Vyper Documentation

Vyper Team (originally created by Vitalik Buterin)

Oct 04, 2023

CONTENTS

CHAPTER

ONE

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

TWO

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. (Activate the vyper-remix plugin in the Plugin manager.)

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 parameters 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==0.3.7
```

You can check if Vyper is installed completely or not by typing the following in your terminal/cmd:

```
vyper --version
```

2.3 nix

View the versions supported through nix at nix package search

Note: The derivation for Vyper is located at nixpkgs

2.3.1 Installing Vyper

```
nix-env -iA nixpkgs.vyper
```

CHAPTER

THREE

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 `_auction_start` and
# `_bidding_time` seconds bidding time on behalf of the
# beneficiary address `_beneficiary`.
@external
def __init__(_beneficiary: address, _auction_start: uint256, _bidding_time: uint256):
    self.beneficiary = _beneficiary
    self.auctionStart = _auction_start  # auction start time can be in the past, present␣
↪or future
    self.auctionEnd = self.auctionStart + _bidding_time
    assert block.timestamp < self.auctionEnd # auction end time should be in the future

# Bid on the auction with the value sent
# together with this transaction.
```

(continues on next page)

(continued from previous page)

```
# The value will only be refunded if the
# auction is not won.
@external
@payable
def bid():
    # Check if bidding period has started.
    assert block.timestamp >= self.auctionStart
    # 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, _auction_start: uint256, _bidding_time: uint256):
    self.beneficiary = _beneficiary
    self.auctionStart = _auction_start  # auction start time can be in the past, present␣
↪or future
    self.auctionEnd = self.auctionStart + _bidding_time
    assert block.timestamp < self.auctionEnd # auction end time should be in the future
```

The contract is initialized with three arguments: `_beneficiary` of type `address`, `_auction_start` with type `uint256` and `_bidding_time` with type `uint256`, the time difference between the start and end of the auction. We then store these three pieces of information into the contract variables `self.beneficiary`, `self.auctionStart` and `self.auctionEnd` respectively. 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():
    # Check if bidding period has started.
    assert block.timestamp >= self.auctionStart
    # 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`.

Here, we first check whether the current time is within the bidding period by comparing with the auction's start and end times 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 three `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

```

(continues on next page)

(continued from previous page)

```
    # 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)
```

(continues on next page)

(continued from previous page)

```
23
24  # Final auction state
25  highestBid: public(uint256)
26  highestBidder: public(address)
27
28  # State of the bids
29  bids: HashMap[address, Bid[128]]
30  bidCounts: HashMap[address, int128]
31
32  # Allowed withdrawals of previous bids
33  pendingReturns: HashMap[address, uint256]
34
35
36  # Create a blinded auction with `_biddingTime` seconds bidding time and
37  # `_revealTime` seconds reveal time on behalf of the beneficiary address
38  # `_beneficiary`.
39  @external
40  def __init__(_beneficiary: address, _biddingTime: uint256, _revealTime: uint256):
41      self.beneficiary = _beneficiary
42      self.biddingEnd = block.timestamp + _biddingTime
43      self.revealEnd = self.biddingEnd + _revealTime
44
45
46  # Place a blinded bid with:
47  #
48  # _blindedBid = keccak256(concat(
49  #       convert(value, bytes32),
50  #       convert(fake, bytes32),
51  #       secret)
52  # )
53  #
54  # The sent ether is only refunded if the bid is correctly revealed in the
55  # revealing phase. The bid is valid if the ether sent together with the bid is
56  # at least "value" and "fake" is not true. Setting "fake" to true and sending
57  # not the exact amount are ways to hide the real bid but still make the
58  # required deposit. The same address can place multiple bids.
59  @external
60  @payable
61  def bid(_blindedBid: bytes32):
62      # Check if bidding period is still open
63      assert block.timestamp < self.biddingEnd
64
65      # Check that payer hasn't already placed maximum number of bids
66      numBids: int128 = self.bidCounts[msg.sender]
67      assert numBids < MAX_BIDS
68
69      # Add bid to mapping of all bids
70      self.bids[msg.sender][numBids] = Bid({
71          blindedBid: _blindedBid,
72          deposit: msg.value
73          })
74      self.bidCounts[msg.sender] += 1
```

(continues on next page)

(continued from previous page)

```
75
76
77  # Returns a boolean value, `True` if bid placed successfully, `False` otherwise.
78  @internal
79  def placeBid(bidder: address, _value: uint256) -> bool:
80      # If bid is less than highest bid, bid fails
81      if (_value <= self.highestBid):
82          return False
83
84      # Refund the previously highest bidder
85      if (self.highestBidder != empty(address)):
86          self.pendingReturns[self.highestBidder] += self.highestBid
87
88      # Place bid successfully and update auction state
89      self.highestBid = _value
90      self.highestBidder = bidder
91
92      return True
93
94
95  # Reveal your blinded bids. You will get a refund for all correctly blinded
96  # invalid bids and for all bids except for the totally highest.
97  @external
98  def reveal(_numBids: int128, _values: uint256[128], _fakes: bool[128], _secrets:␣
    ↪bytes32[128]):
99      # Check that bidding period is over
100     assert block.timestamp > self.biddingEnd
101
102     # Check that reveal end has not passed
103     assert block.timestamp < self.revealEnd
104
105     # Check that number of bids being revealed matches log for sender
106     assert _numBids == self.bidCounts[msg.sender]
107
108     # Calculate refund for sender
109     refund: uint256 = 0
110     for i in range(MAX_BIDS):
111         # Note that loop may break sooner than 128 iterations if i >= _numBids
112         if (i >= _numBids):
113             break
114
115         # Get bid to check
116         bidToCheck: Bid = (self.bids[msg.sender])[i]
117
118         # Check against encoded packet
119         value: uint256 = _values[i]
120         fake: bool = _fakes[i]
121         secret: bytes32 = _secrets[i]
122         blindedBid: bytes32 = keccak256(concat(
123             convert(value, bytes32),
124             convert(fake, bytes32),
125             secret
```

(continues on next page)

(continued from previous page)

```
        ))

        # Bid was not actually revealed
        # Do not refund deposit
        assert blindedBid == bidToCheck.blindedBid

        # 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
    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
```

(continues on next page)

(continued from previous page)

```
    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
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
```

(continues on next page)

(continued from previous page)

```
    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.
```

(continues on next page)

(continued from previous page)

```
    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)

funders: HashMap[address, uint256]
beneficiary: address
deadline: public(uint256)
goal: public(uint256)
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)"

    self.funders[msg.sender] += msg.value

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

    selfdestruct(self.beneficiary)
```

(continues on next page)

(continued from previous page)

```
32
33  # Let participants withdraw their fund
34  @external
35  def refund():
36      assert block.timestamp >= self.deadline and self.balance < self.goal
37      assert self.funders[msg.sender] > 0
38
39      value: uint256 = self.funders[msg.sender]
40      self.funders[msg.sender] = 0
41
42      send(msg.sender, value)
```

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

```
3   funders: HashMap[address, uint256]
4   beneficiary: address
5   deadline: public(uint256)
6   goal: public(uint256)
7   timelimit: public(uint256)
8
9   # Setup global variables
10  @external
11  def __init__(_beneficiary: address, _goal: uint256, _timelimit: uint256):
12      self.beneficiary = _beneficiary
13      self.deadline = block.timestamp + _timelimit
```

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 an address, and the value is a number representing the contribution of each participant. 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.

```
9   # Setup global variables
10  @external
11  def __init__(_beneficiary: address, _goal: uint256, _timelimit: uint256):
12      self.beneficiary = _beneficiary
13      self.deadline = block.timestamp + _timelimit
14      self.timelimit = _timelimit
15      self.goal = _goal
```

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

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

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

    self.funders[msg.sender] += msg.value
```

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

```
# Enough money was raised! Send funds to the beneficiary
@external
def finalize():
    assert block.timestamp >= self.deadline, "deadline has passed"
    assert self.balance >= self.goal, "the goal has not been reached"

    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.

```
# Let participants withdraw their fund
@external
def refund():
    assert block.timestamp >= self.deadline and self.balance < self.goal
    assert self.funders[msg.sender] > 0

    value: uint256 = self.funders[msg.sender]
    self.funders[msg.sender] = 0

    send(msg.sender, value)
```

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 let users get their funds back using the withdraw pattern.

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:
    # 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 != empty(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 == empty(address))


@view
```

(continues on next page)

(continued from previous page)

```
@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
        })
        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:
```

(continues on next page)

(continued from previous page)

```
            # 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)

# 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 != empty(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
```

(continues on next page)

(continued from previous page)

```
    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
@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)
```

(continues on next page)

(continued from previous page)

```
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
```

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 != empty(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.
```

(continues on next page)

(continued from previous page)

```
@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:
```

(continues on next page)

(continued from previous page)

```
    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

# 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
```

(continues on next page)

(continued from previous page)

```
    self.holdings[msg.sender] += buy_order

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

# 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)

    # 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.
```

(continues on next page)

(continued from previous page)

```
@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)

# 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()

# 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

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

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

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 `uint256`, 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.

```
# 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
```

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.

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

(continues on next page)

(continued from previous page)

```
    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.

```
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?

```
# 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.

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

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

```
# Give stock back to the company and get money back as ETH.
@external
def sellStock(sell_order: uint256):
```

(continues on next page)

(continued from previous page)

```
    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.

```
# 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)
```

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.

```
# 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)

```

(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.

```
# 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()`.

```
# 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()
```

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

FOUR

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 Pragmas

Vyper supports several source code directives to control compiler modes and help with build reproducibility.

4.1.1 Version Pragma

The version pragma ensures that a contract is only compiled by the intended compiler version, or range of versions. Version strings use NPM style syntax. Starting from v0.4.0 and up, version strings will use *PEP440 version specifiers <https://peps.python.org/pep-0440/#version-specifiers>_*.

As of 0.3.10, the recommended way to specify the version pragma is as follows:

```
#pragma version ^0.3.0
```

The following declaration is equivalent, and, prior to 0.3.10, was the only supported method to specify the compiler version:

```
# @version ^0.3.0
```

In the above examples, the contract will only compile with Vyper versions `0.3.x`.

4.1.2 Optimization Mode

The optimization mode can be one of `"none"`, `"codesize"`, or `"gas"` (default). For example, adding the following line to a contract will cause it to try to optimize for codesize:

```
#pragma optimize codesize
```

The optimization mode can also be set as a compiler option, which is documented in *Compiler Optimization Modes*. If the compiler option conflicts with the source code pragma, an exception will be raised and compilation will not continue.

4.1.3 EVM Version

The EVM version can be set with the `evm-version` pragma, which is documented in *Setting the Target EVM Version*.

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 separate 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 a custom defined type that allows you to group several variables together:

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

See the *Structs* documentation for more information.

CHAPTER

FIVE

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
`not x`	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 (N bit)

Keyword: `intN` (e.g., `int128`)

A signed integer which can store positive and negative integers. `N` must be a multiple of 8 between 8 and 256 (inclusive).

Values

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

Integer 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 both be of the same type.

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 both be of the same type.

Bitwise Operators

Operator	Description
`x & y`	Bitwise and
`x | y`	Bitwise or
`x ^ y`	Bitwise xor

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

Shifts

Operator	Description
`x << y`	Left shift
`x >> y`	Right shift

Shifting is only available for 256-bit wide types. That is, `x` must be `int256`, and `y` can be any unsigned integer. The right shift for `int256` compiles to a signed right shift (EVM `SAR` instruction).

Note: While at runtime shifts are unchecked (that is, they can be for any number of bits), to prevent common mistakes, the compiler is stricter at compile-time and will prevent out of bounds shifts. For instance, at runtime, `1 << 257` will evaluate to `0`, while that expression at compile-time will raise an `OverflowException`.

5.1.3 Unsigned Integer (N bit)

Keyword: `uintN` (e.g., `uint8`)

A unsigned integer which can store positive integers. `N` must be a multiple of 8 between 8 and 256 (inclusive).

Values

Integer values between 0 and (2^N-1).

Integer 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 `int256` by default. In cases where `uint8` is more appropriate, such as assignment, the literal might be interpreted as `uint8`. Example: `_variable: uint8 = _literal`. In order to explicitly cast a literal to a `uint8` use `convert(_literal, uint8)`.

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 same type.

Arithmetic Operators

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

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

Bitwise Operators

Operator	Description
`x & y`	Bitwise and
`x | y`	Bitwise or
`x ^ y`	Bitwise xor
`~x`	Bitwise not

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

Note: The Bitwise `not` operator is currently only available for `uint256` type.

Shifts

Operator	Description
`x << y`	Left shift
`x >> y`	Right shift

Shifting is only available for 256-bit wide types. That is, `x` must be `uint256`, and `y` can be any unsigned integer. The right shift for `uint256` compiles to a signed right shift (EVM `SHR` instruction).

Note: While at runtime shifts are unchecked (that is, they can be for any number of bits), to prevent common mistakes, the compiler is stricter at compile-time and will prevent out of bounds shifts. For instance, at runtime, `1 << 257` will evaluate to `0`, while that expression at compile-time will raise an `OverflowException`.

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 -18707220957835557353007165858768422651595.9365500928 (-2^{167} / 10^{10}) and 18707220957835557353007165858768422651595.9365500927 ((2^{167} - 1) / 10^{10}).

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

The ABI type (for computing method identifiers) of `decimal` is `fixed168x10`.

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 at an address, in bytes
`is_contract`	`bool`	Boolean indicating if a contract is deployed at an address
`code`	`Bytes`	Contract bytecode

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.

Note: `_address.code` requires the usage of *`slice`* to explicitly extract a section of contract bytecode. If the extracted section exceeds the bounds of bytecode, this will throw. You can check the size of `_address.code` using `_address.codesize`.

5.1.6 M-byte-wide Fixed Size Byte Array

Keyword: `bytesM` This is an M-byte-wide byte array that is otherwise similar to dynamically sized byte arrays. On an ABI level, it is annotated as bytesM (e.g., bytes32).

Example:

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

some_method_id: bytes4 = 0x01abcdef
```

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 max 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.

```
bytes_string: Bytes[100] = b"\x01"
```

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"
```

5.1.9 Enums

Keyword: `enum`

Enums are custom defined types. An enum must have at least one member, and can hold up to a maximum of 256 members. The members are represented by `uint256` values in the form of 2^n where `n` is the index of the member in the range `0 <= n <= 255`.

```
# Defining an enum with two members
enum Roles:
    ADMIN
    USER

# Declaring an enum variable
role: Roles = Roles.ADMIN

# Returning a member
return Roles.ADMIN
```

Operators

Comparisons

Comparisons return a boolean value.

Operator	Description
`x == y`	Equals
`x != y`	Does not equal
`x in y`	x is in y
`x not in y`	x is not in y

Bitwise Operators

Operator	Description
`x & y`	Bitwise and
`x | y`	Bitwise or
`x ^ y`	Bitwise xor
`~x`	Bitwise not

Enum members can be combined using the above bitwise operators. While enum members have values that are power of two, enum member combinations may not.

The `in` and `not in` operators can be used in conjunction with enum member combinations to check for membership.

```
enum Roles:
    MANAGER
    ADMIN
    USER

# Check for membership
```

(continues on next page)

(continued from previous page)

```
@external
def foo(a: Roles) -> bool:
    return a in (Roles.MANAGER | Roles.USER)

# Check not in
@external
def bar(a: Roles) -> bool:
    return a not in (Roles.MANAGER | Roles.USER)
```

Note that `in` is not the same as strict equality (==). `in` checks that *any* of the flags on two enum objects are simultaneously set, while == checks that two enum objects are bit-for-bit equal.

The following code uses bitwise operations to add and revoke permissions from a given `Roles` object.

5.2 Reference Types

Reference types are those whose components can be assigned to in-place without copying. For instance, array and struct members can be individually assigned to without overwriting the whole data structure.

Note: In terms of the calling convention, reference types are passed by value, not by reference. That means that, a calling function does not need to worry about a callee modifying the data of a passed structure.

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]`, except `Bytes[N]`, `String[N]` and enums.

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

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

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

Multidimensional lists are also possible. The notation for the declaration is reversed compared to some other languages, but the access notation is not reversed.

A two dimensional list can be declared with `_name: _ValueType[inner_size][outer_size]`. Elements can be accessed with `_name[outer_index][inner_index]`.

```
# Defining a list with 2 rows and 5 columns and set all values to 0
exampleList2D: int128[5][2] = empty(int128[5][2])

# Setting a value for row the first row (0) and last column (4)
exampleList2D[0][4] = 42
```

(continues on next page)

(continued from previous page)

```
# Setting values
exampleList2D = [[10, 11, 12, 13, 14], [16, 17, 18, 19, 20]]

# Returning the value in row 0 column 4 (in this case 14)
return exampleList2D[0][4]
```

Note: Defining an array in storage whose size is significantly larger than `2**64` can result in security vulnerabilities due to risk of overflow.

5.2.2 Dynamic Arrays

Dynamic arrays represent bounded arrays whose length can be modified at runtime, up to a bound specified in the type. They can be declared with `_name: DynArray[_Type, _Integer]`, where `_Type` can be of value type or reference type (except mappings).

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

# Setting values
exampleList = []
# exampleList.pop()  # would revert!
exampleList.append(42)  # exampleList now has length 1
exampleList.append(120)  # exampleList now has length 2
exampleList.append(356)  # exampleList now has length 3
# exampleList.append(1)  # would revert!

myValue: int128 = exampleList.pop()  # myValue == 356, exampleList now has length 2

# myValue = exampleList[2]  # would revert!

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

Note: Attempting to access data past the runtime length of an array, `pop()` an empty array or `append()` to a full array will result in a runtime `REVERT`. Attempting to pass an array in calldata which is larger than the array bound will result in a runtime `REVERT`.

Note: To keep code easy to reason about, modifying an array while using it as an iterator is disallowed by the language. For instance, the following usage is not allowed:

```
for item in self.my_array:
    self.my_array[0] = item
```

In the ABI, they are represented as `_Type[]`. For instance, `DynArray[int128, 3]` gets represented as `int128[]`, and `DynArray[DynArray[int128, 3], 3]` gets represented as `int128[][]`.

Note: Defining a dynamic array in storage whose size is significantly larger than `2**64` can result in security vulnerabilities due to risk of overflow.

5.2.3 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`.

```
# Defining a struct
struct MyStruct:
    value1: int128
    value2: decimal

# Declaring a struct variable
exampleStruct: MyStruct = MyStruct({value1: 1, value2: 2.0})

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

5.2.4 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 is 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, arrays or structs are not supported 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 its given type.

To reset a variable to its 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`
`uint8`	`0`
`int128`	`0`
`int256`	`0`
`uint256`	`0`

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. Type conversions in Vyper are designed to be safe and intuitive. All type conversions will check that the input is in bounds for the output type. The general principles are:

- Except for conversions involving decimals and bools, the input is bit-for-bit preserved.
- Conversions to bool map all nonzero inputs to 1.
- When converting from decimals to integers, the input is truncated towards zero.
- `address` types are treated as `uint160`, except conversions with signed integers and decimals are not allowed.
- Converting between right-padded (`bytes`, `Bytes`, `String`) and left-padded types, results in a rotation to convert the padding. For instance, converting from `bytes20` to `address` would result in rotating the input by 12 bytes to the right.
- Converting between signed and unsigned integers reverts if the input is negative.
- Narrowing conversions (e.g., `int256 -> int128`) check that the input is in bounds for the output type.
- Converting between bytes and int types results in sign-extension if the output type is signed. For instance, converting `0xff` (`bytes1`) to `int8` returns `-1`.

- Converting between bytes and int types which have different sizes follows the rule of going through the closest integer type, first. For instance, `bytes1 -> int16` is like `bytes1 -> int8 -> int16` (signextend, then widen). `uint8 -> bytes20` is like `uint8 -> uint160 -> bytes20` (rotate left 12 bytes).
- Enums can be converted to and from `uint256` only.

A small Python reference implementation is maintained as part of Vyper's test suite, it can be found here. The motivation and more detailed discussion of the rules can be found here.

CHAPTER

SIX

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.prevrandao`	`uint256`	Current randomness beacon provided by the beacon chain
`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.data`	`Bytes`	Message data
`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)
`tx.gasprice`	`uint256`	Gas price of current transaction in wei

Note: `block.prevrandao` is an alias for `block.difficulty`. Since `block.difficulty` is considered deprecated according to EIP-4399 after "The Merge" (Paris hard fork), we recommend using `block.prevrandao`.

Note: `msg.data` requires the usage of *`slice`* to explicitly extract a section of calldata. If the extracted section exceeds the bounds of calldata, this will throw. You can check the size of `msg.data` using *`len`*.

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 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

SEVEN

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
```

If a function has no return type, it is allowed to omit the `return` statement, otherwise, the function 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. If it is provided, it is limited to 1024 bytes.

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. If it is provided, it is limited to 1024 bytes.

This method's behavior is equivalent to:

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

CHAPTER

EIGHT

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

@external
def add_seven_with_overloading(a: uint256, b: uint256 = 3):
    return a + b
```

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

Note: For external functions with default arguments like `def my_function(x: uint256, b: uint256 = 1)` the Vyper compiler will generate `N+1` overloaded function selectors based on `N` default arguments.

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, two: uint256 = 2) -> uint256:
    return amount * two

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

Note: Since calling an `internal` function is realized by jumping to its entry label, the internal function dispatcher ensures the correctness of the jumps. Please note that for `internal` functions which use more than one default parameter, Vyper versions >=`0.3.8` are strongly recommended due to the security advisory GHSA-ph9x-4vc9-m39g.

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.

Functions marked with `@view` cannot call mutable (`payable` or `nonpayable`) functions. Any external calls are made using the special `STATICCALL` opcode, which prevents state changes at the EVM level.

Functions marked with `@pure` cannot call non-`pure` functions.

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
    ...
```

You can put the `@nonreentrant(<key>)` decorator on a `__default__` function but we recommend against it because in most circumstances it will not work in a meaningful way.

Nonreentrancy locks work by setting a specially allocated storage slot to a `<locked>` value on function entrance, and setting it to an `<unlocked>` value on function exit. On function entrance, if the storage slot is detected to be the `<locked>` value, execution reverts.

You cannot put the `@nonreentrant` decorator on a `pure` function. You can put it on a `view` function, but it only checks that the function is not in a callback (the storage slot is not in the `<locked>` state), as `view` functions can only read the state, not change it.

Note: A mutable function can protect a `view` function from being called back into (which is useful for instance, if a `view` function would return inconsistent state during a mutable function), but a `view` function cannot protect itself from being called back into. Note that mutable functions can never be called from a `view` function because all external calls out from a `view` function are protected by the use of the `STATICCALL` opcode.

Note: A nonreentrant lock has an `<unlocked>` value of 3, and a `<locked>` value of 2. Nonzero values are used to take advantage of net gas metering - as of the Berlin hard fork, the net cost for utilizing a nonreentrant lock is 2300 gas. Prior to v0.3.4, the `<unlocked>` and `<locked>` values were 0 and 1, respectively.

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.

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 - it is a design decision of Ethereum to make no differentiation between sending ether to a contract or a user address.

```
event Payment:
    amount: uint256
    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

@external
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, a dynamic array, or generated from the built-in `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(stop, bound=N):
    ...
```

Here, `stop` can be a variable with integer type, greater than zero. `N` must be a compile-time constant. `i` begins as zero and increments by one until it is equal to `stop`. If `stop` is larger than `N`, execution will revert at runtime. In certain cases, you may not have a guarantee that `stop` is less than `N`, but still want to avoid the possibility of runtime reversion. To accomplish this, use the `bound=` keyword in combination with `min(stop, N)` as the argument to `range`, like `range(min(stop, N), bound=N)`. This is helpful for use cases like chunking up operations on larger arrays across multiple transactions.

Another use of range can be with `START` and `STOP` bounds.

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

Here, `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`. If `a + N` would overflow, execution will revert.

CHAPTER

NINE

SCOPING AND DECLARATIONS

9.1 Variable Declaration

The first time a variable is referenced you must declare its *type*:

```
data: int128
```

In the above example, we declare the 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, 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 specify which element to return, for example `data(0)`.

9.1.2 Declaring Immutable Variables

Variables can be marked as `immutable` during declaration:

```
DATA: immutable(uint256)

@external
def __init__(_data: uint256):
    DATA = _data
```

Variables declared as immutable are similar to constants, except they are assigned a value in the constructor of the contract. Immutable values must be assigned a value at construction and cannot be assigned a value after construction.

The contract creation code generated by the compiler will modify the contract's runtime code before it is returned by appending all values assigned to immutables to the runtime code returned by the constructor. This is important if you are comparing the runtime code generated by the compiler with the one actually stored in the blockchain.

9.1.3 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 Storage Layout

Storage variables are located within a smart contract at specific storage slots. By default, the compiler allocates the first variable to be stored at `slot 0`; subsequent variables are stored in order after that.

There are cases where it is necessary to override this pattern and to allocate storage variables in custom slots. This behaviour is often required for upgradeable contracts, to ensure that both contracts (the old contract, and the new contract) store the same variable within the same slot.

This can be performed when compiling via `vyper` by including the `--storage-layout-file` flag.

For example, consider upgrading the following contract:

```
# old_contract.vy
owner: public(address)
balanceOf: public(HashMap[address, uint256])
```

```
# new_contract.vy
owner: public(address)
minter: public(address)
balanceOf: public(HashMap[address, uint256])
```

This would cause an issue when upgrading, as the `balanceOf` mapping would be located at `slot1` in the old contract, and `slot2` in the new contract.

This issue can be avoided by allocating `balanceOf` to `slot1` using the storage layout overrides. The contract can be compiled with `vyper new_contract.vy --storage-layout-file new_contract_storage.json` where `new_contract_storage.json` contains the following:

```
{
    "owner": {"type": "address", "slot": 0},
    "minter": {"type": "address", "slot": 2},
    "balanceOf": {"type": "HashMap[address, uint256]", "slot": 1}
}
```

For further information on generating the storage layout, see *Storage Layout*.

9.3 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.

9.3.1 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 an immutable or constant value. The following examples will not compile:

```
a: constant(bool) = True

@external
def foo() -> bool:
    # memory variable cannot have the same name as a constant or immutable variable
```

(continues on next page)

(continued from previous page)

```
    a: bool = False
    return a
```

```
a: immutable(bool)

@external
def __init__():
    a = True
@external
def foo(a:bool) -> bool:
    # input argument cannot have the same name as a constant or immutable variable
    return a
```

9.3.2 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
```

9.3.3 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 exiting 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

TEN

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 is 0.

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

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

Note: This function has been deprecated from version 0.3.4 onwards. Please use the & operator instead.

bitwise_not(*x: uint256*) → uint256

Return the bitwise 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
```

Note: This function has been deprecated from version 0.3.4 onwards. Please use the ~ operator instead.

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 is 1.

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

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

Note: This function has been deprecated from version 0.3.4 onwards. Please use the `|` operator instead.

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 is 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
```

Note: This function has been deprecated from version 0.3.4 onwards. Please use the `^` operator instead.

shift(*x: int256 | uint256*, *_shift: integer*) → 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
```

Note: This function has been deprecated from version 0.3.8 onwards. Please use the << and >> operators instead.

10.2 Chain Interaction

Vyper has three built-ins for contract creation; all three contract creation built-ins rely on the code to deploy already being stored on-chain, but differ in call vs deploy overhead, and whether or not they invoke the constructor of the contract to be deployed. The following list provides a short summary of the differences between them.

- **`create_minimal_proxy_to(target: address, ...)`**
  - Creates an immutable proxy to `target`
  - Expensive to call (incurs a single `DELEGATECALL` overhead on every invocation), cheap to create (since it only deploys `EIP-1167` forwarder bytecode)
  - Does not have the ability to call a constructor
  - Does **not** check that there is code at `target` (allows one to deploy proxies counterfactually)
- **`create_copy_of(target: address, ...)`**
  - Creates a byte-for-byte copy of runtime code stored at `target`
  - Cheap to call (no `DELEGATECALL` overhead), expensive to create (200 gas per deployed byte)
  - Does not have the ability to call a constructor
  - Performs an `EXTCODESIZE` check to check there is code at `target`
- **`create_from_blueprint(target: address, ...)`**
  - Deploys a contract using the initcode stored at `target`
  - Cheap to call (no `DELEGATECALL` overhead), expensive to create (200 gas per deployed byte)
  - Invokes constructor, requires a special "blueprint" contract to be deployed
  - Performs an `EXTCODESIZE` check to check there is code at `target`

`create_minimal_proxy_to`(*target: address*, *value: uint256 = 0*[, *salt: bytes32*]) → address

Deploys a small, EIP1167-compliant "minimal proxy contract" that duplicates the logic of the contract at `target`, but has its own state since every call to `target` is made using `DELEGATECALL` to `target`. To the end user, this should be indistinguishable from an independently deployed contract with the same code as `target`.

- `target`: Address of the contract to proxy to
- `value`: The wei value to send to the new contract address (Optional, default 0)
- `salt`: A `bytes32` value utilized by the deterministic `CREATE2` opcode (Optional, if not supplied, `CREATE` is used)

Returns the address of the newly created proxy contract. If the create operation fails (for instance, in the case of a `CREATE2` collision), execution will revert.

```
@external
def foo(target: address) -> address:
    return create_minimal_proxy_to(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 or have upgradeable code as this will affect the operation of the proxy contract.

Note: There is no runtime check that there is code already deployed at `target` (since a proxy may be deployed counterfactually). Most applications may want to insert this check.

Note: Before version 0.3.4, this function was named `create_forwarder_to`.

create_copy_of(*target: address*, *value: uint256 = 0*[, *salt: bytes32*]) → address

Create a physical copy of the runtime code at `target`. The code at `target` is byte-for-byte copied into a newly deployed contract.

- `target`: Address of the contract to copy
- `value`: The wei value to send to the new contract address (Optional, default 0)
- `salt`: A `bytes32` value utilized by the deterministic `CREATE2` opcode (Optional, if not supplied, `CREATE` is used)

Returns the address of the created contract. If the create operation fails (for instance, in the case of a `CREATE2` collision), execution will revert. If there is no code at `target`, execution will revert.

```
@external
def foo(target: address) -> address:
    return create_copy_of(target)
```

Note: The implementation of `create_copy_of` assumes that the code at `target` is smaller than 16MB. While this is much larger than the EIP-170 constraint of 24KB, it is a conservative size limit intended to future-proof deployer contracts in case the EIP-170 constraint is lifted. If the code at `target` is larger than 16MB, the behavior of `create_copy_of` is undefined.

create_from_blueprint(*target: address*, **args*, *value: uint256 = 0*, *raw_args: bool = False*, *code_offset: int = 0*[, *salt: bytes32*]) → address

Copy the code of `target` into memory and execute it as initcode. In other words, this operation interprets the code at `target` not as regular runtime code, but directly as initcode. The `*args` are interpreted as constructor arguments, and are ABI-encoded and included when executing the initcode.

- `target`: Address of the blueprint to invoke
- `*args`: Constructor arguments to forward to the initcode.
- `value`: The wei value to send to the new contract address (Optional, default 0)
- `raw_args`: If `True`, `*args` must be a single `Bytes[...]` argument, which will be interpreted as a raw bytes buffer to forward to the create operation (which is useful for instance, if pre- ABI-encoded data is passed in from elsewhere). (Optional, default `False`)
- `code_offset`: The offset to start the `EXTCODECOPY` from (Optional, default 0)
- `salt`: A `bytes32` value utilized by the deterministic `CREATE2` opcode (Optional, if not supplied, `CREATE` is used)

Returns the address of the created contract. If the create operation fails (for instance, in the case of a `CREATE2` collision), execution will revert. If `code_offset >= target.codesize` (ex. if there is no code at `target`), execution will revert.

```
@external
def foo(blueprint: address) -> address:
    arg1: uint256 = 18
    arg2: String[32] = "some string"
    return create_from_blueprint(blueprint, arg1, arg2, code_offset=1)
```

Note: To properly deploy a blueprint contract, special deploy bytecode must be used. The output of `vyper -f blueprint_bytecode` will produce bytecode which deploys an ERC-5202 compatible blueprint.

Warning: It is recommended to deploy blueprints with the ERC-5202 preamble `0xFE7100` to guard them from being called as regular contracts. This is particularly important for factories where the constructor has side effects (including `SELFDESTRUCT`!), as those could get executed by *anybody* calling the blueprint contract directly. The `code_offset=` kwarg is provided to enable this pattern:

```
@external
def foo(blueprint: address) -> address:
    # `blueprint` is a blueprint contract with some known preamble b"abcd..."
    return create_from_blueprint(blueprint, code_offset=<preamble length>)
```

raw_call(*to: address*, *data: Bytes*, *max_outsize: uint256 = 0*, *gas: uint256 = gasLeft*, *value: uint256 = 0*, *is_delegate_call: bool = False*, *is_static_call: bool = False*, *revert_on_failure: bool = True*) → 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. (Optional, default `0`)
- `gas`: The amount of gas to attach to the call. (Optional, defaults to `msg.gas`).
- `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`)
- `revert_on_failure`: If `True`, the call will revert on a failure, otherwise `success` will be returned (Optional, default `True`)

Note: Returns the data returned by the call as a `Bytes` list, with `max_outsize` as the max length. The actual size of the returned data may be less than `max_outsize`. You can use `len` to obtain the actual size.

Returns nothing if `max_outsize` is omitted or set to `0`.

Returns `success` in a tuple with return value if `revert_on_failure` is set to `False`.

```
@external
@payable
def foo(_target: address) -> Bytes[32]:
    response: Bytes[32] = raw_call(_target, method_id("someMethodName()"), max_
↪outsize=32, value=msg.value)
```

(continues on next page)

(continued from previous page)

```
    return response

@external
@payable
def bar(_target: address) -> Bytes[32]:
    success: bool = False
    response: Bytes[32] = b""
    x: uint256 = 123
    success, response = raw_call(
        _target,
        _abi_encode(x, method_id=method_id("someMethodName(uint256)")),
        max_outsize=32,
        value=msg.value,
        revert_on_failure=False
        )
    assert success
    return response
```

Note: Regarding "forwarding all gas", note that, while Vyper will provide `msg.gas` to the call, in practice, there are some subtleties around forwarding all remaining gas on the EVM which are out of scope of this documentation and could be subject to change. For instance, see the language in EIP-150 around "all but one 64th".

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)
```

raw_revert(*data: Bytes*) → None

Provides low level access to the `REVERT` opcode, reverting execution with the specified data returned.

- `data`: Data representing the error message causing the revert.

```
@external
def foo(_data: Bytes[100]):
    raw_revert(_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 deletes the contract from the blockchain. All non-ether assets associated with this contract are "burned" and the contract is no longer accessible.

Note: This function has been deprecated from version 0.3.8 onwards. The underlying opcode will eventually undergo breaking changes, and its use is not recommended.

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

send(*to: address*, *value: uint256*, *gas: uint256 = 0*) → 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
- `gas`: The amount of gas (the "stipend") to attach to the call. If not set, the stipend defaults to 0.

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

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

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 | uint8*, *r: uint256 | bytes32*, *s: uint256 | bytes32*) → 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 `empty(address)` on error.

Note: Prior to Vyper `0.3.10`, the `ecrecover` function could return an undefined (possibly nonzero) value for invalid inputs to `ecrecover`. For more information, please see GHSA-f5x6-7qgp-jhf3.

```
@external
@view
def foo(hash: bytes32, v: uint8, r:bytes32, s:bytes32) -> address:
    return ecrecover(hash, v, r, s)


@external
@view
def foo(hash: bytes32, v: uint256, r:uint256, s:uint256) -> address:
    return ecrecover(hash, v, r, s)
```

```
>>> ExampleContract.foo(
↪'0x6c9c5e133b8aafb2ea74f524a5263495e7ae5701c7248805f7b511d973dc7055',
     28,
     78616903610408968922803823221221116251138855211764625814919875002740131251724,
     37668412420813231458864536126575229553064045345107737433087067088194345044408
    )
'0x9eE53ad38Bb67d745223a4257D7d48cE973FeB7A'
```

keccak256(*_value*) → bytes32

Return a `keccak256` hash of the given value.

- `_value`: Value to hash. Can be a `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 256-bit output) hash of the given value.

- `_value`: Value to hash. Can be a `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 `bytesM`, `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 (e.g., `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*.

uint2str(*value: unsigned integer*) → String

Returns an unsigned integer's string representation.

- `value`: Unsigned integer to convert.

Returns the string representation of `value`.

```
@external
@view
def foo(b: uint256) -> String[78]:
    return uint2str(b)
```

```
>>> ExampleContract.foo(420)
"420"
```

extract32(*b: Bytes*, *start: uint256*, *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 (`bytesM`, `integer`, or `address`). Defaults to `bytes32`.

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

```
@external
@view
def foo(b: Bytes[32]) -> address:
    return extract32(b, 0, 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

abs(*value: int256*) → int256

Return the absolute value of a signed integer.

- `value`: Integer to return the absolute value of

```
@external
@view
def foo(value: int256) -> int256:
    return abs(value)
```

```
>>> ExampleContract.foo(-31337)
31337
```

ceil(*value: decimal*) → int256

Round a decimal up to the nearest integer.

- `value`: Decimal value to round up

```
@external
@view
def foo(x: decimal) -> int256:
    return ceil(x)
```

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

epsilon(*typename*) → Any

Returns the smallest non-zero value for a decimal type.

- `typename`: Name of the decimal type (currently only `decimal`)

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

```
>>> ExampleContract.foo()
Decimal('1E-10')
```

floor(*value: decimal*) → int256

Round a decimal down to the nearest integer.

- `value`: Decimal value to round down

```
@external
@view
def foo(x: decimal) -> int256:
    return floor(x)
```

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

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

Return the greater 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 of the same type as the input values.

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

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

max_value(*type_*) → numeric

Returns the maximum value of the numeric type specified by `type_` (e.g., `int128`, `uint256`, `decimal`).

```
@external
@view
def foo() -> int256:
    return max_value(int256)
```

```
>>> ExampleContract.foo()
57896044618658097711785492504343953926634992332820282019728792003956564819967
```

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 of the same type as the input values.

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

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

min_value(*type_*) → numeric

Returns the minimum value of the numeric type specified by `type_` (e.g., `int128`, `uint256`, `decimal`).

```
@external
@view
def foo() -> int256:
    return min_value(int256)
```

```
>>> ExampleContract.foo()
-57896044618658097711785492504343953926634992332820282019728792003956564819968
```

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
```

isqrt(*x: uint256*) → uint256

Return the (integer) square root of the provided integer number, using the Babylonian square root algorithm. The rounding mode is to round down to the nearest integer. For instance, `isqrt(101) == 10`.

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

```
>>> ExampleContract.foo(101)
10
```

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

Return the modulo of `(a + b) % c`. Reverts if `c == 0`. As this built-in function is intended to provides access to the underlying `ADDMOD` opcode, all intermediate calculations of this operation are not subject to the `2 ** 256` modulo according to the EVM specifications.

```
@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`. As this built-in function is intended to provides access to the underlying `MULMOD` opcode, all intermediate calculations of this operation are not subject to the `2 ** 256` modulo according to the EVM specifications.

```
@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
```

unsafe_add(*x: integer*, *y: integer*) → integer

Add `x` and `y`, without checking for overflow. `x` and `y` must both be integers of the same type. If the result exceeds the bounds of the input type, it will be wrapped.

```
@external
@view
def foo(x: uint8, y: uint8) -> uint8:
    return unsafe_add(x, y)

@external
```

(continues on next page)

(continued from previous page)

```
@view
def bar(x: int8, y: int8) -> int8:
    return unsafe_add(x, y)
```

```
>>> ExampleContract.foo(1, 1)
2

>>> ExampleContract.foo(255, 255)
254

>>> ExampleContract.bar(127, 127)
-2
```

Note: Performance note: for the native word types of the EVM `uint256` and `int256`, this will compile to a single `ADD` instruction, since the EVM natively wraps addition on 256-bit words.

unsafe_sub(*x: integer*, *y: integer*) → integer

Subtract `x` and `y`, without checking for overflow. `x` and `y` must both be integers of the same type. If the result underflows the bounds of the input type, it will be wrapped.

```
@external
@view
def foo(x: uint8, y: uint8) -> uint8:
    return unsafe_sub(x, y)

@external
@view
def bar(x: int8, y: int8) -> int8:
    return unsafe_sub(x, y)
```

```
>>> ExampleContract.foo(4, 3)
1

>>> ExampleContract.foo(0, 1)
255

>>> ExampleContract.bar(-128, 1)
127
```

Note: Performance note: for the native word types of the EVM `uint256` and `int256`, this will compile to a single `SUB` instruction, since the EVM natively wraps subtraction on 256-bit words.

unsafe_mul(*x: integer*, *y: integer*) → integer

Multiply `x` and `y`, without checking for overflow. `x` and `y` must both be integers of the same type. If the result exceeds the bounds of the input type, it will be wrapped.

```
@external
@view
```

(continues on next page)

(continued from previous page)

```
def foo(x: uint8, y: uint8) -> uint8:
    return unsafe_mul(x, y)

@external
@view
def bar(x: int8, y: int8) -> int8:
    return unsafe_mul(x, y)
```

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

>>> ExampleContract.foo(255, 255)
1

>>> ExampleContract.bar(-128, -128)
0

>>> ExampleContract.bar(127, -128)
-128
```

Note: Performance note: for the native word types of the EVM `uint256` and `int256`, this will compile to a single `MUL` instruction, since the EVM natively wraps multiplication on 256-bit words.

unsafe_div(*x: integer*, *y: integer*) → integer

Divide `x` and `y`, without checking for division-by-zero. `x` and `y` must both be integers of the same type. If the denominator is zero, the result will (following EVM semantics) be zero.

```
@external
@view
def foo(x: uint8, y: uint8) -> uint8:
    return unsafe_div(x, y)

@external
@view
def bar(x: int8, y: int8) -> int8:
    return unsafe_div(x, y)
```

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

>>> ExampleContract.foo(1, 0)
0

>>> ExampleContract.bar(-128, -1)
-128
```

Note: Performance note: this will compile to a single `SDIV` or `DIV` instruction, depending on if the inputs are signed or unsigned (respectively).

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 recent 256 blocks. This function reverts 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 (zero-ed) value of its type. Useful for initializing new memory variables.

- `typename`: Name of the type, except `HashMap[_KeyType, _ValueType]`

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

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

Return the length of a given `Bytes`, `String` or `DynArray[_Type, _Integer]`.

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

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

method_id(*method*, *output_type: type = Bytes[4]*) → Union[Bytes[4], bytes4]

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 `bytes4`). 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()
0xa9059cbb
```

_abi_encode(**args*, *ensure_tuple: bool = True*) → Bytes[<depends on input>]

Takes a variable number of args as input, and returns the ABIv2-encoded bytestring. Used for packing arguments to raw_call, EIP712 and other cases where a consistent and efficient serialization method is needed. Once this function has seen more use we provisionally plan to put it into the `ethereum.abi` namespace.

- `*args`: Arbitrary arguments
- `ensure_tuple`: If set to True, ensures that even a single argument is encoded as a tuple. In other words, `bytes` gets encoded as `(bytes,)`, and `(bytes,)` gets encoded as `((bytes,),)` This is the calling convention for Vyper and Solidity functions. Except for very specific use cases, this should be set to True. Must be a literal.
- `method_id`: A literal hex or Bytes[4] value to append to the beginning of the abi-encoded bytestring.

Returns a bytestring whose max length is determined by the arguments. For example, encoding a `Bytes[32]` results in a `Bytes[64]` (first word is the length of the bytestring variable).

```
@external
@view
def foo() -> Bytes[132]:
    x: uint256 = 1
    y: Bytes[32] = b"234"
    return _abi_encode(x, y, method_id=method_id("foo()"))
```

```
>>> ExampleContract.foo().hex()
"c2985578"
"0000000000000000000000000000000000000000000000000000000000000001"
"0000000000000000000000000000000000000000000000000000000000000040"
"0000000000000000000000000000000000000000000000000000000000000003"
"3233340000000000000000000000000000000000000000000000000000000000"
```

_abi_decode(*b: Bytes*, *output_type: type_*, *unwrap_tuple: bool = True*) → Any

Takes a byte array as input, and returns the decoded values according to the specified output types. Used for unpacking ABIv2-encoded values. Once this function has seen more use we provisionally plan to put it into the `ethereum.abi` namespace.

- b: A byte array of a length that is between the minimum and maximum ABIv2 size bounds of the output type.
- output_type: Name of the output type, or tuple of output types, to be decoded.
- unwrap_tuple: If set to True, the input is decoded as a tuple even if only one output type is specified. In other words, _abi_decode(b, Bytes[32]) gets decoded as (Bytes[32],). This is the convention for ABIv2-encoded values generated by Vyper and Solidity functions. Except for very specific use cases, this should be set to True. Must be a literal.

Returns the decoded value(s), with type as specified by *output_type*.

```
@external
@view
def foo(someInput: Bytes[128]) -> (uint256, Bytes[32]):
    x: uint256 = empty(uint256)
    y: Bytes[32] = empty(Bytes[32])
    x, y =  _abi_decode(someInput, (uint256, Bytes[32]))
    return x, y
```

print(**args*, *hardhat_compat=False*) → None

"prints" the arguments by issuing a static call to the "console" address, 0x000000000000000000636F6E736F6C652E6C6F67. This is supported by some smart contract development frameworks.

The default mode works natively with titanoboa. For hardhat-style frameworks, use hardhat_compat=True).

Note: Issuing of the static call is *NOT* mode-dependent (that is, it is not removed from production code), although the compiler will issue a warning whenever print is used.

CHAPTER

ELEVEN

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 separate 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 used to make external calls, given a contract address:

```
@external
def test(foobar: FooBar):
    foobar.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 casting an address to an interface is possible, e.g. `FooBar(<address_var>)`:

```
foobar_contract: FooBar

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

@external
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
```

(continues on next page)

(continued from previous page)

```
    def query() -> uint256: view
    def update(): nonpayable
    def pay(): payable

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

Vyper offers the option to set the following additional keyword arguments when making external calls:

Keyword	Description
`gas`	Specify gas value for the call
`value`	Specify amount of ether sent with the call
`skip_contract_check`	Drop `EXTCODESIZE` and `RETURNDATASIZE` checks
`default_return_value`	Specify a default return value if no value is returned

The `default_return_value` parameter can be used to handle ERC20 tokens affected by the missing return value bug in a way similar to OpenZeppelin's `safeTransfer` for Solidity:

```
ERC20(USDT).transfer(msg.sender, 1, default_return_value=True) # returns True
ERC20(USDT).transfer(msg.sender, 1) # reverts because nothing returned
```

Warning: When `skip_contract_check=True` is used and the called function returns data (ex.: `x: uint256 = SomeContract.foo(skip_contract_check=True)`, no guarantees are provided by the compiler as to the validity of the returned value. In other words, it is undefined behavior what happens if the called contract did not exist. In particular, the returned value might point to garbage memory. It is therefore recommended to only use `skip_contract_check=True` to call contracts which have been manually ensured to exist at the time of the call.

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 its 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 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.

Note: Interfaces that implement functions with return values that require an upper bound (e.g. `Bytes`, `DynArray`, or `String`), the upper bound defined in the interface represents the lower bound of the implementation. Assuming a function `my_func` returns a value `String[1]` in the interface, this would mean for the implementation function of `my_func` that the return value must have **at least** length 1. This behavior might change in the future.

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
```

(continues on next page)

(continued from previous page)

```
# 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

TWELVE

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 */);
```

(continues on next page)

(continued from previous page)

```
// 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

THIRTEEN

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)
    @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
`@license`	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
`@custom:...`	Custom tag, semantics is application-defined	contract, 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"
    }
  },
  "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

FOURTEEN

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 `IR` 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
```

Storage Layout

To display the default storage layout for a contract:

```
$ vyper -f layout yourFileName.vy
```

This outputs a JSON object detailing the locations for all state variables as determined by the compiler.

To override the default storage layout for a contract:

```
$ vyper --storage-layout-file storageLayout.json yourFileName.vy
```

The input to the `--storage-layout-file` flag must match the format of the `.storage_layout` field from the `vyper -f layout` command.

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 explicitly 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 Try VyperLang!

Try VyperLang! is a JupterHub instance hosted by the Vyper team as a sandbox for developing and testing contracts in Vyper. It requires github for login, and supports deployment via the browser.

14.2.2 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 Compiler Optimization Modes

The vyper CLI tool accepts an optimization mode `"none"`, `"codesize"`, or `"gas"` (default). It can be set using the `--optimize` flag. For example, invoking `vyper --optimize codesize MyContract.vy` will compile the contract, optimizing for code size. As a rough summary of the differences between gas and codesize mode, in gas optimized mode, the compiler will try to generate bytecode which minimizes gas (up to a point), including:

- using a sparse selector table which optimizes for gas over codesize
- inlining some constants, and
- trying to unroll some loops, especially for data copies.

In codesize optimized mode, the compiler will try hard to minimize codesize by

- using a dense selector table
- out-lining code, and
- using more loops for data copies.

14.4 Setting the Target EVM Version

When you compile your contract code, you can specify the target Ethereum Virtual Machine version to compile for, to access or avoid particular features. You can specify the version either with a source code pragma or as a compiler option. It is recommended to use the compiler option when you want flexibility (for instance, ease of deploying across different chains), and the source code pragma when you want bytecode reproducibility (for instance, when verifying code on a block explorer).

Note: If the evm version specified by the compiler options conflicts with the source code pragma, an exception will be raised and compilation will not continue.

For instance, the adding the following pragma to a contract indicates that it should be compiled for the "shanghai" fork of the EVM.

```
#pragma evm-version shanghai
```

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

When compiling via the `vyper` CLI, you can specify the EVM version option using the `--evm-version` flag:

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

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

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

14.4.1 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.

istanbul

- 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`

berlin

- Gas estimates changed for `EXTCODESIZE`, `EXTCODECOPY`, `EXTCODEHASH`, `SLOAD`, `SSTORE`, `CALL`, `CALLCODE`, `DELEGATECALL` and `STATICCALL`
- Functions marked with `@nonreentrant` are protected with different values (3 and 2) than contracts targeting pre-berlin.
- `BASEFEE` is accessible via `block.basefee`

paris

- `block.difficulty` is deprecated in favor of its new alias, `block.prevrandao`.

shanghai (*default*)

- The `PUSH0` opcode is automatically generated by the compiler instead of `PUSH1 0`

cancun (*experimental*)

- The `transient` keyword allows declaration of variables which live in transient storage
- Functions marked with `@nonreentrant` are protected with TLOAD/TSTORE instead of SLOAD/SSTORE
- The `MCOPY` opcode will be generated automatically by the compiler for most memory operations.

14.5 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.

14.5.1 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": "shanghai",  // EVM version to compile for. Can be istanbul,␣
↪berlin, paris, shanghai (default) or cancun (experimental!).
        // optional, optimization mode
        // defaults to "gas". can be one of "gas", "codesize", "none",
        // false  and true (the last two are for backwards compatibility).
        "optimize": "gas",
        // optional, whether or not the bytecode should include Vyper's signature
        // defaults to true
        "bytecodeMetadata": true,
        // 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 - 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)
```

(continues on next page)

(continued from previous page)

```
        //    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
        }
    }
}
```

14.5.2 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": {},
```

(continues on next page)

(continued from previous page)

```
        }
    },
    // 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": "",
                // 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

FIFTEEN

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
```

(continues on next page)

(continued from previous page)

```
---> 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

SIXTEEN

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 deploy 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

SEVENTEEN

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 Ethereum Python Dev Discord `#brownie` 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

storedData: public(int128)

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

@external
```

(continues on next page)

(continued from previous page)

```
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


@pytest.fixture
def adv_storage_contract(AdvancedStorage, accounts):
```

(continues on next page)

(continued from previous page)

```
    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 json

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

from vyper import compiler
from vyper.ast.grammar import parse_vyper_source
```

(continues on next page)

(continued from previous page)

```
from vyper.compiler.settings import Settings


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) + 500000})
        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("__")]

        try:
```

(continues on next page)

(continued from previous page)

```
            fn_names = [fn["name"] for fn in self._classic_contract.functions._functions]
        except web3.exceptions.NoABIFunctionsFound:
            fn_names = []

        for fn_name in fn_names:
            # Override namespace collisions
            if fn_name in protected_fn_names:
                raise AttributeError(f"{fn_name} is protected!")
            else:
                _classic_method = getattr(self._classic_contract.functions, fn_name)
                _concise_method = method_class(
                    _classic_method, self._classic_contract._return_data_normalizers
                )
            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(scope="module")
def tester():
    # set absurdly high gas limit so that london basefee never adjusts
    # (note: 2**63 - 1 is max that evm allows)
    custom_genesis = PyEVMBackend._generate_genesis_params(overrides={"gas_limit": 10**10})
    custom_genesis["base_fee_per_gas"] = 0
    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(scope="module")
def w3(tester):
    w3 = Web3(EthereumTesterProvider(tester))
    w3.eth.set_gas_price_strategy(zero_gas_price_strategy)
    return w3
```

(continues on next page)

(continued from previous page)

```
def _get_contract(w3, source_code, optimize, *args, override_opt_level=None, **kwargs):
    settings = Settings()
    settings.evm_version = kwargs.pop("evm_version", None)
    settings.optimize = override_opt_level or optimize
    out = compiler.compile_code(
        source_code,
        # test that metadata and natspecs get generated
        ["abi", "bytecode", "metadata", "userdoc", "devdoc"],
        settings=settings,
        interface_codes=kwargs.pop("interface_codes", None),
        show_gas_estimates=True,  # Enable gas estimates for testing
    )
    parse_vyper_source(source_code)  # Test grammar.
    json.dumps(out["metadata"])  # test metadata is json serializable
    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)
    tx_hash = deploy_transaction.transact(tx_info)
    address = w3.eth.get_transaction_receipt(tx_hash)["contractAddress"]
    return w3.eth.contract(address, abi=abi, bytecode=bytecode,␣
↪ContractFactoryClass=VyperContract)


def _deploy_blueprint_for(w3, source_code, optimize, initcode_prefix=b"", **kwargs):
    settings = Settings()
    settings.evm_version = kwargs.pop("evm_version", None)
    settings.optimize = optimize
    out = compiler.compile_code(
        source_code,
        ["abi", "bytecode"],
        interface_codes=kwargs.pop("interface_codes", None),
        settings=settings,
        show_gas_estimates=True,  # Enable gas estimates for testing
    )
    parse_vyper_source(source_code)  # Test grammar.
    abi = out["abi"]
    bytecode = HexBytes(initcode_prefix) + HexBytes(out["bytecode"])
    bytecode_len = len(bytecode)
    bytecode_len_hex = hex(bytecode_len)[2:].rjust(4, "0")
    # prepend a quick deploy preamble
    deploy_preamble = HexBytes("61" + bytecode_len_hex + "3d81600a3d39f3")
    deploy_bytecode = HexBytes(deploy_preamble) + bytecode

    deployer_abi = []  # just a constructor
    c = w3.eth.contract(abi=deployer_abi, bytecode=deploy_bytecode)
    deploy_transaction = c.constructor()
    tx_info = {"from": w3.eth.accounts[0], "value": 0, "gasPrice": 0}

```

(continues on next page)

(continued from previous page)

```
    tx_hash = deploy_transaction.transact(tx_info)
    address = w3.eth.get_transaction_receipt(tx_hash)["contractAddress"]

    # sanity check
    assert w3.eth.get_code(address) == bytecode, (w3.eth.get_code(address), bytecode)

    def factory(address):
        return w3.eth.contract(
            address, abi=abi, bytecode=bytecode, ContractFactoryClass=VyperContract
        )

    return w3.eth.contract(address, bytecode=deploy_bytecode), factory


@pytest.fixture(scope="module")
def deploy_blueprint_for(w3, optimize):
    def deploy_blueprint_for(source_code, *args, **kwargs):
        return _deploy_blueprint_for(w3, source_code, optimize, *args, **kwargs)

    return deploy_blueprint_for


@pytest.fixture(scope="module")
def get_contract(w3, optimize):
    def get_contract(source_code, *args, **kwargs):
        return _get_contract(w3, source_code, optimize, *args, **kwargs)

    return get_contract


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

    return get_logs


@pytest.fixture(scope="module")
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:
            # TODO test equality
            assert exc_text in str(excinfo.value), (exc_text, 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
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):
  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(scope="module")
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:
            # TODO test equality
            assert exc_text in str(excinfo.value), (exc_text, 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.get_transaction_receipt(tx_hash)
```

(continues on next page)

(continued from previous page)

```
        return c._classic_contract.events[event_name]().process_receipt(tx_receipt)

    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()
        # 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}),
```

(continues on next page)

(continued from previous page)

```
        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

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

CHAPTER

EIGHTEEN

OTHER RESOURCES AND LEARNING MATERIAL

Vyper has an active community. You can find third party tutorials, examples, courses and other learning material.

18.1 General

- Ape Academy - Learn how to build vyper projects by ApeWorX
- More Vyper by Example by Smart Contract Engineer
- Vyper cheat Sheet
- Vyper Hub for development
- Vyper greatest hits smart contract examples

18.2 Frameworks and tooling

- ApeWorX - The Ethereum development framework for Python Developers, Data Scientists, and Security Professionals
- Foundry x Vyper - Foundry template to compile Vyper contracts
- Snekmate - Vyper smart contract building blocks
- Serpentor - A set of smart contracts tools for governance
- Smart contract development frameworks and tools for Vyper on Ethreum.org

18.3 Security

- VyperPunk - learn to secure and hack Vyper smart contracts
- VyperExamples - Vyper vulnerability examples

18.4 Conference presentations

- Vyper Smart Contract Programming Language by Patrick Collins (2022, 30 mins)
- Python and DeFi by Curve Finance (2022, 15 mins)
- My experience with Vyper over the years by Benjamin Scherrey (2022, 15 mins)
- Short introduction to Vyper by Edison Que (3 mins)

18.5 Unmaintained

These resources have not been updated for a while, but may still offer interesting content.

- Awesome Vyper curated resources
- Brownie - Python framework for developing smart contracts (deprecated)

CHAPTER

NINETEEN

RELEASE NOTES

19.1 v0.3.10 (“Black Adder”)

19.1.1 Date released: 2023-10-04

v0.3.10 is a performance focused release. It adds a `codesize` optimization mode (#3493), adds new vyper-specific `#pragma` directives (#3493), uses Cancun's `MCOPY` opcode for some compiler generated code (#3483), and generates selector tables which now feature O(1) performance (#3496).

Breaking changes:

- add runtime code layout to initcode (#3584)
- drop evm versions through istanbul (#3470)
- remove vyper signature from runtime (#3471)
- only allow valid identifiers to be nonreentrant keys (#3605)

Non-breaking changes and improvements:

- O(1) selector tables (#3496)
- implement bound= in ranges (#3537, #3551)
- add optimization mode to vyper compiler (#3493)
- improve batch copy performance (#3483, #3499, #3525)

Notable fixes:

- fix `ecrecover()` behavior when signature is invalid (GHSA-f5x6-7qgp-jhf3, #3586)
- fix: order of evaluation for some builtins (#3583, #3587)
- fix: memory allocation in certain builtins using `msize` (#3610)
- fix: `_abi_decode()` input validation in certain complex expressions (#3626)
- fix: pycryptodome for arm builds (#3485)
- let params of internal functions be mutable (#3473)
- typechecking of folded builtins in (#3490)

- update tload/tstore opcodes per latest 1153 EIP spec (#3484)
- fix: raw_call type when max_outsize=0 is set (#3572)
- fix: implements check for indexed event arguments (#3570)
- fix: type-checking for `_abi_decode()` arguments (#3626)

Other docs updates, chores and fixes:

- relax restrictions on internal function signatures (#3573)
- note on security advisory in release notes for versions `0.2.15`, `0.2.16`, and `0.3.0` (#3553)
- fix: yanked version in release notes (#3545)
- update release notes on yanked versions (#3547)
- improve error message for conflicting methods IDs (#3491)
- document epsilon builtin (#3552)
- relax version pragma parsing (#3511)
- fix: issue with finding installed packages in editable mode (#3510)
- add note on security advisory for `ecrecover` in docs (#3539)
- add `asm` option to cli help (#3585)
- add message to error map for repeat range check (#3542)
- fix: public constant arrays (#3536)

19.2 v0.3.9 ("Common Adder")

Date released: 2023-05-29

This is a patch release fix for v0.3.8. @bout3fiddy discovered a codesize regression for blueprint contracts in v0.3.8 which is fixed in this release. @bout3fiddy also discovered a runtime performance (gas) regression for default functions in v0.3.8 which is fixed in this release.

Fixes:

- initcode codesize blowup (#3450)
- add back global calldatasize check for contracts with default fn (#3463)

19.3 v0.3.8

Date released: 2023-05-23

Non-breaking changes and improvements:

- `transient` storage keyword (#3373)
- ternary operators (#3398)
- `raw_revert()` builtin (#3136)
- shift operators (#3019)

- make `send()` gas stipend configurable (#3158)
- use new `push0` opcode (#3361)
- python 3.11 support (#3129)
- drop support for python 3.8 and 3.9 (#3325)
- build for `aarch64` (#2687)

Note that with the addition of `push0` opcode, `shanghai` is now the default compilation target for vyper. When deploying to a chain which does not support `shanghai`, it is recommended to set `--evm-version` to `paris`, otherwise it could result in hard-to-debug errors.

Major refactoring PRs:

- refactor front-end type system (#2974)
- merge front-end and codegen type systems (#3182)
- simplify `GlobalContext` (#3209)
- remove `FunctionSignature` (#3390)

Notable fixes:

- assignment when rhs is complex type and references lhs (#3410)
- uninitialized immutable values (#3409)
- success value when mixing `max_outsize=0` and `revert_on_failure=False` (GHSA-w9g2-3w7p-72g9)
- block certain kinds of storage allocator overflows (GHSA-mgv8-gggw-mrg6)
- store-before-load when a dynarray appears on both sides of an assignment (GHSA-3p37-3636-q8wv)
- bounds check for loops of the form `for i in range(x, x+N)` (GHSA-6r8q-pfpv-7cgj)
- alignment of call-site posargs and kwargs for internal functions (GHSA-ph9x-4vc9-m39g)
- batch nonpayable check for default functions calldatasize < 4 (#3104, #3408, cf. GHSA-vxmm-cwh2-q762)

Other docs updates, chores and fixes:

- call graph stability (#3370)
- fix `vyper-serve` output (#3338)
- add `custom:` natspec tags (#3403)
- add missing pc maps to `vyper_json` output (#3333)
- fix constructor context for internal functions (#3388)
- add deprecation warning for `selfdestruct` usage (#3372)
- add bytecode metadata option to vyper-json (#3117)
- fix compiler panic when a `break` is outside of a loop (#3177)
- fix complex arguments to builtin functions (#3167)
- add support for all types in ABI imports (#3154)
- disable uadd operator (#3174)
- block bitwise ops on decimals (#3219)
- raise `UNREACHABLE` (#3194)
- allow enum as mapping key (#3256)

- block boolean `not` operator on numeric types (#3231)
- enforce that loop's iterators are valid names (#3242)
- fix typechecker hotspot (#3318)
- rewrite typechecker journal to handle nested commits (#3375)
- fix missing pc map for empty functions (#3202)
- guard against iterating over empty list in for loop (#3197)
- skip enum members during constant folding (#3235)
- bitwise `not` constant folding (#3222)
- allow accessing members of constant address (#3261)
- guard against decorators in interface (#3266)
- fix bounds for decimals in some builtins (#3283)
- length of literal empty bytestrings (#3276)
- block `empty()` for HashMaps (#3303)
- fix type inference for empty lists (#3377)
- disallow logging from `pure`, `view` functions (#3424)
- improve optimizer rules for comparison operators (#3412)
- deploy to ghcr on push (#3435)
- add note on return value bounds in interfaces (#3205)
- index `id` param in `URI` event of `ERC1155ownable` (#3203)
- add missing `asset` function to `ERC4626` built-in interface (#3295)
- clarify `skip_contract_check=True` can result in undefined behavior (#3386)
- add `custom` NatSpec tag to docs (#3404)
- fix `uint256_addmod` doc (#3300)
- document optional kwargs for external calls (#3122)
- remove `slice()` length documentation caveats (#3152)
- fix docs of `blockhash` to reflect revert behaviour (#3168)
- improvements to compiler error messages (#3121, #3134, #3312, #3304, #3240, #3264, #3343, #3307, #3313 and #3215)

These are really just the highlights, as many other bugfixes, docs updates and refactoring (over 150 pull requests!) made it into this release! For the full list, please see the changelog. Special thanks to contributions from @tserg, @trocher, @z80dev, @emc415 and @benber86 in this release!

New Contributors:

- @omahs made their first contribution in (#3128)
- @ObiajuluM made their first contribution in (#3124)
- @trocher made their first contribution in (#3134)
- @ozmium22 made their first contribution in (#3149)
- @ToonVanHove made their first contribution in (#3168)

- @emc415 made their first contribution in (#3158)
- @lgtm-com made their first contribution in (#3147)
- @tdurieux made their first contribution in (#3224)
- @victor-ego made their first contribution in (#3263)
- @miohtama made their first contribution in (#3257)
- @kelvinfan001 made their first contribution in (#2687)

19.4 v0.3.7

Date released: 2022-09-26

Breaking changes:

- chore: drop python 3.7 support (#3071)
- fix: relax check for statically sized calldata (#3090)

Non-breaking changes and improvements:

- fix: assert description in `Crowdfund.finalize()` (#3058)
- fix: change mutability of example ERC721 interface (#3076)
- chore: improve error message for non-checksummed address literal (#3065)
- feat: `isqrt()` builtin (#3074) (#3069)
- feat: add `block.prevrandao` as alias for `block.difficulty` (#3085)
- feat: `epsilon()` builtin (#3057)
- feat: extend ecrecover signature to accept additional parameter types (#3084)
- feat: allow constant and immutable variables to be declared public (#3024)
- feat: optionally disable metadata in bytecode (#3107)

Bugfixes:

- fix: empty nested dynamic arrays (#3061)
- fix: foldable builtin default args in imports (#3079) (#3077)

Additional changes and improvements:

- doc: update broken links in SECURITY.md (#3095)
- chore: update discord link in docs (#3031)
- fix: broken links in various READMEs (#3072)
- chore: fix compile warnings in examples (#3033)
- feat: append lineno to the filename in error messages (#3092)
- chore: migrate lark grammar (#3082)
- chore: loosen and upgrade semantic version (#3106)

New Contributors

- @emilianobonassi made their first contribution in #3107

- @unparalleled-js made their first contribution in #3106
- @pcaversaccio made their first contribution in #3085
- @nfwsncked made their first contribution in #3058
- @z80 made their first contribution in #3057
- @Benny made their first contribution in #3024
- @cairo made their first contribution in #3072
- @fiddy made their first contribution in #3069

Special thanks to returning contributors @tserg, @pandadefi, and @delaaxe.

19.5 v0.3.6

Date released: 2022-08-07

Bugfixes:

- Fix `in` expressions when list members are variables (#3035)

19.6 v0.3.5

THIS RELEASE HAS BEEN PULLED

Date released: 2022-08-05

Non-breaking changes and improvements:

- Add blueprint deployer output format (#3001)
- Allow arbitrary data to be passed to `create_from_blueprint` (#2996)
- Add CBOR length to bytecode for decoders (#3010)
- Fix compiler panic when accessing enum storage vars via `self` (#2998)
- Fix: allow `empty()` in constant definitions and in default argument position (#3008)
- Fix: disallow `self` address in pure functions (#3027)

19.7 v0.3.4

Date released: 2022-07-27

Non-breaking changes and improvements:

- Add enum types (#2874, #2915, #2925, #2977)
- Add `_abi_decode` builtin (#2882)
- Add `create_from_blueprint` and `create_copy_of` builtins (#2895)
- Add `default_return_value` kwarg for calls (#2839)
- Add `min_value` and `max_value` builtins for numeric types (#2935)
- Add `uint2str` builtin (#2879)

- Add vyper signature to bytecode (#2860)

Other fixes and improvements:

- Call internal functions from constructor (#2496)
- Arithmetic for new int types (#2843)
- Allow `msg.data` in `raw_call` without `slice` (#2902)
- Per-method calldatasize checks (#2911)
- Type inference and annotation of arguments for builtin functions (#2817)
- Allow varargs for `print` (#2833)
- Add `error_map` output format for tooling consumption (#2939)
- Multiple evaluation of contract address in call (GHSA-4v9q-cgpw-cf38)
- Improve ast output (#2824)
- Allow `@nonreentrant` on view functions (#2921)
- Add `shift()` support for signed integers (#2964)
- Enable dynarrays of strings (#2922)
- Fix off-by-one bounds check in certain safepow cases (#2983)
- Optimizer improvements (#2647, #2868, #2914, #2843, #2944)
- Reverse order in which exceptions are reported (#2838)
- Fix compile-time blowup for large contracts (#2981)
- Rename `vyper-ir` binary to `fang` (#2936)

Many other small bugfixes, optimizations and refactoring also made it into this release! Special thanks to @tserg and @pandadefi for contributing several important bugfixes, refactoring and features to this release!

19.8 v0.3.3

Date released: 2022-04-22

This is a bugfix release. It patches an off-by-one error in the storage allocation mechanism for dynamic arrays reported by @haltman-at in #2820

Other fixes and improvements:

- Add a `print` built-in which allows printing debugging messages in hardhat. (#2818)
- Fix various error messages (#2798, #2805)

19.9 v0.3.2

Date released: 2022-04-17

Breaking changes:

- Increase the bounds of the `decimal` type (#2730)
- Generalize and simplify the semantics of the `convert` builtin (#2694)
- Restrict hex and bytes literals (#2736, #2872)

Non-breaking changes and improvements:

- Implement dynamic arrays (#2556, #2606, #2615)
- Support all ABIv2 integer and bytes types (#2705)
- Add storage layout override mechanism (#2593)
- Support `<address>.code` attribute (#2583)
- Add `tx.gasprice` builtin (#2624)
- Allow structs as constant variables (#2617)
- Implement `skip_contract_check` kwarg (#2551)
- Support EIP-2678 ethPM manifest files (#2628)
- Add `metadata` output format (#2597)
- Allow `msg.*` variables in internal functions (#2632)
- Add `unsafe_` arithmetic builtins (#2629)
- Add subroutines to Vyper IR (#2598)
- Add `select` opcode to Vyper IR (#2690)
- Allow lists of any type as loop variables (#2616)
- Improve suggestions in error messages (#2806)

Notable Fixes:

- Clamping of returndata from external calls in complex expressions (GHSA-4mrx-6fxm-8jpg, GHSA-j2x6-9323-fp7h)
- Bytestring equality for (N<=32) (GHSA-7vrm-3jc8-5wwm)
- Typechecking of constant variables (#2580, #2603)
- Referencing immutables in constructor (#2627)
- Arrays of interfaces in for loops (#2699)

Lots of optimizations, refactoring and other fixes made it into this release! For the full list, please see the changelog.

Special thanks to @tserg for typechecker fixes and significant testing of new features! Additional contributors to this release include @abdullathedruid, @hi-ogawa, @skellet0r, @fubuloubu, @onlymaresia, @SwapOperator, @hitsuzen-eth, @Sud0u53r, @davidhq.

19.10 v0.3.1

Date released: 2021-12-01

Breaking changes:

- Disallow changes to decimal precision when used as a library (#2479)

Non-breaking changes and improvements:

- Add immutable variables (#2466)
- Add uint8 type (#2477)
- Add gaslimit and basefee env variables (#2495)
- Enable checkable raw_call (#2482)
- Propagate revert data when external call fails (#2531)
- Improve LLL annotations (#2486)
- Optimize short-circuiting boolean operations (#2467, #2493)
- Optimize identity precompile usage (#2488)
- Remove loaded limits for int128 and address (#2506)
- Add machine readable ir_json format (#2510)
- Optimize raw_call for the common case when the input is in memory (#2481)
- Remove experimental OVM transpiler (#2532)
- Add CLI flag to disable optimizer (#2522)
- Add docs for LLL syntax and semantics (#2494)

Fixes:

- Allow non-constant revert reason strings (#2509)
- Allow slices of complex expressions (#2500)
- Remove seq_unchecked from LLL codegen (#2485)
- Fix external calls with default parameters (#2526)
- Enable lists of structs as function arguments (#2515)
- Fix .balance on constant addresses (#2533)
- Allow variable indexing into constant/literal arrays (#2534)
- Fix allocation of unused storage slots (#2439, #2514)

Special thanks to @skellet0r for some major features in this release!

19.11 v0.3.0

A critical security vulnerability has been discovered in this version and we strongly recommend using version 0.3.1 or higher. For more information, please see the Security Advisory GHSA-5824-cm3x-3c38.

Date released: 2021-10-04

Breaking changes:

- Change ABI encoding of single-struct return values to be compatible with Solidity (#2457)
- Drop Python 3.6 support (#2462)

Non-breaking changes and improvements:

- Rewrite internal calling convention (#2447)
- Allow any ABI-encodable type as function arguments and return types (#2154, #2190)
- Add support for deterministic deployment of minimal proxies using CREATE2 (#2460)
- Optimize code for certain copies (#2468)
- Add -o CLI flag to redirect output to a file (#2452)
- Other docs updates (#2450)

Fixes:

- _abi_encode builtin evaluates arguments multiple times (#2459)
- ABI length is too short for nested tuples (#2458)
- Returndata is not clamped for certain numeric types (#2454)
- __default__ functions do not respect nonreentrancy keys (#2455)
- Clamps for bytestrings in initcode are broken (#2456)
- Missing clamps for decimal args in external functions (GHSA-c7pr-343r-5c46)
- Memory corruption when returning a literal struct with a private function call inside of it (GHSA-xv8x-pr4h-73jv)

Special thanks to contributions from @skellet0r and @benjyz for this release!

19.12 v0.2.16

A critical security vulnerability has been discovered in this version and we strongly recommend using version 0.3.1 or higher. For more information, please see the Security Advisory GHSA-5824-cm3x-3c38.

Date released: 2021-08-27

Non-breaking changes and improvements:

- Expose _abi_encode as a user-facing builtin (#2401)
- Export the storage layout as a compiler output option (#2433)
- Add experimental OVM backend (#2416)
- Allow any ABI-encodable type as event arguments (#2403)
- Optimize int128 clamping (#2411)
- Other docs updates (#2405, #2422, #2425)

Fixes:

- Disallow nonreentrant decorator on constructors (#2426)
- Fix bounds checks when handling msg.data (#2419)
- Allow interfaces in lists, structs and maps (#2397)
- Fix trailing newline parse bug (#2412)

Special thanks to contributions from @skellet0r, @sambacha and @milancermak for this release!

19.13 v0.2.15

A critical security vulnerability has been discovered in this version and we strongly recommend using version 0.3.1 or higher. For more information, please see the Security Advisory GHSA-5824-cm3x-3c38.

Date released: 23-07-2021

Non-breaking changes and improvements - Optimization when returning nested tuples (#2392)

Fixes: - Annotated kwargs for builtins (#2389) - Storage slot allocation bug (#2391)

19.14 v0.2.14

THIS RELEASE HAS BEEN PULLED

Date released: 20-07-2021

Non-breaking changes and improvements: - Reduce bytecode by sharing code for clamps (#2387)

Fixes: - Storage corruption from re-entrancy locks (#2379)

19.15 v0.2.13

THIS RELEASE HAS BEEN PULLED

Date released: 06-07-2021

Non-breaking changes and improvements:

- Add the `abs` builtin function (#2356)
- Streamline the location of arrays within storage (#2361)

19.16 v0.2.12

Date released: 16-04-2021

This release fixes a memory corruption bug (#2345) that was introduced in the v0.2.x series and was not fixed in VVE-2020-0004. Read about it further in VVE-2021-0001.

Non-breaking changes and improvements:

- Optimize `calldataload` (#2352)
- Add the `int256` signed integer type (#2351)

- EIP2929 opcode repricing and Berlin support (#2350)
- Add `msg.data` environment variable #2343 (#2343)
- Full support for Python 3.9 (#2233)

19.17 v0.2.11

Date released: 27-02-2021

This is a quick patch release to fix a memory corruption bug that was introduced in v0.2.9 (#2321) with excessive memory deallocation when releasing internal variables

19.18 v0.2.10

THIS RELEASE HAS BEEN PULLED

Date released: 17-02-2021

This is a quick patch release to fix incorrect generated ABIs that was introduced in v0.2.9 (#2311) where storage variable getters were incorrectly marked as `nonpayable` instead of `view`

19.19 v0.2.9

THIS RELEASE HAS BEEN PULLED

Date released: 16-02-2021

Non-breaking changes and improvements: - Add license to wheel, Anaconda support (#2265) - Consider events during type-check with *implements:* (#2283) - Refactor ABI generation (#2284) - Remove redundant checks in parser/signatures (#2288) - Streamling ABI-encoding logic for tuple return types (#2302) - Optimize function ordering within bytecode (#2303) - Assembly-level optimizations (#2304) - Optimize nonpayable assertion (#2307) - Optimize re-entrancy locks (#2308)

Fixes: - Change forwarder proxy bytecode to ERC-1167 (#2281) - Reserved keywords check update (#2286) - Incorrect type-check error in literal lists (#2309)

Tons of Refactoring work courtesy of (@iamdefinitelyahuman)!

19.20 v0.2.8

Date released: 04-12-2020

Non-breaking changes and improvements:

- AST updates to provide preliminary support for Python 3.9 (#2225)
- Support for the `not in` comparator (#2232)
- Lift restriction on calldata variables shadowing storage variables (#2226)
- Optimize `shift` bytecode when 2nd arg is a literal (#2201)
- Warn when EIP-170 size limit is exceeded (#2208)

Fixes:

- Allow use of `slice` on a calldata `bytes32` (#2227)
- Explicitly disallow iteration of a list of structs (#2228)
- Improved validation of address checksums (#2229)
- Bytes are always represented as hex within the AST (#2231)
- Allow `empty` as an argument within a function call (#2234)
- Allow `empty` static-sized array as an argument within a `log` statement (#2235)
- Compile-time issue with `Bytes` variables as a key in a mapping (#2239)

19.21 v0.2.7

Date released: 10-14-2020

This is a quick patch release to fix a runtime error introduced in `v0.2.6` (#2188) that could allow for memory corruption under certain conditions.

Non-breaking changes and improvements:

- Optimizations around `assert` and `raise` (#2198)
- Simplified internal handling of memory variables (#2194)

Fixes:

- Ensure internal variables are always placed sequentially within memory (#2196)
- Bugfixes around memory de-allocation (#2197)

19.22 v0.2.6

THIS RELEASE HAS BEEN PULLED

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)

19.23 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)

19.24 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)

19.25 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)

19.26 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)

19.27 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.

19.28 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.

19.29 v0.1.0-beta.16

Date released: 09-01-2020

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

19.30 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!

19.31 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)

19.32 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 uninitialized 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)

19.33 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!

19.34 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!

19.35 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.

19.36 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)

19.37 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

TWENTY

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.

20.1 Types of Contributions

In particular, we need help in the following areas:

- Improving the documentation
- Responding to questions from other users on StackExchange and Discussions
- Add to the discussions on the Vyper (Smart Contract Programming Language) Discord
- Suggesting Improvements
- Fixing and responding to Vyper's GitHub issues

20.2 How to Suggest Improvements

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

20.3 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.

20.4 Fix Bugs

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

20.5 Style Guide

Our *style guide* outlines best practices for the Vyper repository. Please ask us on the Vyper (Smart Contract Programming Language) Discord `#compiler-dev` channel if you have questions about anything that is not outlined in the style guide.

20.6 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.

20.6.1 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 Vyper (Smart Contract Programming Language) Discord `#compiler-dev` 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

TWENTYONE

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.

21.1 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.

21.2 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.

21.2.1 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.

21.2.2 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

# Bad, do not import beyond the root namespace
from vyper.ast.annotation import annotate_python_ast
```

21.2.3 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.

21.2.4 Strings

Strings substitutions **should** be performed via formatted string literals rather than the `str.format` method or other techniques.

21.2.5 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.

21.3 Tests

We use the pytest framework for testing, and *eth-tester* for our local development chain.

21.3.1 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.

21.3.2 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/`.

21.3.3 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.

21.3.4 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.

21.4 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.

21.4.1 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.

21.4.2 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.

21.4.3 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: `#`, `=`, `-`, `*`, `^`.

21.5 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.

21.6 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.

21.6.1 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.

21.6.2 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.

Here's an example commit message adhering to the above practices:

```
Summarize changes in around 50 characters or less

More detailed explanatory text, if necessary. Wrap it to about 72
characters or so. In some contexts, the first line is treated as the
subject of the commit and the rest of the text as the body. The
blank line separating the summary from the body is critical (unless
you omit the body entirely); various tools like `log`, `shortlog`
and `rebase` can get confused if you run the two together.

Explain the problem that this commit is solving. Focus on why you
are making this change as opposed to how (the code explains that).
Are there side effects or other unintuitive consequences of this
change? Here's the place to explain them.

Further paragraphs come after blank lines.

 - Bullet points are okay, too

 - Typically a hyphen or asterisk is used for the bullet, preceded
   by a single space, with blank lines in between, but conventions
   vary here

If you use an issue tracker, put references to them at the bottom,
like this:

Resolves: #XXX
See also: #XXY, #XXXZ
```

CHAPTER

TWENTYTWO

VYPER VERSIONING GUIDELINE

22.1 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 and its 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 seamlessly with later versions, or that they will be reasonably informed when this isn't possible. The Vyper package itself and its 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.

22.2 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.

22.2.1 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

22.2.2 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

22.2.3 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 its 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

22.2.4 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 vulnerabilities, 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*.

22.2.5 Vyper *Next*

There may be multiple Major versions in the 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.

22.3 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.

22.4 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

Symbols

A

B

C

D

E

F

I

J

K

L

M

N

O

P

R

S

T

U

V

Z