._function.function_identifier
            ].pop()
            # To make tests faster just supply some high gas value.
            modifier_dict.update({"gas": fn_abi.get("gas", 0) + 50000})
        elif len(kwargs) == 1:
            modifier, modifier_dict = kwargs.popitem()
            if modifier not in self.ALLOWED_MODIFIERS:
                raise TypeError(f"The only allowed keyword arguments are: {self.ALLOWED_MODIFIERS}")
        else:
            raise TypeError(f"Use up to one keyword argument, one of: {self.ALLOWED_MODIFIERS}")
        return getattr(self._function(*args), modifier)(modifier_dict)


class VyperContract:
    """
    An alternative Contract Factory which invokes all methods as `call()`,
    unless you add a keyword argument. The keyword argument assigns the prep method.
    This call
    > contract.withdraw(amount, transact={'from': eth.accounts[1], 'gas': 100000, ...})
    is equivalent to this call in the classic contract:
    > contract.functions.withdraw(amount).transact({'from': eth.accounts[1], 'gas': 100000, ...})
    """

    def __init__(self, classic_contract, method_class=VyperMethod):
        classic_contract._return_data_normalizers += CONCISE_NORMALIZERS
        self._classic_contract = classic_contract
        self.address = self._classic_contract.address
        protected_fn_names = [fn for fn in dir(self) if not fn.endswith("__")]
        for fn_name in self._classic_contract.functions:
            # Override namespace collisions
            if fn_name in protected_fn_names:
                _concise_method = mk_collision_prop(fn_name)
            else:
                _classic_method = getattr(self._classic_contract.functions, fn_name)
                _concise_method = method_class(
                    _classic_method, self._classic_contract._return_data_normalizers
```

(continues on next page)

(continued from previous page)

```
                )
            setattr(self, fn_name, _concise_method)

    @classmethod
    def factory(cls, *args, **kwargs):
        return compose(cls, Contract.factory(*args, **kwargs))


def _none_addr(datatype, data):
    if datatype == "address" and int(data, base=16) == 0:
        return (datatype, None)
    else:
        return (datatype, data)


CONCISE_NORMALIZERS = (_none_addr,)


@pytest.fixture
def tester():
    custom_genesis = PyEVMBackend._generate_genesis_params(overrides={"gas_limit": 4500000})
    backend = PyEVMBackend(genesis_parameters=custom_genesis)
    return EthereumTester(backend=backend)


def zero_gas_price_strategy(web3, transaction_params=None):
    return 0  # zero gas price makes testing simpler.


@pytest.fixture
def w3(tester):
    w3 = Web3(EthereumTesterProvider(tester))
    w3.eth.setGasPriceStrategy(zero_gas_price_strategy)
    return w3


def _get_contract(w3, source_code, *args, **kwargs):
    out = compiler.compile_code(
        source_code,
        ["abi", "bytecode"],
        interface_codes=kwargs.pop("interface_codes", None),
        evm_version=kwargs.pop("evm_version", None),
    )
    LARK_GRAMMAR.parse(source_code + "\n")  # Test grammar.
    abi = out["abi"]
    bytecode = out["bytecode"]
    value = kwargs.pop("value_in_eth", 0) * 10 ** 18  # Handle deploying with an eth value.
    c = w3.eth.contract(abi=abi, bytecode=bytecode)
    deploy_transaction = c.constructor(*args)
    tx_info = {
        "from": w3.eth.accounts[0],
        "value": value,
        "gasPrice": 0,
    }
    tx_info.update(kwargs)
```

(continues on next page)

(continued from previous page)

```
    tx_hash = deploy_transaction.transact(tx_info)
    address = w3.eth.getTransactionReceipt(tx_hash)["contractAddress"]
    contract = w3.eth.contract(
        address, abi=abi, bytecode=bytecode, ContractFactoryClass=VyperContract,
    )
    return contract


@pytest.fixture
def get_contract(w3):
    def get_contract(source_code, *args, **kwargs):
        return _get_contract(w3, source_code, *args, **kwargs)

    return get_contract


@pytest.fixture
def get_logs(w3):
    def get_logs(tx_hash, c, event_name):
        tx_receipt = w3.eth.getTransactionReceipt(tx_hash)
        logs = c._classic_contract.events[event_name]().processReceipt(tx_receipt)
        return logs

    return get_logs


@pytest.fixture
def assert_tx_failed(tester):
    def assert_tx_failed(function_to_test, exception=TransactionFailed, exc_
↪text=None):
        snapshot_id = tester.take_snapshot()
        with pytest.raises(exception) as excinfo:
            function_to_test()
        tester.revert_to_snapshot(snapshot_id)
        if exc_text:
            assert exc_text in str(excinfo.value)

    return assert_tx_failed
```

The final two fixtures are optional and will be discussed later. The rest of this chapter assumes that you have this code set up in your `conftest.py` file.

Alternatively, you can import the fixtures to `conftest.py` or use pytest plugins.

17.2.2 Writing a Basic Test

Assume the following simple contract `storage.vy`. It has a single integer variable and a function to set that value.

```
storedData: public(int128)

@external
def __init__(_x: int128):
  self.storedData = _x

@external
```

(continues on next page)

(continued from previous page)

```
def set(_x: int128):
  self.storedData = _x
```

We create a test file `test_storage.py` where we write our tests in pytest style.

```
import pytest

INITIAL_VALUE = 4


@pytest.fixture
def storage_contract(w3, get_contract):
    with open("examples/storage/storage.vy") as f:
        contract_code = f.read()
        # Pass constructor variables directly to the contract
        contract = get_contract(contract_code, INITIAL_VALUE)
    return contract


def test_initial_state(storage_contract):
    # Check if the constructor of the contract is set up properly
    assert storage_contract.storedData() == INITIAL_VALUE


def test_set(w3, storage_contract):
    k0 = w3.eth.accounts[0]

    # Let k0 try to set the value to 10
    storage_contract.set(10, transact={"from": k0})
    assert storage_contract.storedData() == 10  # Directly access storedData

    # Let k0 try to set the value to -5
    storage_contract.set(-5, transact={"from": k0})
    assert storage_contract.storedData() == -5
```

First we create a fixture for the contract which will compile our contract and set up a Web3 contract object. We then use this fixture for our test functions to interact with the contract.

Note: To run the tests, call `pytest` or `python -m pytest` from your project directory.

17.2.3 Events and Failed Transactions

To test events and failed transactions we expand our simple storage contract to include an event and two conditions for a failed transaction: `advanced_storage.vy`

```
event DataChange:
    setter: indexed(address)
    value: int128

storedData: public(int128)

@external
def __init__(_x: int128):
```

(continues on next page)

(continued from previous page)

```
  self.storedData = _x

@external
def set(_x: int128):
  assert _x >= 0, "No negative values"
  assert self.storedData < 100, "Storage is locked when 100 or more is stored"
  self.storedData = _x
  log DataChange(msg.sender, _x)

@external
def reset():
  self.storedData = 0
```

Next, we take a look at the two fixtures that will allow us to read the event logs and to check for failed transactions.

```
@pytest.fixture
def assert_tx_failed(tester):
    def assert_tx_failed(function_to_test, exception=TransactionFailed, exc_
↪text=None):
        snapshot_id = tester.take_snapshot()
        with pytest.raises(exception) as excinfo:
            function_to_test()
        tester.revert_to_snapshot(snapshot_id)
        if exc_text:
            assert exc_text in str(excinfo.value)

    return assert_tx_failed
```

The fixture to assert failed transactions defaults to check for a `TransactionFailed` exception, but can be used to check for different exceptions too, as shown below. Also note that the chain gets reverted to the state before the failed transaction.

```
@pytest.fixture
def get_logs(w3):
    def get_logs(tx_hash, c, event_name):
        tx_receipt = w3.eth.getTransactionReceipt(tx_hash)
        logs = c._classic_contract.events[event_name]().processReceipt(tx_receipt)
        return logs

    return get_logs
```

This fixture will return a tuple with all the logs for a certain event and transaction. The length of the tuple equals the number of events (of the specified type) logged and should be checked first.

Finally, we create a new file `test_advanced_storage.py` where we use the new fixtures to test failed transactions and events.

```
import pytest
from web3.exceptions import ValidationError

INITIAL_VALUE = 4


@pytest.fixture
def adv_storage_contract(w3, get_contract):
    with open("examples/storage/advanced_storage.vy") as f:
        contract_code = f.read()
```

(continues on next page)

(continued from previous page)

```
        # Pass constructor variables directly to the contract
        contract = get_contract(contract_code, INITIAL_VALUE)
    return contract


def test_initial_state(adv_storage_contract):
    # Check if the constructor of the contract is set up properly
    assert adv_storage_contract.storedData() == INITIAL_VALUE


def test_failed_transactions(w3, adv_storage_contract, assert_tx_failed):
    k1 = w3.eth.accounts[1]

    # Try to set the storage to a negative amount
    assert_tx_failed(lambda: adv_storage_contract.set(-10, transact={"from": k1}))

    # Lock the contract by storing more than 100. Then try to change the value
    adv_storage_contract.set(150, transact={"from": k1})
    assert_tx_failed(lambda: adv_storage_contract.set(10, transact={"from": k1}))

    # Reset the contract and try to change the value
    adv_storage_contract.reset(transact={"from": k1})
    adv_storage_contract.set(10, transact={"from": k1})
    assert adv_storage_contract.storedData() == 10

    # Assert a different exception (ValidationError for non matching argument type)
    assert_tx_failed(
        lambda: adv_storage_contract.set("foo", transact={"from": k1}), ValidationError
    )

    # Assert a different exception that contains specific text
    assert_tx_failed(
        lambda: adv_storage_contract.set(1, 2, transact={"from": k1}),
        ValidationError,
        "invocation failed due to improper number of arguments",
    )


def test_events(w3, adv_storage_contract, get_logs):
    k1, k2 = w3.eth.accounts[:2]

    tx1 = adv_storage_contract.set(10, transact={"from": k1})
    tx2 = adv_storage_contract.set(20, transact={"from": k2})
    tx3 = adv_storage_contract.reset(transact={"from": k1})

    # Save DataChange logs from all three transactions
    logs1 = get_logs(tx1, adv_storage_contract, "DataChange")
    logs2 = get_logs(tx2, adv_storage_contract, "DataChange")
    logs3 = get_logs(tx3, adv_storage_contract, "DataChange")

    # Check log contents
    assert len(logs1) == 1
    assert logs1[0].args.value == 10

    assert len(logs2) == 1
    assert logs2[0].args.setter == k2
```

(continues on next page)

(continued from previous page)

```
    assert not logs3  # tx3 does not generate a log
```

CHAPTER 18

Release Notes

18.1 v0.2.6

Date released: 10-10-2020

Non-breaking changes and improvements:

- Release and reuse memory slots within the same function (#2188)
- Allow implicit use of `uint256` as iterator type in range-based for loops (#2180)
- Optimize clamping logic for `int128` (#2179)
- Calculate array index offsets at compile time where possible (#2187)
- Improved exception for invalid use of dynamically sized struct (#2189)
- Improved exception for incorrect arg count in function call (#2178)
- Improved exception for invalid subscript (#2177)

Fixes:

- Memory corruption issue when performing function calls inside a tuple or another function call (#2186)
- Incorrect function output when using multidimensional arrays (#2184)
- Reduced ambiguity bewteen `address` and `Bytes[20]` (#2191)

18.2 v0.2.5

Date released: 30-09-2020

Non-breaking changes and improvements:

- Improve exception on incorrect interface (#2131)
- Standalone binary preparation (#2134)

- Improve make freeze (#2135)
- Remove Excessive Scoping Rules on Local Variables (#2166)
- Optimize nonpayable check for contracts that do not accept ETH (#2172)
- Optimize safemath on division-by-zero with a literal divisor (#2173)
- Optimize multiple sequential memory-zeroings (#2174)
- Optimize size-limit checks for address and bool types (#2175)

Fixes:

- Constant folding on lhs of assignments (#2137)
- ABI issue with bytes and string arrays inside tuples (#2140)
- Returning struct from a external function gives error (#2143)
- Error messages with struct display all members (#2160)
- The returned struct value from the external call doesn't get stored properly (#2164)
- Improved exception on invalid function-scoped assignment (#2176)

18.3 v0.2.4

Date released: 03-08-2020

Non-breaking changes and improvements:

- Improve EOF Exceptions (#2115)
- Improve exception messaging for type mismatches (#2119)
- Ignore trailing newline tokens (#2120)

Fixes:

- Fix ABI translations for structs that are returned from functions (#2114)
- Raise when items that are not types are called (#2118)
- Ensure hex and decimal AST nodes are serializable (#2123)

18.4 v0.2.3

Date released: 16-07-2020

Non-breaking changes and improvements:

- Show contract names in raised exceptions (#2103)
- Adjust function offsets to not include decorators (#2102)
- Raise certain exception types immediately during module-scoped type checking (#2101)

Fixes:

- Pop `for` loop values from stack prior to returning (#2110)
- Type checking non-literal array index values (#2108)
- Meaningful output during `for` loop type checking (#2096)

18.5 v0.2.2

Date released: 04-07-2020

Fixes:

- Do not fold exponentiation to a negative power (#2089)
- Add repr for mappings (#2090)
- Literals are only validated once (#2093)

18.6 v0.2.1

Date released: 03-07-2020

This is a major breaking release of the Vyper compiler and language. It is also the first release following our versioning scheme (#1887).

Breaking changes:

- `@public` and `@private` function decorators have been renamed to `@external` and `@internal` (VIP #2065)
- The `@constant` decorator has been renamed to `@view` (VIP #2040)
- Type units have been removed (VIP #1881)
- Event declaraion syntax now resembles that of struct declarations (VIP #1864)
- `log` is now a statement (VIP #1864)
- Mapping declaration syntax changed to `HashMap[key_type, value_type]` (VIP #1969)
- Interfaces are now declared via the `interface` keyword instead of `contract` (VIP #1825)
- `bytes` and `string` types are now written as `Bytes` and `String` (#2080)
- `bytes` and `string` literals must now be bytes or regular strings, respectively. They are no longer interchangeable. (VIP #1876)
- `assert_modifiable` has been removed, you can now directly perform assertions on calls (#2050)
- `value` is no longer an allowable variable name in a function input (VIP #1877)
- The `slice` builtin function expects `uint256` for the `start` and `length` args (VIP #1986)
- `len` return type is now `uint256` (VIP #1979)
- `value` and `gas` kwargs for external function calls must be given as `uint256` (VIP #1878)
- The `outsize` kwarg in `raw_call` has been renamed to `max_outsize` (#1977)
- The `type` kwarg in `extract32` has been renamed to `output_type` (#2036)
- Public array getters now use `uint256` for their input argument(s) (VIP #1983)
- Public struct getters now return all values of a struct (#2064)
- `RLPList` has been removed (VIP #1866)

The following non-breaking VIPs and features were implemented:

- Implement boolean condition short circuiting (VIP #1817)
- Add the `empty` builtin function for zero-ing a value (#1676)

- Refactor of the compiler process resulting in an almost 5x performance boost! (#1962)
- Support ABI State Mutability Fields in Interface Definitions (VIP #2042)
- Support `@pure` decorator (VIP #2041)
- Overflow checks for exponentiation (#2072)
- Validate return data length via `RETURNDATASIZE` (#2076)
- Improved constant folding (#1949)
- Allow raise without reason string (VIP #1902)
- Make the type argument in `method_id` optional (VIP #1980)
- Hash complex types when used as indexed values in an event (#2060)
- Ease restrictions on calls to self (#2059)
- Remove ordering restrictions in module-scope of contract (#2057)
- `raw_call` can now be used to perform a `STATICCALL` (#1973)
- Optimize precompiles to use `STATICCALL` (#1930)

Some of the bug and stability fixes:

- Arg clamping issue when using multidimensional arrays (#2071)
- Support calldata arrays with the `in` comparator (#2070)
- Prevent modification of a storage array during iteration via `for` loop (#2028)
- Fix memory length of revert string (#1982)
- Memory offset issue when returning tuples from private functions (#1968)
- Issue with arrays as default function arguments (#2077)
- Private function calls no longer generate a call signature (#2058)

Significant codebase refactor, thanks to (@iamdefinitelyahuman)!

NOTE: `v0.2.0` was not used due to a conflict in PyPI with a previous release. Both tags `v0.2.0` and `v0.2.1` are identical.

18.7 v0.1.0-beta.17

Date released: 24-03-2020

The following VIPs and features were implemented for Beta 17:

- `raw_call` and `slice` argument updates (VIP #1879)
- NatSpec support (#1898)

Some of the bug and stability fixes:

- ABI interface fixes (#1842)
- Modifications to how ABI data types are represented (#1846)
- Generate method identifier for struct return type (#1843)
- Return tuple with fixed array fails to compile (#1838)
- Also lots of refactoring and doc updates!

This release will be the last to follow our current release process. All future releases will be governed by the versioning scheme (#1887). The next release will be v0.2.0, and contain many breaking changes.

18.8 v0.1.0-beta.16

Date released: 09-01-2020

Beta 16 was a quick patch release to fix one issue: (#1829)

18.9 v0.1.0-beta.15

Date released: 06-01-2020

NOTE: we changed our license to Apache 2.0 (#1772)

The following VIPs were implemented for Beta 15:

- EVM Ruleset Switch (VIP #1230)
- Add support for EIP-1344, Chain ID Opcode (VIP #1652)
- Support for EIP-1052, `EXTCODEHASH` (VIP #1765)

Some of the bug and stability fixes:

- Removed all traces of Javascript from the codebase (#1770)
- Ensured sufficient gas stipend for precompiled calls (#1771)
- Allow importing an interface that contains an `implements` statement (#1774)
- Fixed how certain values compared when using `min` and `max` (#1790)
- Removed unnecessary overflow checks on `addmod` and `mulmod` (#1786)
- Check for state modification when using tuples (#1785)
- Fix Windows path issue when importing interfaces (#1781)
- Added Vyper grammar, currently used for fuzzing (#1768)
- Modify modulus calculations for literals to be consistent with the EVM (#1792)
- Explicitly disallow the use of exponentiation on decimal values (#1792)
- Add compile-time checks for divide by zero and modulo by zero (#1792)
- Fixed some issues with negating constants (#1791)
- Allow relative imports beyond one parent level (#1784)
- Implement SHL/SHR for bitshifting, using Constantinople rules (#1796)
- `vyper-json` compatibility with `solc` settings (#1795)
- Simplify the type check when returning lists (#1797)
- Add branch coverage reporting (#1743)
- Fix struct assignment order (#1728)
- Added more words to reserved keyword list (#1741)
- Allow scientific notation for literals (#1721)

- Avoid overflow on sqrt of Decimal upper bound (#1679)
- Refactor ABI encoder (#1723)
- Changed opcode costs per EIP-1884 (#1764)

Special thanks to (@iamdefinitelyahuman) for lots of updates this release!

18.10 v0.1.0-beta.14

Date released: 13-11-2019

Some of the bug and stability fixes:

- Mucho Documentation and Example cleanup!
- Python 3.8 support (#1678)
- Disallow scientific notation in literals, which previously parsed incorrectly (#1681)
- Add implicit rewrite rule for `bytes[32]` -> `bytes32` (#1718)
- Support `bytes32` in `raw_log` (#1719)
- Fixed EOF parsing bug (#1720)
- Cleaned up arithmetic expressions (#1661)
- Fixed off-by-one in check for homogeneous list element types (#1673)
- Fixed stack valency issues in if and for statements (#1665)
- Prevent overflow when using `sqrt` on certain datatypes (#1679)
- Prevent shadowing of internal variables (#1601)
- Reject unary substraction on unsigned types (#1638)
- Disallow `orelse` syntax in `for` loops (#1633)
- Increased clarity and efficiency of zero-padding (#1605)

18.11 v0.1.0-beta.13

Date released: 27-09-2019

The following VIPs were implemented for Beta 13:

- Add `vyper-json` compilation mode (VIP #1520)
- Environment variables and constants can now be used as default parameters (VIP #1525)
- Require unitialized memory be set on creation (VIP #1493)

Some of the bug and stability fixes:

- Type check for default params and arrays (#1596)
- Fixed bug when using assertions inside for loops (#1619)
- Fixed zero padding error for ABI encoder (#1611)
- Check `calldatasize` before `calldataload` for function selector (#1606)

18.12 v0.1.0-beta.12

Date released: 27-08-2019

The following VIPs were implemented for Beta 12:

- Support for relative imports (VIP #1367)
- Restricted use of environment variables in private functions (VIP #1199)

Some of the bug and stability fixes:

- `@nonreentrant/@constant` logical inconsistency (#1544)
- Struct passthrough issue (#1551)
- Private underflow issue (#1470)
- Constancy check issue (#1480)
- Prevent use of conflicting method IDs (#1530)
- Missing arg check for private functions (#1579)
- Zero padding issue (#1563)
- `vyper.cli` rearchitecture of scripts (#1574)
- AST end offsets and Solidity-compatible compressed sourcemap (#1580)

Special thanks to (@iamdefinitelyahuman) for lots of updates this release!

18.13 v0.1.0-beta.11

Date released: 23-07-2019

Beta 11 brings some performance and stability fixes.

- Using calldata instead of memory parameters. (#1499)
- Reducing of contract size, for large parameter functions. (#1486)
- Improvements for Windows users (#1486) (#1488)
- Array copy optimisation (#1487)
- Fixing `@nonreentrant` decorator for return statements (#1532)
- `sha3` builtin function removed (#1328)
- Disallow conflicting method IDs (#1530)
- Additional `convert()` supported types (#1524) (#1500)
- Equality operator for strings and bytes (#1507)
- Change in `compile_codes` interface function (#1504)

Thanks to all the contributors!

18.14 v0.1.0-beta.10

Date released: 24-05-2019

- Lots of linting and refactoring!
- Bugfix with regards to using arrays as parameters to private functions (#1418). Please check your contracts, and upgrade to latest version, if you do use this.
- Slight shrinking in init produced bytecode. (#1399)
- Additional constancy protection in the `for .. range` expression. (#1397)
- Improved bug report (#1394)
- Fix returning of External Contract from functions (#1376)
- Interface unit fix (#1303)
- Not Equal (!=) optimisation (#1303) 1386
- New `assert <condition>, UNREACHABLE` statement. (#711)

Special thanks to (Charles Cooper), for some excellent contributions this release.

18.15 v0.1.0-beta.9

Date released: 12-03-2019

- Add support for list constants (#1211)
- Add `sha256` function (#1327)
- Renamed `create_with_code_of` to `create_forwarder_to` (#1177)
- `@nonreentrant` Decorator (#1204)
- Add opcodes and opcodes_runtime flags to compiler (#1255)
- Improved External contract call interfaces (#885)

18.16 Prior to v0.1.0-beta.9

Prior to this release, we managed our change log in a different fashion. Here is the old changelog:

- **2019.04.05**: Add stricter checking of unbalanced return statements. (#590)
- **2019.03.04**: `create_with_code_of` has been renamed to `create_forwarder_to`. (#1177)
- **2019.02.14**: Assigning a persistent contract address can only be done using the `bar_contact = ERC20(<address>)` syntax.
- **2019.02.12**: ERC20 interface has to be imported using `from vyper.interfaces import ERC20` to use.
- **2019.01.30**: Byte array literals need to be annoted using `b""`, strings are represented as "".
- **2018.12.12**: Disallow use of `None`, disallow use of `del`, implemented `clear()` built-in function.
- **2018.11.19**: Change mapping syntax to use `map()`. (VIP564)
- **2018.10.02**: Change the convert style to use types instead of string. (VIP1026)

- **2018.09.24**: Add support for custom constants.
- **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

CHAPTER 19

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.

19.1 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

19.2 How to Suggest Improvements

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

19.3 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.

19.4 Fix Bugs

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

19.5 Style Guide

Our *style guide* outlines best practices for the Vyper repository. Please ask us on Gitter if you have questions about anything that is not outlined in the style guide.

19.6 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.

19.6.1 Implementing New Features

If you are writing a new feature, please ensure you write appropriate Pytest 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!

CHAPTER 20

Style Guide

This document outlines the code style, project structure and practices followed by the Vyper development team.

Note: Portions of the current codebase do not adhere to this style guide. We are in the process of a large-scale refactor and this guide is intended to outline the structure and best practices *during and beyond* this refactor. Refactored code and added functionality **must** adhere to this guide. Bugfixes and modifications to existing functionality **may** adopt the same style as the related code.

20.1 Project Organization

- Each subdirectory within Vyper **should** be a self-contained package representing a single pass of the compiler or other logical component.
- Functionality intended to be called from modules outside of a package **must** be exposed within the base `__init__.py`. All other functionality is for internal use only.
- It **should** be possible to remove any package and replace it with another that exposes the same API, without breaking functionality in other packages.

20.2 Code Style

All code **must** conform to the PEP 8 style guide with the following exceptions:

- Maximum line length of 100

We handle code formatting with black with the line-length option set to 80. This ensures a consistent style across the project and saves time by not having to be opinionated.

20.2.1 Naming Conventions

Names **must** adhere to PEP 8 naming conventions:

- **Modules** have short, all-lowercase names. Underscores can be used in the module name if it improves readability.
- **Class names** use the CapWords convention.
- **Exceptions** follow the same conventions as other classes.
- **Function** names are lowercase, with words separated by underscores when it improves readability.
- **Method** names and **instance** variables follow the same conventions as functions.
- **Constants** use all capital letters with underscores separating words.

Leading Underscores

A single leading underscore marks an object as private.

- Classes and functions with one leading underscore are only used in the module where they are declared. They **must not** be imported.
- Class attributes and methods with one leading underscore **must** only be accessed by methods within the same class.

Booleans

- Boolean values **should** be prefixed with `is_`.
- Booleans **must not** represent *negative* properties, (e.g. `is_not_set`). This can result in double-negative evaluations which are not intuitive for readers.
- Methods that return a single boolean **should** use the `@property` decorator.

Methods

The following conventions **should** be used when naming functions or methods. Consistent naming provides logical consistency throughout the codebase and makes it easier for future readers to understand what a method does (and does not) do.

- `get_`: For simple data retrieval without any side effects.
- `fetch_`: For retreivals that may have some sort of side effect.
- `build_`: For creation of a new object that is derived from some other data.
- `set_`: For adding a new value or modifying an existing one within an object.
- `add_`: For adding a new attribute or other value to an object. Raises an exception if the value already exists.
- `replace_`: For mutating an object. Should return `None` on success or raise an exception if something is wrong.
- `compare_`: For comparing values. Returns `True` or `False`, does not raise an exception.
- `validate_`: Returns `None` or raises an exception if something is wrong.
- `from_`: For class methods that instantiate an object based on the given input data.

For other functionality, choose names that clearly communicate intent without being overly verbose. Focus on *what* the method does, not on *how* the method does it.

20.2.2 Imports

Import sequencing is handled with isort. We follow these additional rules:

Standard Library Imports

Standard libraries **should** be imported absolutely and without aliasing. Importing the library aids readability, as other users may be familiar with that library.

```
# Good
import os
os.stat('.')

# Bad
from os import stat
stat('.')
```

Internal Imports

Internal imports are those between two modules inside the same Vyper package.

- Internal imports **may** use either `import` or `from ..` syntax. The imported value **shoould** be a module, not an object. Importing modules instead of objects avoids circular dependency issues.
- Internal imports **may** be aliased where it aids readability.
- Internal imports **must** use absolute paths. Relative imports cause issues when the module is moved.

```
# Good
import vyper.ast.nodes as nodes
from vyper.ast import nodes

# Bad, `get_node` is a function
from vyper.ast.nodes import get_node

# Bad, do not use relative import paths
from . import nodes
```

Cross-Package Imports

Cross-package imports are imports between one Vyper package and another.

- Cross-package imports **must not** request anything beyond the root namespace of the target package.
- Cross-package imports **may** be aliased where it aids readability.
- Cross-package imports **may** use `from [module] import [package]` syntax.

```
# Good
from vyper.ast import fold
from vyper import ast as vy_ast
```

(continues on next page)

(continued from previous page)

```
# Bad, do not import beyond the root namespace
from vyper.ast.annotation import annotate_python_ast
```

20.2.3 Exceptions

We use *custom exception classes* to indicate what has gone wrong during compilation.

- All raised exceptions **must** use an exception class that appropriately describes what has gone wrong. When none fits, or when using a single exception class for an overly broad range of errors, consider creating a new class.
- Builtin Python exceptions **must not** be raised intentionally. An unhandled builtin exception indicates a bug in the codebase.
- Use `CompilerPanic` for errors that are not caused by the user.

20.2.4 Strings

Strings substitutions **should** be performed via formatted string literals rather than the `str.format` method or other techniques.

20.2.5 Type Annotations

- All publicly exposed classes and methods **should** include PEP 484 annotations for all arguments and return values.
- Type annotations **should** be included directly in the source. Stub files **may** be used where there is a valid reason. Source files using stubs **must** still be annotated to aid readability.
- Internal methods **should** include type annotations.

20.3 Tests

We use the pytest framework for testing, and *eth-tester* for our local development chain.

20.3.1 Best Practices

- `pytest` functionality **should not** be imported with `from ...` style syntax, particularly `pytest.raises`. Importing the library itself aids readability.
- Tests **must not** be interdependent. We use `xdist` to execute tests in parallel. You **cannot** rely on which order tests will execute in, or that two tests will execute in the same process.
- Test cases **should** be designed with a minimalistic approach. Each test should verify a single behavior. A good test is one with few assertions, and where it is immediately obvious exactly what is being tested.
- Where logical, tests **should** be parametrized or use property-based testing.
- Tests **must not** involve mocking.

20.3.2 Directory Structure

Where possible, the test suite **should** copy the structure of main Vyper package. For example, test cases for `vyper/context/types/` should exist at `tests/context/types/`.

20.3.3 Filenames

Test files **must** use the following naming conventions:

- `test_[module].py`: When all tests for a module are contained in a single file.
- `test_[module]_[functionality].py`: When tests for a module are split across multiple files.

20.3.4 Fixtures

- Fixtures **should** be stored in `conftest.py` rather than the test file itself.
- `conftest.py` files **must not** exist more than one subdirectory beyond the initial `tests/` directory.
- The functionality of a fixture **must** be fully documented, either via docstrings or comments.

20.4 Documentation

It is important to maintain comprehensive and up-to-date documentation for the Vyper language.

- Documentation **must** accurately reflect the current state of the master branch on Github.
- New functionality **must not** be added without corresponding documentation updates.

20.4.1 Writing Style

We use imperative, present tense to describe APIs: "return" not "returns". One way to test if we have it right is to complete the following sentence:

> "If we call this API it will: . . . "

For narrative style documentation, we prefer the use of first-person "we" form over second-person "you" form.

Additionally, we **recommend** the following best practices when writing documentation:

- Use terms consistently.
- Avoid ambiguous pronouns.
- Eliminate unneeded words.
- Establish key points at the start of a document.
- Focus each paragraph on a single topic.
- Focus each sentence on a single idea.
- Use a numbered list when order is important and a bulleted list when order is irrelevant.
- Introduce lists and tables appropriately.

Google's technical writing courses are a valuable resource. We recommend reviewing them before any significant documentation work.

20.4.2 API Directives

- All API documentation **must** use standard Python directives.
- Where possible, references to syntax **should** use appropriate Python roles.
- External references **may** use intersphinx roles.

20.4.3 Headers

- Each documentation section **must** begin with a label of the same name as the filename for that section. For example, this section's filename is `style-guide.rst`, so the RST opens with a label `_style-guide`.
- Section headers **should** use the following sequence, from top to bottom: `#`, `=`, `-`, `*`, `^`.

20.5 Internal Documentation

Internal documentation is vital to aid other contributors in understanding the layout of the Vyper codebase.

We handle internal documentation in the following ways:

- A `README.md` **must** be included in each first-level subdirectory of the Vyper package. The readme explain the purpose, organization and control flow of the subdirectory.
- All publicly exposed classes and methods **must** include detailed docstrings.
- Internal methods **should** include docstrings, or at minimum comments.
- Any code that may be considered "clever" or "magic" **must** include comments explaining exactly what is happening.

Docstrings **should** be formatted according to the NumPy docstring style.

20.6 Commit Messages

Contributors **should** adhere to the following standards and best practices when making commits to be merged into the Vyper codebase.

Maintainers **may** request a rebase, or choose to squash merge pull requests that do not follow these standards.

20.6.1 Conventional Commits

Commit messages **should** adhere to the Conventional Commits standard. A convetional commit message is structured as follows:

```
<type>[optional scope]: <description>

[optional body]

[optional footer]
```

The commit contains the following elements, to communicate intent to the consumers of your library:

- **fix**: a commit of the *type* `fix` patches a bug in your codebase (this correlates with `PATCH` in semantic versioning).

- **feat**: a commit of the *type* `feat` introduces a new feature to the codebase (this correlates with `MINOR` in semantic versioning).
- **BREAKING CHANGE**: a commit that has the text `BREAKING CHANGE:` at the beginning of its optional body or footer section introduces a breaking API change (correlating with `MAJOR` in semantic versioning). A BREAKING CHANGE can be part of commits of any *type*.

The use of commit types other than `fix:` and `feat:` is recommended. For example: `docs:`, `style:`, `refactor:`, `test:`, `chore:`, or `improvement:`. These tags are not mandated by the specification and have no implicit effect in semantic versioning.

20.6.2 Best Practices

We **recommend** the following best practices for commit messages (taken from How To Write a Commit Message):

- Limit the subject line to 50 characters.
- Use imperative, present tense in the subject line.
- Capitalize the subject line.
- Do not end the subject line with a period.
- Separate the subject from the body with a blank line.
- Wrap the body at 72 characters.
- Use the body to explain what and why vs. how.

CHAPTER 21

Vyper Versioning Guideline

21.1 Motivation

Vyper has different groups that are considered "users":

- Smart Contract Developers (Developers)
- Package Integrators (Integrators)
- Security Professionals (Auditors)

Each set of users must understand which changes to the compiler may require their attention, and how these changes may impact their use of the compiler. This guide defines what scope each compiler change may have, it's potential impact based on the type of user, so that users can stay informed about the progress of Vyper.

Group	How they use Vyper
Developers	Write smart contracts in Vyper
Integrators	Integerating Vyper package or CLI into tools
Auditors	Aware of Vyper features and security issues

A big part of Vyper's "public API" is the language grammar. The syntax of the language is the main touchpoint all parties have with Vyper, so it's important to discuss changes to the language from the viewpoint of dependability. Users expect that all contracts written in an earlier version of Vyper will work seemlessly with later versions, or that they will be reasonably informed when this isn't possible. The Vyper package itself and it's CLI utilities also has a fairly well-defined public API, which consists of the available features in Vyper's exported package, the top level modules under the package, and all CLI scripts.

21.2 Version Types

This guide was adapted from semantic versioning. It defines a format for version numbers that looks like `MAJOR.MINOR.PATCH[-STAGE.DEVNUM]`. We will periodically release updates according to this format, with the release decided via the following guidelines.

21.2.1 Major Release `X.0.0`

Changes to the grammar cannot be made in a backwards incompatible way without changing Major versions (e.g. `v1.x` -> `v2.x`). It is to be expected that breaking changes to many features will occur when updating to a new Major release, primarily for Developers that use Vyper to compile their contracts. Major releases will have an audit performed prior to release (e.g. `x.0.0` releases) and all `moderate` or `severe` vulnerabilities will be addressed that are reported in the audit report. `minor` or `informational` vulnerabilities *should* be addressed as well, although this may be left up to the maintainers of Vyper to decide.

Group	Look For
Developers	Syntax deprecation, new features
Integrators	No changes
Auditors	Audit report w/ resolved changes

21.2.2 Minor Release `x.Y.0`

Minor version updates may add new features or fix a `moderate` or `severe` vulnerability, and these will be detailed in the Release Notes for that release. Minor releases may change the features or functionality offered by the package and CLI scripts in a backwards-incompatible way that requires attention from an integrator. Minor releases are required to fix a `moderate` or `severe` vulnerability, but a `minor` or `informational` vulnerability can be fixed in Patch releases, alongside documentation updates.

Group	Look For
Developers	New features, security bug fixes
Integrators	Changes to external API
Auditors	`moderate` or `severe` patches

21.2.3 Patch Release `x.y.Z`

Patch version releases will be released to fix documentation issues, usage bugs, and `minor` or `informational` vulnerabilities found in Vyper. Patch releases should only update error messages and documentation issues relating to it's external API.

Group	Look For
Developers	Doc updates, usage bug fixes, error messages
Integrators	Doc updates, usage bug fixes, error messages
Auditors	`minor` or `informational` patches

21.2.4 Vyper Security

As Vyper develops, it is very likely that we will encounter inconsistencies in how certain language features can be used, and software bugs in the code the compiler generates. Some of them may be quite serious, and can render a user's compiled contract vulnerable to exploitation for financial gain. As we become aware of these vunlerabilities, we will work according to our security policy to resolve these issues, and eventually will publish the details of all reported vulnerabilities here. Fixes for these issues will also be noted in the *Release Notes*.

21.2.5 Vyper *Next*

There may be multiple Major versions in process of development. Work on new features that break compatibility with the existing grammar can be maintained on a separate branch called `next` and represents the next Major release of Vyper (provided in an unaudited state without Release Notes). The work on the current branch will remain on the `master` branch with periodic new releases using the process as mentioned above.

Any other branches of work outside of what is being tracked via `master` will use the `-alpha.[release #]` (Alpha) to denote WIP updates, and `-beta.[release #]` (Beta) to describe work that is eventually intended for release. `-rc.[release #]` (Release Candidate) will only be used to denote candidate builds prior to a Major release. An audit will be solicited for `-rc.1` builds, and subsequent releases *may* incorporate feedback during the audit. The last Release Candidate will become the next Major release, and will be made available alongside the full audit report summarizing the findings.

21.3 Pull Requests

Pull Requests can be opened against either `master` or `next` branch, depending on their content. Changes that would increment a Minor or Patch release should target `master`, whereas changes to syntax (as detailed above) should be opened against `next`. The `next` branch will be periodically rebased against the `master` branch to pull in changes made that were added to the latest supported version of Vyper.

21.4 Communication

Major and Minor versions should be communicated on appropriate communications channels to end users, and Patch updates will usually not be discussed, unless there is a relevant reason to do so.

Index